Proving the Correctness of the BTTF Wave Algorithm for Distributed Termination Detection

Michel Degenhardt

Studentnumber: 1397257
Year: 2014
Field of Study: Computer Science

Department of Computer Science
Vrije Universiteit
De Boelelaan 1081a
1081 HV Amsterdam

Abstract

The BTTF Wave algorithm is an algorithm designed to detect if a computation in a distributed system has terminated. This algorithm adds information to messages of the computation in order to allow the confirmation of the receipt of multiple messages by a single confirmation message. By doing this, the algorithm achieves the optimal message complexity O(N) in the worst case, and significant better message complexity, up to O(1), in the average and best case scenario's. In this paper, we provide both an informal and a formal description of this algorithm, as well as a proof of its correctness.

1 *Introduction*

Termination detection in a distributed system is a theoretical problem that has been a topic of interest throughout the past 25 years. First introduced independently by Dijkstra and Scholten in [7] and by Francez in [9], it has grown to become one of the fundamental points of study in the development of algorithms for distributed systems. Even though the majority of research on the question was done in the ten years after the problem was introduced, the topic continues to gather interest even today (see, for example, [12] or [13]). For a good overview of the work that has been done on termination detection algorithms, we refer our reader to [15].

The problem of termination detection arises when, in a distributed computation, no further events of the computation are applicable. Even though the computation has terminated, this can not be detected by looking at the state of a single individual process, because the state of such a process could be identical if an event of the computation would be applicable in a different process. The problem of distributed termination detection is to identify when the system as a whole is in such a terminated state, and then to propagate this knowledge to all individual processes.

After its introduction, a large variety of algorithms have been proposed to solve this problem. One strategy that has been suggested is to make use of confirmation messages to detect the arrival of messages sent by a process ([7], [14], [19]). Another strategy proposed is to repeatedly send waves of messages through the system to gather the states of the individual processes ([6], [9]). Further strategies that were suggested included the use of snapshots ([1]), using counters to measure the distance to an active process ([8]) or assigning a certain weight to each message and each active process ([11], [16]).

In [3], Chandy and Misra showed that in the worst case, any termination detection algorithm would at least require a number of control messages equal to the number of messages exchanged in the basic computation. This message optimality has been achieved in the algorithm proposed by Dijkstra and Scholten [7]. Furthermore, message optimality for a computation where multiple processes start out active has been achieved by the modifications suggested by Shavit and Francez in [19].

Recent work on this problem has focused on creating an algorithm that works in a dynamical system ([13]), on creating algorithms that are Fault Tolerant ([12]), or on detection delay ([14]). By contrast, the possibility of achieving better best-case and average-case message complexity has received relatively little attention, even though significant improvements are possible. Some of those improvements have been achieved by the BTTF Wave algorithm. It has a worst case message complexity equal to the algorithm by Shavit and Francez, but achieves significantly better message complexity in the best and average case.

This algorithm, which is part of the Back To The Future family of algorithms, was designed by Farhad Arbab with the assistance of Kees Blom. It makes extensive use of an apparent causal precedence relationship between different messages, which is based on the idea that once a process has received a message, this message can be considered to have caused any messages sent by that process from that point onwards. Furthermore, this relation is transitive, in the sense that an original message can be considered the cause of any messages that were the result of the receipt of a message caused by the original message. Together, the messages caused by an original message are called the future of the original message. The algorithm derives its name from the fact that by using confirmation messages, the future of such an original message is derived at the process that sent the original message. In other words, we go back to the future.

Besides the design and the extensive description of the algorithm by Farhad Arbab, an implementation of the algorithm in Java has also been created by Kees Blom. Extensive testing has been performed on this implementation, supporting the theoretical estimations of the message complexity of the algorithm.

This paper attempts to complete the description of the BTTF Wave algorithm by providing a formal analysis of the algorithm and its message complexity. In order to help the reader understand the relatively complicated algorithm, we focus on what we consider to be the core of the algorithm, while devoting significantly less attention to a number of more complicated features and optimizations present in the algorithm as it has been designed by Farhad Arbab. Whereas an extensive formal proof will be provided for the core of the algorithm, we limit our discussion of these features and optimizations to a short informal description and a sketch of the influence their implementation has on the proof and message complexity.

The rest of this paper is structured as follows. After describing the problem in section 2, we outline the algorithm informally in section 3. We then proceed to prove that the given algorithm is a correct termination detection algorithm in section 5, based on a formal description given in section 4. In section 6, we compare the algorithm to existing solutions, focusing on its message complexity. We also discuss the features and optimizations left out of the core algorithm here.

2 *Problem Description*

2.1 Distributed System

A distributed system is defined as a set of N sequential processes, that cooperate in order to solve a problem. This cooperatively solving of a problem is called the basic computation or main computation. The processes are connected through a set of E channels, through which they can communicate by sending and receiving messages. It is quite natural to represent this set of processes and channels as a graph, where each process is represented as a node of the graph, and each channel is represented as an edge of the graph. In our further discussions, we assume that the graph formed this way is *strongly connected*, i.e. there exists a path (potentially traversing multiple channels) from every node to every other node, allowing any process to send a message to any other process of which it knows the identifier.

As they work to solve the problem, processes will perform a large number of internal actions, potentially changing the state of the process. However, for the purpose of termination detection, the details of these internal transitions are not relevant. We will therefore model the potential states of a process as just two: a process is considered to be active when it is working to solve the problem, and it is considered passive when it is done with the work on its part of the problem. An active process can send messages, receive messages, and change its state to passive at any time. The only action available to a passive process is to receive a message, which will cause it to become active again.

This brings us to communication. As mentioned, processes communicate by sending and receiving messages. We assume that communication happens asynchronously: a send event in the sending process need not be matched immediately by a receive event in the receiving process. Instead, in the period between the send and the receive statement, a message may be in one of three physical locations. First, it can be located at the sending process. Eventually, the message will depart from the sending process and will be in transit in a channel. After a finite time, the message will arrive at the receiving process, were it is available for a receive statement. Please note that this implies that a message can not remain in transit indefinitely.

In the majority of this paper, we assume that the only receive statement available to a process is a blocking receive. If a process calls a receive statement before a message has actually arrived at that process, that receive statement will block until a message arrives. The alternative, a receive with timeout, has been included in the design of the original algorithm, and will be discussed as one of the additional features of the algorithm in section 6. That section will also include an explanation of the restrictions on possible programs that are the result of this assumption (and which are not present when a receive with timeout is allowed).

A distributed system is considered to be terminated when all processes are in a passive state and no messages are in transit. When this has happened, all processes are done with their local part of the computation, and no process will receive further work to perform. Since passive processes can only become active upon receiving a message, such a configuration of the system is stable.

The problem of termination detection in a distributed system is then to develop an algorithm that allows the processes participating in a basic computation to become aware of the fact that the entire system has terminated. Since a process can only detect its local state and can only communicate by sending and receiving messages, this problem is not trivial. Fortunately, as the graph representing the system is strongly connected, it suffices to allow a single process to become aware of the termination of the system. Once one process is aware of this, it can easily broadcast this information to all processes in the system.

For a more extensive treatise on the formal description of a distributed system, we refer our reader to [20].

2.1.1 Examples

To illustrate these concepts, we'll introduce a small distributed system, along with a number of simple computations. The distributed system we'll consider consists of three processes, named A1, A2 and P1. Each of these processes is connected to both others. Note that this graph is indeed strongly connected.

In each of the computations we'll discuss, processes A1 and A2 will start out active, and process P1 will start out passive. Furthermore, it should be noted that each computation only represents a possible permutation as it would be seen by an external observer. The same set of events at each individual sequential process could often lead to a different global computation as well. The only hard constraints on a permutation of a computation are that each receive event is preceded by the corresponding send event, and that events at each individual process must keep the same chronological order.

The first computation is intended to illustrate how processes switch from active to passive and interact through messages. It shows how an active process can send a message, how an active process can receive a message, how a passive process can become activated upon receiving a message, and how active processes can turn passive. Also note how multiple processes are active at the same time.

1. A1 and A2 start out active.
2. A1 sends a message M1 to process A2.
3. A2 receives M1.
4. A1 sends a message M2 to process P1.
5. P1 receives M2 and becomes active.
6. P1 turns passive.
7. A1 turns passive.
8. A2 turns passive.

The next computation is intended to show that messages do not necessarily arrive in the order they are sent. This is possible because the only restriction on the delivery of messages is that these messages are delivered in finite time. There are no restrictions on the order of delivery, allowing messages to overtake each other.

1. A1 and A2 start out active.
2. A1 sends a message M1 to A2.
3. A1 sends a message M2 to A2.
4. A2 receives M2.
5. A1 sends a message M3 to A2.
6. A2 receives M3.
7. A2 receives M1.
8. A2 turns passive.
9. A1 turns passive.

The third computation is intended to illustrate how a computation may not have terminated, even though all processes are passive. In the following computation, after step 4, all processes are passive. However, because message M1 has not been delivered at that point, the computation is not yet terminated at that time. Indeed, the delivery of M1 leads to further messages.

1. A1 and A2 start out active.
2. A1 sends a message M1 to P1.
3. A2 turns passive.
4. A1 turns passive.
5. P1 receives message M1 and turns active.
6. P1 sends a message M2 to A1.
7. A1 receives M2 and turns active.
8. A1 turns passive.
9. P1 turns passive.

The final computation is intended to illustrate the functioning of a blocking receive. It is useful to keep in mind that even though the external observer sees the receive call of process A1 at point 4 block until point 8, the process itself is not aware of this fact. It simply performs a receive call, and continues when that call completes, without any awareness of the passing of time.

1. A1 and A2 start out active.
2. A2 turns passive.
3. A1 sends a message M1 to process P1.
4. A1 attempts to receive a message. As no message has arrived, this call blocks.
5. P1 receives M1 and turns active.
6. P1 sends a message M2 to A1.
7. P1 turns passive.
8. M2 arrives at A1. The receive call of A1 completes.
9. A1 turns passive.

3 *Informal Description of the Algorithm*

There are three important members of the BTTF family of algorithms. For each of those three algorithms, we will provide an informal description of the way the algorithm functions. As has been discussed previously, a number of features and optimizations that were available in the original description and implementation of the algorithm by Farhad Arbab and Kees Blom have been left out of this description of the core functionality of the algorithm, and will be discussed later. It should also be noted that this description only considers the internal functioning of the algorithm. The exact functionality available to a programmer using the algorithm should be specified through an API, as can be seen in their implementation.

This informal description is intended to provide the reader with a good understanding of the way the algorithm functions. However, if any part of the description is unclear, we refer the reader to Appendix A, where a description in pseudocode is available.

The first of the three algorithms is the BTTF Transitory Quiescence Detection Algorithm. This algorithm is intended to make the state in which there are no pending messages detectable, i.e. after running the algorithm each process has some amount of information available which, when combined, informs us that there are no messages in transit. The second algorithm is YAWA, an abbreviation for Yet Another Wave Algorithm. This is a wave based termination detection algorithm inspired by the algorithm of Dijkstra, Feijen and van Gasteren [6]. The third and most important member of the BTTF family is BTTF Wave, which combines the first two algorithms into a final termination detection algorithm that achieves the desirable properties of both. In this section, we'll give an informal description of each of these three algorithms.

3.1 BTTF Transitory Quiescence Detection Algorithm

The goal of the BTTF Transitory Quiescence Detection Algorithm is to ascertain that the presence of messages in transit can be detected by evaluating information locally available in each process. The simplest way of achieving this is by sending a confirmation message for each basic message that has been sent. Since each process knows whether or not the messages it sent have been received, the presence of messages in transit can be determined by checking if a confirmation message has been received for each message that has been sent. Unfortunately, the number of control messages sent when implementing this algorithm always equals exactly the number of messages sent by the original computation. It has been shown by Chandy and Misra in [3] that this number of control messages is a lower bound; any termination detection algorithm will in the worst case require at least a number of control message equal the number of messages exchanged in the original computation. However, in the best and average case, significant improvement in the number of control messages is possible.

In the BTTF TQ algorithm, this improvement is achieved by utilizing an apparent causal precedence relationship between messages. The last message a process received before sending a message is considered to be the cause of the message.

This relationship can be used to confirm the receipt of multiple basic messages with a single control message. If we send a confirmation message (which we'll call C) to the process that sent the message (Y) which caused the message (Z) we received, we can confirm to the sender of that original message (Y) the receipt of both the message we received (Z) and the receipt of the message that caused it (Y). Furthermore, the apparent causal precedence relationship is transitive, in the sense that there may exist a message (X) which caused the message (Y) that was the cause of the message (Z) we received. This implies that there may exist an entire message chain, starting with a message sent by a process that was active when the computation started, and ending with a process that does not send any messages after receiving the last message in the chain. The receipt of the entire message chain can then be confirmed with a single confirmation message. Therefore, we only send a confirmation message when we know that the last message we received will not cause any other messages to be sent. In other words, we send a confirmation message when we turn passive (as no more messages will be sent before a message is received) or when we receive a message (as new messages sent will be considered caused by the newly received message).

However, it is rare that a process will only send a single message in response to receiving a message. Far more often, the process will send the same message to multiple targets, or send different messages to different recipients. As a result, rather than message chains, the messages sent in response to the message from the original process will be organized into message trees, where each message in the tree has as its children those messages of which it is considered the direct cause. Any branch of such a message tree will be form a message chain as previously defined. A confirmation message now confirms the receipt of all messages in a single branch of this tree, rather than all messages in the tree.

It should be noted that only a process that was active at the start of the computation can send the first message of a message chain, and therefore only a process that starts out active can send a message that becomes the root of a message tree. A process that starts out passive can not send any messages before it has turned active by receiving a message, which would be considered the cause of any messages it sends. Because of this special property of processes that start the computation active, we will name such a process a TQRoot.

Upon receiving a confirmation message, the TQRoot that sent the first message of the message chain will need to know a number of things in order to be able to track the arrival of all messages in the tree. First of all, it needs to know the tree of which this message is a branch, to be able to mark the arrival of the appropriate messages. Secondly, for each node along the branch, the root needs to know how many other branches were created at this point, to be able to determine when all messages in this tree have arrived. This information can be delivered to the root by sending a short description of the entire message chain along with normal messages and the eventual control message. This 'piggybacking' of information does not add to the total number of control messages and is relatively cheap, because in general in a distributed system the time taken to deliver a message is determined mainly by the network delay rather than by the size of the message. The description of the message chain should contain a way to identify messages and for each message, a way to determine how many other messages were sent at the same time (the multiplicity of the message), giving the root all the information it needs.

Adding the multiplicity of a message to the original message does introduce a bit of a problem, though. If the message should contain a way to determine how many other messages were sent at the same time, this number needs to be known before the message is actually sent. However, the main computation may decide to send a message, do some computation, then send another message, without receiving a message in between. If the first message is sent immediately, then the presence of the second message won't be listed there, meaning that the TQRoot can not deduce the presence of the second message from the description sent along with the first message. Luckily for us, the network can cause arbitrary but finite delays when delivering a message. This allows us to delay the departure of the first message till a more appropriate moment, while hiding this additional delay from the main computation. Note that this introduced delay will always be finite, because the original computation will eventually try to receive a message (in which case we flush the buffer because later messages can be appended to the message chain of the newly received message) or it will go passive.

The problem of determining how many other messages were caused by a message arises again in section 6, where we discuss the use of prioritySend (sending a message without delaying it) and the use of a receive with timeout. In that section, we discuss an alternative solution to the problem, along with the consequences of using it.

3.1.1 Example

In order to illustrate this algorithm, we again take a look at several example computations. As was the case in the computations in section 2.1.1, processes A1 and A2 will start out active (and therefore will be TQRoots), whereas process P1 will start out passive. Again, we only consider the computation from the point of view of an external observer, not from the point of view of individual processes.

Each message will be accompanied by a list of triplets <messageSetId, sender, numberOfTargets>. Each individual triplet represents a group of messages sent out by the same process at the same time. The last triplet in the list represents the oldest set of messages, and a message from this set can be considered the cause of the messages in the triplet immediately before it. This transitive relation continues over the entire list, making the list of triplets a representation of a message chain.

Our first computation illustrates how a simple message tree is constructed and becomes completed.

1. A1 and A2 start out active.
2. A1 sends a message M1 to A2.
3. A1 accompanies this message M1 by the list of triplets [<A1a, A1, 1>].
4. Furthermore, A1 creates a new message tree, with as root the triplet <A1a, A1, 1>.
5. A2 receives M1 and the accompanying message chain, and stops being a TQRoot.
6. A2 turns passive. It sends a confirmation message, along with the message chain [<A1a, A1, 1>], to A1.
7. A1 receives this confirmation message. It adds a child "COMPLETE" to the node <A1a, A1, 1> in the corresponding message tree.
8. As each node in the message tree has a number of children equal to the numberOfTargets in the node, the message tree is removed.
9. A1 turns passive, and stops being a TQRoot.

Our second example illustrates the creation of longer message chains, as well as the delay of the actual departure of a message until the process turns passive.

In both examples, notice how a process won't create any new message trees after it has received a message or turned passive, even if it started as a TQRoot. Instead, after that point in the computation, any further messages it sends are considered part of a different message tree. In our second example, after receiving M1, any further messages A2 creates will be added to the message tree corresponding to M1 (or a different tree, if it would later receive other messages). Similarly, after it has become passive, A1 will not send any further messages until it receives a message from a different process. Any messages it will send will be added to the message tree of that message. Therefore, after those two moments, no further message trees will be created.

1. A1 and A2 start out active.
2. A1 sends a message M1 to A2.
3. A1 accompanies this message M1 by the list of triplets [<A1a, A1, 1>], and creates the corresponding message tree.
4. A1 turns passive, and stops being a TQRoot.
5. A2 receives M1, and stops being a TQRoot.

6. A2 sends a message M2 to P1. The departure of this message is delayed.
7. A2 turns passive, allowing M2 to depart. It accompanies this message by the message chain [<A2a, A2, 1>, <A1a, A1, 1>].
8. P1 receives M2, and turns active.
9. P1 turns passive. It sends a confirmation message with the message chain [<A2a, A2, 1>, <A1a, A1, 1>] to A1 (the sender of the first batch of messages in the chain).
10. A1 receives this confirmation message. It adds a child <A2a, A2, 1> under the root, then adds "COMPLETE" as a child to the newly created node. Then, the entire tree is removed.

Our third example illustrates how the sending of a confirmation message is delayed until a process receives a different message. This example also shows a simple example of a branching message tree.

Of course, the sending of a new message can also be delayed until a process receives a different message. No example is given of this.

1. A1 and A2 start out active.
2. A1 sends M1 to A2, creating the message chain [<A1a, A1, 1>] and the corresponding message tree
3. A2 receives M1, and stops being a TQRoot.
4. A2 sends M2 to P1. The sending of this message is delayed.
5. A2 sends M3 to P1. The sending of this message is delayed.
6. A2 turns passive. Both M2 and M3 depart. Each is accompanied by the message chain [<A2a, A2, 2>, <A1a, A1, 1>].
7. P1 receives M2 and turns active.
8. P1 receives M3. It sends a confirmation message with the message chain [<A2a, A2, 2>, <A1a, A1, 1>] to A1.
9. A1 receives this confirmation message. It adds <A2a, A2, 2> as a child to <A1a, A1, 1>, and "COMPLETE" as a child to <A2a, A2, 2> in the corresponding message tree.
10. However, because <A2a, A2, 2> should have 2 children, the tree is not yet removed.
11. P1 turns passive. It sends a confirmation message with the message chain [<A2a, A2, 2>, <A1a, A1, 1>] to A1.
12. A1 receives this confirmation message. It adds a second child "COMPLETE" to <A2a, A2, 2>. Then, the entire tree is removed.
13. A1 turns passive, and stops being a TQRoot.

In order to better understand the way message trees are handled, we'll take a closer look at the message tree created in the last example. This tree is created by A1, who is a TQRoot when it creates his first message. It creates the message chain [<A1a, A1, 1>], and creates a new message tree with <A1a, A1, 1> as its root node. In this notation, A1a is the name the process has given to this batch of messages, which allows him to detect the correct tree to which this message chain belongs. A1 is simply the process name, which will tell later processes where they should send their confirmation messages. Finally, because the batch of messages sent out in this case consists of only a single message, the numberOfTargets of this triplet is one, and the eventual message tree will have only a single child for this node.

Note that A1 does not have to delay the departure of M1, because any further messages A1 sends will cause the creation of new message trees.

When A2 sends M2, it knows that it will append the corresponding triplet to the message chain it received. However, A2 delays the actual departure of the message, because there might be more messages that should also become part of this chain. The same thing happens when A2 sends M3. However, when A2 turns passive, it knows that no further messages will be added to the chain. Therefore, it appends the triplet belonging to its current batch to the chain, and then both messages depart, each accompanied by the same message chain. Note how the numberOfTargets in this triplet is 2, to mark that two different messages have been sent as part of this batch.

When P1 receives M2, it stores the accompanying message chain to allow appending of a triplet if P1 decides to send messages. However, when P1 receives M3, it knows that no messages will be appended to the previous chain, so it can send a confirmation message for the chain that accompanied M2.

When A1 receives this confirmation message, it finds the corresponding tree in its forest by matching the first triplet of the chain (<A1a, A1, 1>) to the root of the corresponding message tree. The second triplet (<A2a, A2, 2>) in the message chain represents a batch of messages that was caused by one of the messages represented by the triplet <A1a, A1, 1>. Therefore, this triplet is added as a child of <A1a, A1, 1>. Even though there are no further triplets listed in the message chain, the arrival of a confirmation message informs A1 that at least one of the messages represented by the triplet <A2a, A2, 2> arrived at its destination.

Next, A1 checks to see if the tree can be removed. However, as the triplet <A2a, A2, 2> only has a single child but a numberOfTargets of 2, there is at least one message in the system that was caused by the initial batch of messages for which A1 can't be sure that it has arrived. Therefore, the tree isn't removed at this point in time.

When P1 turns passive, it knows that no further triplets will be appended to the chain that accompanied M3, so it sends this chain as a confirmation message back to A1, who is responsible for this tree as it was the process that sent the original batch of messages.

When this message chain arrives at A1, it again finds the corresponding tree by matching the first triplet of the chain to the root of the tree. Furthermore, it can match the second triplet of the chain to one of the children of the root of the tree, allowing A1 to conclude that this message chain represents a message from the same batch of messages. As there are no further triplets in the chain, it again adds a node "COMPLETE" to the triplet <A2a, A2, 2>.

Now, each triplet in the tree has a number of children exactly equal to its numberOfTargets. Therefore, A1 can conclude that all messages that were caused by a message represented in this chain have arrived, and therefore A1 removes this tree from its message forest.

3.2 YAWA

Since each message chain ends with a message to the root of the message tree, the setup discussed above can be used quite naturally to implement a wave algorithm. A wave algorithm is defined in [20] according to Definition 1.

Definition 1 *A wave algorithm is a distributed algorithm that satisfies the following three requirements.*

1. ***Termination.*** *Each computation is finite:*
   $\forall C : C < \infty$
   Where C is a computation.

2. ***Decision.*** *Each computation contains at least one decide event:*
   $\forall C : \exists e \in C : e \text{ is a decide event.}$

3. ***Dependence.*** *In each computation each decide event is causally preceded by an event in each process:*
   $\forall C : \forall e \in C : (e \text{ is a decide event} \rightarrow \forall q \in P \exists f \in C_q : f \prec e)$
   where P is the set of processes, C_q is the computation at process q, and $\prec$ denotes a relationship which orders the events in the computation such that each send event is ordered before the corresponding receive, and each event in a process is ordered according to the chronological order of events in that process.

To implement a wave using the notion of message trees, one process will function as initiator for the YAWA algorithm. This process will start the wave, will be responsible for the message tree that is associated with the wave, will receive all confirmation messages related to the wave and will eventually decide whether the computation has terminated.

To guarantee point 2 of the definition, it is enough to make sure that the initiator of the algorithm decides when all messages from the tree have been confirmed. To guarantee point 3, we let each process send the messages of the wave algorithm to each other process it communicates with. The fact that the network is fully connected then guarantees that each process receives a message from the wave at least once. Finally, to guarantee point 1 of the definition, we need to make sure that the computation terminates. Messages will arrive in finite time, and be processed in finite time, so the only thing we have to make sure is that the total number of messages remains finite. This can be guaranteed by having each process forward the messages of the wave only once.

Having developed an algorithm to distribute waves through the system and get the results of the wave back to the initiator, it is possible to illustrate the working of the wave distribution by transforming it to a complete termination detection algorithm. This is done by overlaying an existing wave algorithm for termination detection onto our wave algorithm.

When he designed the YAWA algorithm, Farhad Arbab chose to modify the algorithm proposed by Dijkstra, Feijen and van Gasteren [6] for this purpose, as it is a simple and elegant example of a wave-based algorithm. That algorithm uses tokens of a certain color to determine if messages have been sent since the last time a token passed. Each process starts white, turns red when sending a message, and turns white again when sending a token message. The token starts white, and turns red when being passed on by a red process. Since the Dijkstra Feijen van Gasteren algorithm assumes synchronous communication, it can be guaranteed that, if a white token returns to the initiator, all processes are passive and no messages are in transit.

The same idea can be used in our wave algorithm, where (for the simple version of the YAWA algorithm) the assumption of synchronous communication is replaced by the assumption that the order of messages in a communication channel remains preserved. When distributing the tokens of the wave algorithm, every channel that has been used for communication will be traversed at least once. As a result, if not all messages of the underlying computation have been delivered, there must be at least one message of that computation that is delivered during this round of the wave algorithm, since otherwise a message of the wave algorithm must overtake a message of the original computation. Since the total number of messages sent by a terminating computation is finite, eventually all message will have been delivered. All tokens of the wave algorithm started after that point will remain white, signaling to the initiator that the process has terminated.

The number of messages required for this algorithm can be further reduced by having token messages "chase" basic messages. If a process forwards any token it receives to those processes to which it sent basic messages since the last time it received a token, it can be guaranteed that all basic messages have been received when the message tree belonging to the tokens becomes complete. This reduces the number of rounds required for the algorithm to complete. Furthermore, with some modifications, this can be used to drop the requirement of maintaining the order of messages in a channel. However, this modification was not taken into account when we focused on the core part of the algorithm, as the same effect is reached without this modification in the BTTF Wave algorithm.

3.3 BTTF Wave

The two algorithms described above can be combined into a single termination detection algorithm, which has some interesting properties. The basic idea of this combination is to use the BTTF Transitory Quiescence Detection Algorithm to gather the information regarding the presence of messages belonging to the underlying computation in those processes that form the root of message chains, and then run the wave part of the YAWA algorithm to actually determine whether the system has terminated. In this adaptation of the YAWA algorithm, a process can only forward token messages if it is guaranteed that it will create no further message trees and all message trees of which it is the root have been completed. Note that there are two possible ways in which it can be guaranteed that no further message trees will be created. Either the process currently is passive, or it is active but it already received a message. In both cases, further messages sent by this process will be added to a different tree, so no new message trees will be created. Since a process will not create other message trees, a single wave suffices, regardless of whether the requirement that the order of messages in a communication channel remains preserved is met.

3.3.1 Example

In order to illustrate how message trees can be used to create a wave algorithm, and to see how this transforms the Transitory Quiescence algorithm into a complete termination detection algorithm, we again create an example. Process P1 will be the initiator of the YAWA part of the algorithm, and will be responsible for the eventual termination detection.

To keep things simple, we assume that only A1 starts as TQRoot, and we further assume that A1 turns passive without sending any messages. Both A2 and P1 start out passive.

As it starts out passive and without any message trees, P1 can initiate the algorithm whenever it wants. It does this by sending messages to its neighbors A1 and A2.

We assume that A2 starts out passive. This means that when a message from the YAWA part of the algorithm arrives at this process, it is immediately forwarded to all neighbors of the process, as can be seen when the tokens reach A2.

However, when a message from the YAWA algorithm arrives at a process that started as a TQRoot, chances are the rest of the wave will be delayed. We see this happening when the first token arrives at A1, as nothing is done with it at that point in time because the process is still active without having previously received any messages. Similarly, the wave would be delayed when there would still be an uncompleted message tree at a process. Only when all trees in a forest have been removed and the process is no longer a TQRoot will that process forward tokens from the wave.

Finally, whenever a message from the YAWA algorithm arrives at a process that received a message from the YAWA algorithm before, that process sends a confirmation to the initiator, regardless of whether the process already forwarded messages from the YAWA algorithm.

1. A1 starts out active, A2 starts out passive, P1 starts out passive but is the initiator for the YAWA algorithm.

2. P1 sends token messages T1 to A1 and T2 to A2.

3. T1 arrives at A1. A1 doesn't do anything at this time, because it is still active.

4. T2 arrives at A2. Because A2 is not a TQRoot and its message forest is empty, it responds by sending tokens T3 to A1 and T4 to P1.
5. T3 arrives at A1. Because A1 received a token before, it sends the message chain belonging to T3 back to the initiator, P1.
6. T4 arrives at P1. P1 considers this a confirmation message, and completes the first branch of the message tree.
7. the message chain belonging to T2/T3 arrives at P1. P1 completes another branch of its message tree.
8. A1 turns passive, and is no longer a TQRoot. It now sends a token T5 to A2 and T6 to P1. Each of these is appended to the message chain belonging to T1.
9. T5 arrives at A2. Because A2 received a token before, it sends the entire message chain back to P1.
10. the message chain belonging to T1/T5 arrives at P1. P1 completes another branch of the message tree.
11. T6 arrives at P1. P1 completes the last branch of the message tree, and concludes termination.

4 *Formal Description*

In this section, we will give a formal description of the concepts introduced in previous sections. This formal description will form the foundation on which we prove the correctness of the algorithm. In the creation of this formal description, we have made heavy use of the pseudocode description that can be found in appendix A. If at any point part of this formal description is unclear, studying the pseudocode description should help build understanding.

Definition 2 *In a distributed system*
$\mathbb{P} = \{p_1, \ldots, p_n\}$ *is a set of processes,*
$\mathbb{S} = \{active, passive\}$ *is a set of states,*
$\mathbb{E} \subseteq \{(p, q) : p, q \in \mathbb{P}\}$ *is a set of channels,*
$\mathbb{M}$ *is a set of messages, with* $\{Conf, Tok\} \subseteq \mathbb{M}$*,*
$\mathbb{B} = \{True, False\}$ *is a set of truth values,*
$\mathbb{N}$ *is the set of natural numbers, and*
$\mathbb{ID}$ *is a set of identifiers, with* $Comp \in \mathbb{ID}$*.*

We start by defining a number of different sets, which will be used in the rest of the formal description.

Definition 3 *A distributed system is strongly connected if*
$\forall p, q \in \mathbb{P}\ \exists p_1, p_2, \ldots, p_k \in \mathbb{P}$ *(potentially* $k = 0$*):*
$\{(p, p_1), (p_1, p_2), \ldots, (p_{k-1}, p_k), (p_k, q)\} \subseteq \mathbb{E}$

Definition 4 *A path over the least nodes between two processes p and q is defined as*
$\{(p, p_1), (p_1, p_2), \ldots, (p_{k-1}, p_k), (p_k, q)\} \subseteq \mathbb{E}$ *such that*
$\neg\exists p_{1'}, p_{2'}, \ldots, p_{k'} \in \mathbb{P} : \{(p, p_{1'}), (p_{1'}, p_{2'}), \ldots, (p_{k'-1}, p_{k'}), (p_{k'}, q)\} \subseteq \mathbb{E} \land k' < k$

Definition 5 *In a process p, a first node on the path to a process q is defined as the endpoint of an edge starting in p that is part of a path over the least nodes between p and q.*

In order for our algorithm to work, any process must be able to send a message to any other process of which it knows the name. Part of this requires that there exists a path from any process to any other process. The definition specifies this by making explicit the set of channels connecting the two processes. The path over the least nodes then is the smallest set of channels that forms a path between two processes.

A first node on the path to a process q is the process to which a message should be sent in order to eventually arrive at q.

Lemma 6 *If a distributed system is strongly connected, then $\forall p, q$ there exists a path over the least nodes between p and q.*

Proof.

1. According to definition 3, $\exists p_1, p_2, \ldots, p_k \in \mathbb{P} : \{(p, p_1), (p_1, p_2), \ldots, (p_{k-1}, p_k), (p_k, q)\} \subseteq \mathbb{E}$.
2. Let K be the set of k. K must be a non-empty subset of $\mathbb{N}$.
3. According to the well-ordering principle, such a subset must have a smallest element. Let us call this smallest element k'.
4. Since $k' \in K$, $\exists p_1, p_2, \ldots, p'_k \in \mathbb{P} : \{(p, p_1), (p_1, p_2), \ldots, (p_{k'-1}, p'_k), (p'_k, q)\} \subseteq \mathbb{E}$.
5. Furthermore, $\neg\exists p_1, p_2, \ldots, p_{k''} \in \mathbb{P} : \{(p, p_1), (p_1, p_2), \ldots, (p_{k''-1}, p_{k''}), (p_{k''}, q)\} \subseteq \mathbb{E} \wedge k'' < k'$, because this would require $k'' < k' \wedge k'' \in K$, which would be in contradiction with the assumption that k' is the smallest element of K.
6. This means that $\{(p, p_1), (p_1, p_2), \ldots, (p_{k'-1}, p'_k), (p'_k, q)\} \subseteq \mathbb{E}$ is a path over the least nodes between p and q. $\square$

Assumption 7 *The distributed system we are describing is strongly connected.*

This assumption merely makes our requirement explicit.

Definition 8 *A messageInfo object is defined as a triplet $(id,\ p,\ n)$ where $id \in \mathbb{ID}$, $p \in \mathbb{P}$, and $n \in \mathbb{N}$.*
$(Comp,\ p,\ 0)$ with $p \in \mathbb{P}$ is a special messageInfo object.
$\mathbb{MI}$ is the set of all messageInfo objects.

The messageInfo object forms a core element of our algorithm, that represents a batch of messages. *id* is an identifier for the entire batch, p denotes the process that sent the batch of messages, and n denotes the number of messages in the batch. The messageInfo object $(Comp,\ p,\ 0)$ represents a batch of zero messages, and is used if a message did not cause further messages.

Definition 9 *A messageChain is defined recursively as a list:*
$()$ is a messageChain.
$(I,\ MC)$ with $I \in \mathbb{MI}$ and MC a message chain, is a messageChain.

A messageChain is simply a list of messageInfo objects. The idea here is that a message from the batch of messages represented by the first messageInfo caused the batch of messages represented by the second messageInfo. Similarly, a message from the second batch caused the third batch, and so on, and so forth.

Definition 10 *The operation MC ++ MI where MC is a messageChain, MI is a messageInfo, and the result is a messageChain is defined recursively as follows:*
if $MC = ()$ then $MC \mathbin{++} MI = (MI,\ ())$
if $MC = (I,\ MC')$ then $MC \mathbin{++} MI = (I,\ MC' \mathbin{++} MI)$

The operation ++ adds a messageInfo object to the end of a messageChain. This allows messageInfo objects to be added to an existing messageChain if a message from the last batch of that chain caused further messages.

Definition 11 *A messageTree is defined recursively:*
$(I,\ ())$ *with* $I \in \mathbb{MI}$ *is a messageTree.*
$((id,\ p,\ k),\ (T_1,\ T_2,\ \ldots,\ T_n))$ *with* $(id,\ p,\ k) \in \mathbb{MI}$*,* $k \geq n$*, and* $\forall i \in (1,\ \ldots,\ n) : T_i$ *a messageTree, is a messageTree.*

Another important concept for our algorithm is the messageTree. $(I,\ ())$ denotes a node without children, whereas $(I,\ (T_1,\ T_2,\ \ldots,\ T_n))$ denotes a node with children. Again, the idea of this representation is that a message from the batch represented by the parent node caused the batch represented by a child.

Definition 12 *The operation* $T + MC$ *where* T *is a messageTree,* MC *is a messageChain and the result is a messageTree is defined recursively as follows:*

if $T = ((id,\ p,\ n),\ ())$ *and* $MC = ((id,\ p,\ n),\ ())$
then $T + MC = ((id,\ p,\ n),\ ((Comp,\ p,\ 0),\ ())\ )$

if $T = ((id,\ p,\ n),\ (T_1,\ T_2,\ \ldots,\ T_{n'}))$ *and* $MC = ((id,\ p,\ n),\ ())$
then $T + MC = ((id,\ p,\ n),\ (T_1,\ T_2,\ \ldots,\ T_{n'},\ ((Comp,\ p,\ 0),\ ())\ ))$

if $T = ((id,\ p,\ n), ())$ *and* $MC = ((id,\ p,\ n),\ ((id',\ p',\ n'),\ MC'))$
then $T + MC = ((id,\ p,\ n),\ (\ ((id',\ p',\ n')) + ((id',\ p',\ n'),\ MC')))$

if $T = ((id,\ p,\ n),\ (T_1,\ \ldots,\ ((id',\ p',\ n'),\ T_i),\ \ldots,\ T_{n''}))$ *and* $MC = ((id,\ p,\ n),\ ((id',\ p',\ n'),\ MC'))$
then $T + MC = ((id,\ p,\ n),\ (T_1,\ \ldots,\ ((id',\ p',\ n'),\ T_i) + ((id',\ p',\ n'),\ MC'),\ \ldots,\ T_{n''}))$

if $T = ((id,\ p,\ n),\ (T_1,\ T_2,\ \ldots,\ T_{n''}))$ *and* $MC = ((id,\ p,\ n),\ ((id',\ p',\ n'),\ MC'))$
with $\forall i \in (1,\ \ldots,\ n'')\ T_i \neq ((id',\ p',\ n'),\ T')$
then $T + MC = ((id,\ p,\ n),\ (T_1,\ T_2,\ \ldots,\ T_{n''},\ ((id',\ p',\ n')) + ((id',\ p',\ n'),\ MC')\ ))$

Next, it is useful to define how a messageChain can be added to a messageTree. This means that each messageInfo in the messageChain must be added as a child of its predecessor in the chain. Furthermore, if a messageInfo in the messageChain does not have a successor, the messageTree will need to mark that one of the messages represented by the batch has arrived. The node $(Comp,\ p,\ 0)$ is used for that. Those nodes will become leaf nodes for the eventual tree. Note that if the first messageInfo in the chain and tree don't overlap, the chain can't be added to the tree.

The idea of this definition is to follow the nodes of the tree until you reach a node in the chain that is not yet represented in the tree, then add that node and its children. To do this, the definition systematically works its way through all possibilities. It starts by assuming that both the messageTree and the messageChain consists of a single node, in which case the single node of the messageTree is given a single child. If the first node of the messageTree already had more children when the messageChain consists of a single node, an extra child is added as a sibling of the existing children.

If the messageChain is longer, there are three possibilities. The first is that the messageTree consists of a single node. In that case, the second node of the messageChain will form the root of a new subtree, which is constructed by a recursive call to the + operation. The second possibility is that the messageTree already had multiple children, where one of the children matches the second node in the messageChain. In that case, the rest of the messageChain should be added to the subtree rooted in that node. In the third and final case, the messageTree does have a number of children, but none of those match the rest of the messageChain. In that case, the rest of the messageChain needs to become a subtree that is a sibling of the existing children.

Definition 13 *A messageTree is called a complete messageTree if it is of the form* $((Comp,\ p,\ 0),\ ())$*, or if it is of the form* $((id,\ p,\ n),\ (T_1,\ T_2,\ \ldots,\ T_n))$ *and* $\forall i \in \{1,\ \ldots,\ n\} : T_i$ *is a complete messageTree.*

A messageTree is a complete messageTree if each node has a number of children exactly equal to the number of messages in the batch that it represents. As a Comp node represents a batch of zero messages, it doesn't need to have any children. However, if a node is not a Comp node, it needs to have a number of children exactly equal to the integer in its messageInfo.

Definition 14 *A messageForest is a set of messageTree objects.*

The final concept necessary for the BTTF Transitory Quiescence algorithm is a relatively simple one.

Definition 15 $\hat{a}_i = (s_i,\ tqroot_i,\ lmc_i,\ f_i,\ init_i,\ fmc_i,\ ter_i,\ rec_i)$*,*
where $p_i \in \mathbb{P}$*,* $s_i \in \mathbb{S}$*,* $tqroot_i \in \mathbb{B}$*,* lmc_i *a messageChain,* f_i *a messageForest,* $init_i \in \mathbb{B}$*,* fmc_i *a messageChain,* $ter_i \in \{True,\ False,\ Pending\}$*,* $rec_i \in \mathbb{B}$
$\mathcal{A} = \{(p_i, \hat{a}_i)\}$ *is a set of process attribute pairs*

Definition 16 $\mathcal{M}_{buff} = \{((m,\ dest),\ p)\}$ *is the set of buffered messages at process* p*,*
$\mathcal{M}_{trans} = \{((m,\ mc, dest),\ e)\}$ *is the set of messages in channel* e*, and*
$\mathcal{M}_{arr} = \{((m,\ mc,\ dest),\ p)\}$ *is the set of messages that are located (but not buffered) at process* p*,*
where $m \in \mathbb{M}$*,* $dest \in \mathbb{P}$*,* $p \in \mathbb{P}$*,* mc *a messageChain,* $e \in \mathbb{E}$*.*

Definition 17 *A configuration is a sextuplet* $(\mathbb{P},\ \mathbb{E},\ \mathcal{A},\ \mathcal{M}_{buff},\ \mathcal{M}_{trans},\ \mathcal{M}_{arr})$*.*
Typically, $\mathbb{C}$ *will denote the set of configurations.*

A configuration is a representation of the exact combination of processes and associated variables. It defines what processes exist, what channels connect these processes, the exact attributes of each process, and the location of each message in the system.

The attributes a process can have are exactly the variables of the pseudocode description, minus the buffer. p_i denotes the process that holds the variables, s_i denotes whether that process is active or passive, $tqroot_i$ is a boolean that is true if the process is a TQRoot, lmc_i stores the last messageChain received by the process, f_i holds the messageForest for which this process is responsible, $init_i$ is a boolean that is true if the process is an initiator for the YAWA algorithm, fmc_i holds the first messageChain that is part of the YAWA algorithm that was received by the process, ter_i is a boolean that will become true when the process has detected termination, and rec_i is a boolean that denotes if a process is blocking in order to receive.

Each message in the system is accompanied by its destination, to allow the correct process to receive the message. A buffered message is represented by a message located at a process, but not accompanied by a messageChain. When a message has left the buffer, it is always accompanied by a messageChain, and is either located in a channel or at a process.

Definition 18 *There are various notation conventions we would like to define. We denote the various positions in* $\hat{a}_i$ *by* s*,* tq*,* lmc*,* f*,* $init$*,* fmc*,* ter*, and* rec*.*
This notation can be used to denote replacing the value at a specific position by appending it to the new value within square brackets.
For example, $\hat{a}_i[k]_s$ *denotes replacing the value at position* s *in* $\hat{a}_i$ *with* k*.*

This notation will prove useful when we will be changing the values in $\hat{a}_i$. Note how multiple replacements can be stacked by appending them to each other.

Definition 19 *A configuration of a distributed system is called a potential initial configuration if*
$c = (\mathbb{P}, \mathbb{E}, \mathcal{A}, \emptyset, \emptyset, \emptyset)$
where $\forall i \in \{1, \ldots, n\}$:
$\hat{a}_i = (active, True, (), \emptyset, init_1, (), False, False) \vee \hat{a}_i = (passive, False, (), \emptyset, init_1, (), False, False)$
$\wedge \exists j \in \{1, \ldots, n\} : init_j = True$
$\wedge \forall k, l \in \{1, \ldots, n\} : init_k = init_l = True \to k = l$

Definition 20 *A configuration of a distributed system is a terminated configuration if*
$c = (\mathbb{P}, \mathbb{E}, \mathcal{A}, \mathcal{M}_{buff}, \mathcal{M}_{trans}, \mathcal{M}_{arr})$
where $\forall i \in \{1, \ldots, n\}$:
$\hat{a}_i(s) = passive \wedge$
$\forall m \in \mathcal{M} : m = Tok \vee m = Conf$

There are two special configurations. A potential initial configuration is a configuration in which the system might be when a computation starts. A terminated configuration is a configuration in which the system might be when the basic computation has terminated. Note that we do not require the BTTF Wave algorithm to have terminated for a configuration to be a terminated configuration.

In an initial configuration, there are no messages in the system and no processes hold messageChains or has a messageTree in its messageForest. If a process starts out active, it is a TQRoot, so tqroot is true. Conversely, if a process starts out passive, tqroot is false. Furthermore, there is exactly one process for which init is true, as there is exactly one initiator.

A configuration is a terminated configuration if all processes are passive and all messages of the basic computation have been delivered. This means that the only messages still in the system are either Tok or Conf messages.

4.1 Computation

A core concept in our formal description of the algorithm is the concept of a computation. A computation describes all changes that happened in order to change the system from an initial configuration to a terminated configuration. In order to describe a computation, we introduce the notion of a computation step, which describes the possible transitions from one configuration to another.

Definition 21 *A computation step* $\to_{BTTF}$ *is a relation on* $\mathbb{C} \times \mathbb{C}$.

A computation step is a transition from one configuration to another. This both details the transition of the distributed system (an active process sends a message, an active process turns passive, and so on) as well as the transitions related to the BTTF Wave algorithm. The description we will give here will follow mostly the same pattern as the pseudocode description that can be found in appendix A. When things are unclear here, further explanation can be found in the appendix.

The basic concept of this computation step is that we describe the attributes of the processes before the step is executed ($\mathcal{A}$), and the attributes of the processes after the computation step is executed ($\mathcal{A}'$). Furthermore, we describe what messages exists after the computation step ($\mathcal{M}'$) in relation to the messages that existed before the computation step ($\mathcal{M}$).

Definition 22 (SEND) *The following two rules describe those computation steps that allow a process to send a message:*

(a) *% an active TQRoot can send a message*

$$\frac{\begin{array}{l} (p_i,\ (active,\ True,\ lmc_i,\ f_i,\ init_i,\ fmc_i,\ ter_i,\ False)) \in \mathcal{A}, \\ \mathcal{A}' = \mathcal{A} \cup \{(p_i,\ \hat{a}_i[f_i \cup \{((id,\ p_i,\ 1),\ ())\}\ ]_f\ )\} \setminus \{(p_i,\ \hat{a}_i)\}, \\ \mathcal{M}'_{trans} = \mathcal{M}_{trans} \cup \{((m,\ ((id,\ p_i,\ 1),\ ()),\ q),\ (p_i,\ q'))\}, \\ q' \textit{ is a first node on the path to } q, \end{array}}{(\mathbb{P},\ \mathbb{E},\ \mathcal{A},\ \mathcal{M}_{buff},\ \mathcal{M}_{trans},\ \mathcal{M}_{arr}) \rightarrow_{BTTF} (\mathbb{P},\ \mathbb{E},\ \mathcal{A}',\ \mathcal{M}_{buff},\ \mathcal{M}'_{trans},\ \mathcal{M}_{arr})}$$

(b) *% an active process that's not a TQRoot can send a message*

$$\frac{\begin{array}{l} (p_i,\ (active,\ False,\ lmc_i,\ f_i,\ init_i,\ fmc_i,\ ter_i,\ False)) \in \mathcal{A}, \\ \mathcal{M}'_{buff} = \mathcal{M}_{buff} \cup \{((m,\ dest),\ p_i)\} \end{array}}{(\mathbb{P},\ \mathbb{E},\ \mathcal{A},\ \mathcal{M}_{buff},\ \mathcal{M}_{trans},\ \mathcal{M}_{arr}) \rightarrow_{BTTF} (\mathbb{P},\ \mathbb{E},\ \mathcal{A},\ \mathcal{M}'_{buff},\ \mathcal{M}_{trans},\ \mathcal{M}_{arr})}$$

Our first definition defines the possible ways in which a process can send a message. The basic requirements for a process to send a message are that the process is active and not blocking to receive. Here, two cases can be identified. Either the process sending a message is a TQRoot or the process sending isn't a TQRoot. If the sending process is a TQRoot, a new messageTree is added to the messageForest of that TQRoot, and the message, accompanied by a messageChain, is put on the channel that will eventually cause the message to arrive at its destination. Note that according to Lemma 6, the path over the least nodes that is used to deliver the message actually exists. It follows that a first node on the path to q exists as well. If the sending process isn't a TQRoot, on the other hand, all that happens is that the message is buffered till a later time. This is represented by having a message that is unaccompanied by a messageChain sitting at the process.

Definition 23 (RECEIVE) *The following rules describe those computation steps that allow a process to indicate that it is ready to receive a message:*

(a) *% a TQRoot can receive a message with trees in its messageForest (nonInitiatorBecomesEmpty doesn't do anything)*

$$\frac{\begin{array}{l} (p_i,\ (s_i,\ True,\ (),\ f_i,\ init_i,\ fmc_i,\ ter_i,\ False)) \in \mathcal{A}, \\ f_i \neq \emptyset, \\ \mathcal{A}' = \mathcal{A} \cup \{(p_i,\ \hat{a}_i[False]_{tq}[True]_{rec})\} \setminus \{(p_i, \hat{a}_i)\} \end{array}}{(\mathbb{P},\ \mathbb{E},\ \mathcal{A},\ \mathcal{M}_{buff},\ \mathcal{M}_{trans},\ \mathcal{M}_{arr}) \rightarrow_{BTTF} (\mathbb{P},\ \mathbb{E},\ \mathcal{A}',\ \mathcal{M}_{buff},\ \mathcal{M}_{trans},\ \mathcal{M}_{arr})}$$

(b) *% a TQRoot can receive a message without trees in its messageForest, with empty fmc, and false init (nonInitiatorBecomesEmpty doesn't do anything)*

$$\frac{\begin{array}{l} (p_i,\ (s_i,\ True,\ (),\ \emptyset,\ False,\ (),\ ter_i,\ False)) \in \mathcal{A}, \\ \mathcal{A}' = \mathcal{A} \cup \{(p_i,\ \hat{a}_i[False]_{tq}[True]_{rec})\} \setminus \{(p_i, \hat{a}_i)\} \end{array}}{(\mathbb{P},\ \mathbb{E},\ \mathcal{A},\ \mathcal{M}_{buff},\ \mathcal{M}_{trans},\ \mathcal{M}_{arr}) \rightarrow_{BTTF} (\mathbb{P},\ \mathbb{E},\ \mathcal{A}',\ \mathcal{M}_{buff},\ \mathcal{M}_{trans},\ \mathcal{M}_{arr})}$$

(c) *% a TQRoot can receive a message without trees in its messageForest, with empty fmc, and true init (The process sends a token to all neighbors, ter becomes pending)*

$(p_i,\ (s_i,\ True,\ (),\ \emptyset,\ True,\ (),\ False,\ False)) \in \mathcal{A},$

$$\dfrac{\begin{array}{l}\mathcal{A}' = \mathcal{A} \cup \{(p_i,\ \hat{a}_i[False]_{tq}[\{((id,\ p_i,\ n'),\ ())\}]_f[Pending]_{ter}[True]_{rec})\} \setminus \{(p_i,\ \hat{a}_i)\},\\ \mathcal{M}'_{trans} = \mathcal{M}_{trans} \cup \{((Tok,\ ((id,\ p_i,\ n'),\ ()), q),\ (p_i,\ q)) : (p_i,\ q) \in \mathbb{E}\},\\ n' = |\{(p_i,\ q) : (p_i,\ q) \in \mathbb{E}\}|\end{array}}{(\mathbb{P},\ \mathbb{E},\ \mathcal{A},\ \mathcal{M}_{buff},\ \mathcal{M}_{trans},\ \mathcal{M}_{arr}) \rightarrow_{BTTF} (\mathbb{P},\ \mathbb{E},\ \mathcal{A}',\ \mathcal{M}_{buff},\ \mathcal{M}'_{trans},\ \mathcal{M}_{arr})}$$

(d) *% a TQRoot can receive a message without trees in its messageForest, with non-empty fmc (The process sends a token to all neighbors)*

$$\dfrac{\begin{array}{l}(p_i,\ (s_i,\ True,\ (),\ \emptyset,\ False,\ fmc_i,\ ter_i,\ False)) \in \mathcal{A},\\ fmc_i \neq (),\\ \mathcal{A}' = \mathcal{A} \cup \{(p_i,\ \hat{a}_i[False]_{tq}[True]_{rec})\} \setminus \{(p_i, \hat{a}_i)\},\\ \mathcal{M}'_{trans} = \mathcal{M}_{trans} \cup \{((Tok,\ fmc_i ++ (id,\ p_i,\ n'), q),\ (p_i,\ q)) : (p_i,\ q) \in \mathbb{E}\},\\ n' = |\{(p_i,\ q) : (p_i,\ q) \in \mathbb{E}\}|\end{array}}{(\mathbb{P},\ \mathbb{E},\ \mathcal{A},\ \mathcal{M}_{buff},\ \mathcal{M}_{trans},\ \mathcal{M}_{arr}) \rightarrow_{BTTF} (\mathbb{P},\ \mathbb{E},\ \mathcal{A}',\ \mathcal{M}_{buff},\ \mathcal{M}'_{trans},\ \mathcal{M}_{arr})}$$

(e) *% an active non-TQRoot can receive a message with an empty buffer*

$$\dfrac{\begin{array}{l}(p_i,\ (active,\ False,\ ((id,\ q,\ n'),\ lmc'_i),\ f_i,\ init_i,\ fmc_i,\ ter_i,\ False)) \in \mathcal{A},\\ \mathcal{A}' = \mathcal{A} \cup \{(p_i,\ \hat{a}_i[()]_{lmc}[True]_{rec})\} \setminus \{(p_i, \hat{a}_i)\},\\ \neg\exists((m,\ dest),\ p_i) \in \mathcal{M}_{buff},\\ \mathcal{M}'_{trans} = \mathcal{M}_{trans} \cup \{((Conf,\ ((id,\ q,\ n'),\ lmc'_i),\ q),\ (p_i,\ q'))\},\\ q' \textit{ is a first node on the path to } q\end{array}}{(\mathbb{P},\ \mathbb{E},\ \mathcal{A},\ \mathcal{M}_{buff},\ \mathcal{M}_{trans},\ \mathcal{M}_{arr}) \rightarrow_{BTTF} (\mathbb{P},\ \mathbb{E},\ \mathcal{A}',\ \mathcal{M}_{buff},\ \mathcal{M}'_{trans},\ \mathcal{M}_{arr})}$$

(f) *% an active non-TQRoot can receive a message with non-empty buffer*

$$\dfrac{\begin{array}{l}(p_i,\ (active,\ False,\ lmc_i,\ f_i,\ init_i,\ fmc_i,\ ter_i,\ False)) \in \mathcal{A},\\ \mathcal{A}' = \mathcal{A} \cup \{(p_i,\ \hat{a}_i[()]_{lmc}[True]_{rec})\} \setminus \{(p_i, \hat{a}_i)\},\\ \exists((m,\ dest),\ p_i) \in \mathcal{M}_{buff},\\ \mathcal{M}'_{buff} = \mathcal{M}_{buff} \setminus \{((m_1,\ dest_1),\ p_i),\ \ldots,\ ((m_{n'},\ dest_{n'}),\ p_i)\},\\ \mathcal{M}'_{trans} = \mathcal{M}_{trans} \cup \{((m_1,\ lmc_i ++ (id,\ p_i,\ n'),\ dest_1),\ (p_i,\ q_1)),\ \ldots,\ ((m_{n'},\ lmc_i ++ (id,\ p_i,\ n'),\\ dest_{n'}),\ (p_i,\ q_{n'}))\},\\ \forall i \in \{1,\ \ldots,\ n'\}\ q_i \textit{ is a first node on the path to } dest_i\end{array}}{(\mathbb{P},\ \mathbb{E},\ \mathcal{A},\ \mathcal{M}_{buff},\ \mathcal{M}_{trans},\ \mathcal{M}_{arr}) \rightarrow_{BTTF} (\mathbb{P},\ \mathbb{E},\ \mathcal{A}',\ \mathcal{M}'_{buff},\ \mathcal{M}'_{trans},\ \mathcal{M}_{arr})}$$

(g) *% a passive process can receive a message*

$$\dfrac{\begin{array}{l}(p_i,\ (passive,\ False,\ (),\ f_i,\ init_i,\ fmc_i,\ ter_i,\ False)) \in \mathcal{A},\\ \mathcal{A}' = \mathcal{A} \cup \{(p_i,\ \hat{a}_i[True]_{rec})\} \setminus \{(p_i, \hat{a}_i)\}\end{array}}{(\mathbb{P},\ \mathbb{E},\ \mathcal{A},\ \mathcal{M}_{buff},\ \mathcal{M}_{trans},\ \mathcal{M}_{arr}) \rightarrow_{BTTF} (\mathbb{P},\ \mathbb{E},\ \mathcal{A}',\ \mathcal{M}_{buff},\ \mathcal{M}_{trans},\ \mathcal{M}_{arr})}$$

Our second definition deals with processes receiving a message. Because a TQRoot stops being a TQRoot when it receives a message, this can trigger the YAWA part of the algorithm. As a result, the set of rules is somewhat complicated. Essentially, what happens is that the first two rules will be used when the

YAWA part of the algorithm isn't needed, either because there are still trees in the forest of the process (which means that the process knows termination can't have occurred yet) or because the process has not been reached by a token yet (init is false, fmc is empty). The second two rules show what a TQRoot should do when the YAWA part of the algorithm should be executed. If the TQRoot is the initiator for the YAWA algorithm, it starts the algorithm, which it denotes by changing the variable *ter* to pending. Furthermore, it sends tokens to its neighbors, and creates the message tree to track these tokens. A TQRoot that isn't initiator for the YAWA algorithm, on the other hand, can simply forward the tokens to its neighbors. Next up come the active non-TQRoot processes that want to receive a message. If their buffer is empty, this means that they must send a confirmation message to the root of the last message chain they received. If there are messages in their buffer, on the other hand, these messages no longer need to be buffered. Finally, all a passive process that wants to receive has to do is block until a message arrives. Its buffer was already cleared when it turned passive, and it also stopped being a TQRoot at that time. In all cases, rather then directly receiving a message, each receive statement causes (among other things) the variable rec to become true, indicating that the process is blocking to receive. Because all other transitions of the basic computation require rec to be false, this makes it impossible for this process to perform actions in the basic computation other then the execution of an actual receive statement (which is rule (a) of definition RECEIVECOMPLETES).

Definition 24 (PASSIVE) *The following rules describe those computation steps that allow a process to turn passive:*

(a) % a TQRoot can turn passive with trees in its messageForest (nonInitiatorBecomesEmpty doesn't do anything)

$(p_i,\ (active,\ True,\ lmc_i,\ f_i,\ init_i,\ fmc_i,\ ter_i,\ False)) \in \mathcal{A},$
$f_i \neq \emptyset,$
$\mathcal{A}' = \mathcal{A} \cup \{(p_i,\ \hat{a}_i[passive]_s[False]_{tq})\} \setminus \{(p_i, \hat{a}_i)\}$

$$\frac{}{(\mathbb{P},\ \mathbb{E},\ \mathcal{A},\ \mathcal{M}_{buff},\ \mathcal{M}_{trans},\ \mathcal{M}_{arr}) \rightarrow_{BTTF} (\mathbb{P},\ \mathbb{E},\ \mathcal{A}',\ \mathcal{M}_{buff},\ \mathcal{M}_{trans},\ \mathcal{M}_{arr})}$$

(b) % a TQRoot can turn passive without trees in its messageForest, with empty fmc and false init (non-InitiatorBecomesEmpty doesn't do anything)

$(p_i,\ (active,\ True,\ lmc_i,\ \emptyset,\ False,\ (),\ ter_i,\ False)) \in \mathcal{A},$
$\mathcal{A}' = \mathcal{A} \cup \{(p_i,\ \hat{a}_i[passive]_s[False]_{tq})\} \setminus \{(p_i, \hat{a}_i)\}$

$$\frac{}{(\mathbb{P},\ \mathbb{E},\ \mathcal{A},\ \mathcal{M}_{buff},\ \mathcal{M}_{trans},\ \mathcal{M}_{arr}) \rightarrow_{BTTF} (\mathbb{P},\ \mathbb{E},\ \mathcal{A}',\ \mathcal{M}_{buff},\ \mathcal{M}_{trans},\ \mathcal{M}_{arr})}$$

(c) % a TQRoot can turn passive without trees in its messageForest, with empty fmc and true init (The process sends a token to all neighbors, ter becomes pending)

$(p_i,\ (active,\ True,\ lmc_i,\ \emptyset,\ True,\ (),\ False,\ False)) \in \mathcal{A},$
$\mathcal{A}' = \mathcal{A} \cup \{(p_i,\ \hat{a}_i[passive]_s[False]_{tq}[\{((id,\ p_i,\ n'),\ ())\}\ ]_f[Pending]_{ter})\} \setminus \{(p_i, \hat{a}_i)\},$
$\mathcal{M}'_{trans} = \mathcal{M}_{trans} \cup \{((Tok,\ ((id,\ p_i,\ n'),\ ()), q),\ (p_i,\ q)) : (p_i,\ q) \in \mathbb{E}\},$
$n' = |\{(p_i,\ q) : (p_i,\ q) \in \mathbb{E}\}|$

$$\frac{}{(\mathbb{P},\ \mathbb{E},\ \mathcal{A},\ \mathcal{M}_{buff},\ \mathcal{M}_{trans},\ \mathcal{M}_{arr}) \rightarrow_{BTTF} (\mathbb{P},\ \mathbb{E},\ \mathcal{A}',\ \mathcal{M}_{buff},\ \mathcal{M}'_{trans},\ \mathcal{M}_{arr})}$$

(d) % a TQRoot can turn passive without trees in its messageForest, with non-empty fmc (The process sends a token to all neighbors)

$(p_i,\ (active,\ True,\ lmc_i,\ \emptyset,\ False,\ fmc_i,\ ter_i,\ False)) \in \mathcal{A},$
$fmc_i \neq (),$
$\mathcal{A}' = \mathcal{A} \cup \{(p_i,\ \hat{a}_i[passive]_s[False]_{tq})\} \setminus \{(p_i, \hat{a}_i)\},$
$\mathcal{M}'_{trans} = \mathcal{M}_{trans} \cup \{((Tok,\ fmc_i + +(id,\ p_i,\ n'), q),\ (p_i,\ q)) : (p_i,\ q) \in \mathbb{E}\},$
$n' = |\{(p_i,\ q) : (p_i,\ q) \in \mathbb{E}\}|$

$(\mathbb{P},\ \mathbb{E},\ \mathcal{A},\ \mathcal{M}_{buff},\ \mathcal{M}_{trans},\ \mathcal{M}_{arr}) \rightarrow_{BTTF} (\mathbb{P},\ \mathbb{E},\ \mathcal{A}',\ \mathcal{M}_{buff},\ \mathcal{M}'_{trans},\ \mathcal{M}_{arr})$

(e) *% an active process can turn passive with empty buffer*
$(p_i,\ (active,\ False,\ ((id,\ q,\ n'),\ lmc'_i),\ f_i,\ init_i,\ fmc_i,\ ter_i,\ False)) \in \mathcal{A},$
$\mathcal{A}' = \mathcal{A} \cup \{(p_i,\ \hat{a}_i[passive]_s[()]_{lmc})\} \setminus \{(p_i, \hat{a}_i)\},$
$\neg\exists((m,\ dest),\ p_i) \in \mathcal{M}_{buff},$
$\mathcal{M}'_{trans} = \mathcal{M}_{trans} \cup \{((Conf,\ ((id,\ q,\ n'),\ lmc'_i),\ q),\ (p_i,\ q'))\},$
q' *is the first node on a path to* q

$(\mathbb{P},\ \mathbb{E},\ \mathcal{A},\ \mathcal{M}_{buff},\ \mathcal{M}_{trans},\ \mathcal{M}_{arr}) \rightarrow_{BTTF} (\mathbb{P},\ \mathbb{E},\ \mathcal{A}',\ \mathcal{M}_{buff},\ \mathcal{M}'_{trans},\ \mathcal{M}_{arr})$

(f) *% an active process can turn passive with non-empty buffer*
$(p_i,\ (active,\ False,\ lmc_i,\ f_i,\ init_i,\ fmc_i,\ ter_i,\ False)) \in \mathcal{A},$
$\mathcal{A}' = \mathcal{A} \cup \{(p_i,\ \hat{a}_i[passive]_s[()]_{lmc})\} \setminus \{(p_i, \hat{a}_i)\},$
$\exists((m,\ dest),\ p_i) \in \mathcal{M}_{buff},$
$\mathcal{M}'_{buff} = \mathcal{M}_{buff} \setminus \{((m_1,\ dest_1),\ p_i),\ \ldots,\ ((m_{n'},\ dest_{n'}),\ p_i)\},$
$\mathcal{M}'_{trans} = \mathcal{M}_{trans} \cup \{((m_1,\ lmc_i + +(id,\ p_i,\ n'),\ dest_1),\ (p_i,\ q_1)),\ \ldots,\ ((m_{n'},\ lmc_i + +(id,\ p_i,\ n'),\ dest_{n'}),\ (p_i,\ q_{n'}))\},$
$\forall i \in \{1,\ \ldots,\ n'\}\ q_i$ *is the first node on a path to* $dest_i$

$(\mathbb{P},\ \mathbb{E},\ \mathcal{A},\ \mathcal{M}_{buff},\ \mathcal{M}_{trans},\ \mathcal{M}_{arr}) \rightarrow_{BTTF} (\mathbb{P},\ \mathbb{E},\ \mathcal{A}',\ \mathcal{M}'_{buff},\ \mathcal{M}'_{trans},\ \mathcal{M}_{arr})$

Our third definition deals with processes turning passive. Notice the similarities with definition RECEIVE. Just as is the case in that definition, the execution of these rules may or may not trigger the execution of the YAWA part of the algorithm. If there are still trees in the messageForest of the process, or if the process has not yet been reached by a token, it is not needed to execute the YAWA part of the algorithm. If the TQRoot is the initiator, and it does not have messageTrees in its messageForest, it starts the YAWA part of the algorithm, which it denotes by changing the variable *ter* to pending. Furthermore, it sends tokens to its neighbors, and creates the message tree to track these tokens. A TQRoot that isn't initiator for the YAWA algorithm, on the other hand, can simply forward the tokens to its neighbors. Next up come the non-TQRoot processes that want to turn passive. If their buffer is empty, this means that they must send a confirmation message to the root of the last message chain they received. If there are messages in their buffer, on the other hand, these messages no longer need to be buffered. In all cases, in addition to the listed changes, the process also changes its state to passive, as that is what we're trying to accomplish.

Definition 25 (DELIVERY) *The following rules describe those computation steps that allow a message to travel through the network to its destination:*

(a) *% a message can arrive at a process*
$\exists((m,\ mc,\ p),\ (q,\ r)) \in \mathcal{M}_{trans},$

$\mathcal{M}'_{trans} = \mathcal{M}_{trans} \backslash \{((m, mc, p), (q, r))\}$,
$\mathcal{M}'_{arr} = \mathcal{M}_{arr} \cup \{((m, mc, p), r)\}$

$(\mathbb{P}, \mathbb{E}, \mathcal{A}, \mathcal{M}_{buff}, \mathcal{M}_{trans}, \mathcal{M}_{arr}) \rightarrow_{BTTF} (\mathbb{P}, \mathbb{E}, \mathcal{A}, \mathcal{M}_{buff}, \mathcal{M}'_{trans}, \mathcal{M}'_{arr})$

(b) % a process may forward a message
$\exists((m, mc, p), q) \in \mathcal{M}_{arr}$,
$p \neq q$,
$\mathcal{M}'_{arr} = \mathcal{M}_{arr} \backslash \{((m, mc, p), q)\}$,
$\mathcal{M}'_{trans} = \mathcal{M}_{trans} \cup \{((m, mc, p), (q, q'))\}$,
q' is the first node on a path to p

$(\mathbb{P}, \mathbb{E}, \mathcal{A}, \mathcal{M}_{buff}, \mathcal{M}_{trans}, \mathcal{M}_{arr}) \rightarrow_{BTTF} (\mathbb{P}, \mathbb{E}, \mathcal{A}, \mathcal{M}_{buff}, \mathcal{M}'_{trans}, \mathcal{M}'_{arr})$

That completes the possible transitions in the basic computation. However, there are still a number of transitions in the BTTF Wave algorithm that need to be covered. First of all, messages that have departed from their sender need to arrive at their destination in finite time. The rules of this definition ensure that will happen. The first rule allows a message to move from a channel to a process. The second rule allows a process that is not the destination of a message to forward a message along the path over the least nodes to its destination. Each of these two rules reduce the distance between the message and its destination, ensuring that the message will eventually arrive.

Definition 26 (RECEIVECOMPLETES) *The following rule describes the computation step that allows a process that is ready to receive can actually receive a message:*

(a) % a process that's blocking to receive can complete its receive
$(p_i, (s_i, False, (), f_i, init_i, fmc_i, ter_i, True)) \in \mathcal{A}$,
$\mathcal{A}' = \mathcal{A} \cup \{(p_i, \hat{a}_i[active]_s[mc]_{lmc}[False]_{rec})\} \setminus \{(p_i, \hat{a}_i)\}$,
$\exists((m, mc, p_i), p_i) \in \mathcal{M}_{arr}$,
$m \neq Tok \vee Conf$,
$\mathcal{M}'_{arr} = \mathcal{M}_{arr} \backslash \{((m, mc, p_i), p_i)\}$

$(\mathbb{P}, \mathbb{E}, \mathcal{A}, \mathcal{M}_{buff}, \mathcal{M}_{trans}, \mathcal{M}_{arr}) \rightarrow_{BTTF} (\mathbb{P}, \mathbb{E}, \mathcal{A}', \mathcal{M}_{buff}, \mathcal{M}_{trans}, \mathcal{M}'_{arr})$

After a message has been delivered to its destination, it still needs to be actually received by the destination process. This requires the destination process to be blocking to receive. All rules that cause a process to be blocking to receive also turn *tqroot* to $False$ and *lmc* to (), so we can require those variables to have those values. As a result of receiving a message, the process will turn active, and the messageChain accompanying the message that is received will be stored in *lmc*. Finally, when the process completes the receive, it is no longer blocking to receive.

Definition 27 (CONFIRMATION) *The following rules describe the computation steps which describe what happens when a process receives a confirmation message:*

(a) (1) % a confirmation message can arrive at a process without completing a messageTree
$(p_i, (s_i, tqroot_i, lmc_i, \{t_1, \ldots, ((id, p_i, n'), (t'_1, \ldots, t'_k)), \ldots, t_l\}, init_i, fmc_i, ter_i, rec_i)) \in \mathcal{A}$,
$\mathcal{A}' = \mathcal{A} \cup \{(p_i, \hat{a}_i[\{t_1, \ldots, ((id, p_i, n'), (t'_1, \ldots, t'_k)) + ((id, p_i, n'), mc), \ldots, t_l\}]_f\ )\} \setminus \{(p_i, \hat{a}_i)\}$,
$((id, p_i, n'), (t'_1, \ldots, t'_k)) + ((id, p_i, n'), mc)$ *is not a complete messageTree,*

$\exists((Conf,\ ((id,\ p_i,\ n'),\ mc),\ p_i),\ p_i) \in \mathcal{M}_{arr}$,
$\mathcal{M}'_{arr} = \mathcal{M}_{arr} \backslash \{((Conf,\ ((id,\ p_i,\ n'),\ mc),\ p_i),\ p_i)\}$

$$\overline{(\mathbb{P},\ \mathbb{E},\ \mathcal{A},\ \mathcal{M}_{buff},\ \mathcal{M}_{trans},\ \mathcal{M}_{arr}) \rightarrow_{BTTF} (\mathbb{P},\ \mathbb{E},\ \mathcal{A}',\ \mathcal{M}_{buff},\ \mathcal{M}_{trans},\ \mathcal{M}'_{arr})}$$

(2) *% a confirmation message can arrive at a process, completing a messageTree, with multiple trees in the forest*
$(p_i,\ (s_i,\ tqroot_i,\ lmc_i,\ \{t_1,\ \ldots,\ t_{j-1},\ ((id,\ p_i,\ n'),\ (t'_1,\ \ldots,\ t'_k)),\ t_{j+1},\ \ldots,\ t_l\},\ init_i,\ fmc_i,\ ter_i,\ rec_i)) \in \mathcal{A}$,
$l >= 2$,
$\mathcal{A}' = \mathcal{A} \cup \{(p_i,\ \hat{a}_i[\{t_1,\ \ldots,\ t_{j-1},\ t_{j+1},\ \ldots,\ t_l\}]_f\ )\} \setminus \{(p_i, \hat{a}_i)\}$,
$((id,\ p_i,\ n'),\ (t'_1,\ \ldots,\ t'_k)) + ((id,\ p_i,\ n'),\ mc)$ *is a complete messageTree,*
$\exists((Conf,\ ((id,\ p_i,\ n'),\ mc),\ p_i),\ p_i) \in \mathcal{M}_{arr}$,
$\mathcal{M}'_{arr} = \mathcal{M}_{arr} \backslash \{((Conf,\ ((id,\ p_i,\ n'),\ mc),\ p_i),\ p_i)\}$

$$\overline{(\mathbb{P},\ \mathbb{E},\ \mathcal{A},\ \mathcal{M}_{buff},\ \mathcal{M}_{trans},\ \mathcal{M}_{arr}) \rightarrow_{BTTF} (\mathbb{P},\ \mathbb{E},\ \mathcal{A}',\ \mathcal{M}_{buff},\ \mathcal{M}_{trans},\ \mathcal{M}'_{arr})}$$

(b) *% a confirmation message can arrive at a process, completing a messageTree, with a single tree in the forest, with tqroot = false, empty fmc, false init (nonInitiatorBecomesEmpty doesn't do anything)*
$(p_i,\ (s_i,\ False,\ lmc_i,\ \{((id,\ p_i,\ n'),\ (t_1,\ \ldots,\ t_k))\},\ False,\ (),\ ter_i,\ rec_i)) \in \mathcal{A}$,
$\mathcal{A}' = \mathcal{A} \cup \{(p_i,\ \hat{a}_i[\emptyset]_f)\} \setminus \{(p_i, \hat{a}_i)\}$,
$((id,\ p_i,\ n'),\ (t_1,\ \ldots,\ t_k)) + ((id,\ p_i,\ n'),\ mc)$ *is a complete messageTree,*
$\exists((Conf,\ ((id,\ p_i,\ n'),\ mc),\ p_i),\ p_i) \in \mathcal{M}_{arr}$,
$\mathcal{M}'_{arr} = \mathcal{M}_{arr} \backslash \{((Conf,\ ((id,\ p_i,\ n'),\ mc),\ p_i),\ p_i)\}$

$$\overline{(\mathbb{P},\ \mathbb{E},\ \mathcal{A},\ \mathcal{M}_{buff},\ \mathcal{M}_{trans},\ \mathcal{M}_{arr}) \rightarrow_{BTTF} (\mathbb{P},\ \mathbb{E},\ \mathcal{A}',\ \mathcal{M}_{buff},\ \mathcal{M}_{trans},\ \mathcal{M}'_{arr})}$$

(c) *% a confirmation message can arrive at a process, completing a messageTree, with a single tree in the forest, with tqroot = false, empty fmc, true init, false ter (init sends token to all neighbors)*
$(p_i,\ (s_i,\ False,\ lmc_i,\ \{((id,\ p_i,\ n'),\ (t_1,\ \ldots,\ t_k))\},\ True,\ (),\ False,\ rec_i)) \in \mathcal{A}$,
$\mathcal{A}' = \mathcal{A} \cup \{(p_i,\ \hat{a}_i[\{((id', p_i, n''),\ ())\}]_f[Pending]_{ter})\} \setminus \{(p_i, \hat{a}_i)\}$,
$((id,\ p_i,\ n'),\ (t_1,\ \ldots,\ t_k)) + ((id,\ p_i,\ n'),\ mc)$ *is a complete messageTree,*
$\exists((Conf,\ ((id,\ p_i,\ n'),\ mc),\ p_i),\ p_i) \in \mathcal{M}_{arr}$,
$\mathcal{M}'_{arr} = \mathcal{M}_{arr} \backslash \{((Conf,\ ((id,\ p_i,\ n'),\ mc),\ p_i),\ p_i)\}$,
$\mathcal{M}'_{trans} = \mathcal{M}_{trans} \cup \{((Tok,\ ((id', p_i, n''),\ ()),\ q),\ (p_i,\ q)) : (p_i,\ q) \in \mathbb{E}\}$,
$n'' = |\{(p_i,\ q) : (p_i,\ q) \in \mathbb{E}\}|$

$$\overline{(\mathbb{P},\ \mathbb{E},\ \mathcal{A},\ \mathcal{M}_{buff},\ \mathcal{M}_{trans},\ \mathcal{M}_{arr}) \rightarrow_{BTTF} (\mathbb{P},\ \mathbb{E},\ \mathcal{A}',\ \mathcal{M}_{buff},\ \mathcal{M}'_{trans},\ \mathcal{M}'_{arr})}$$

(d) *% a confirmation message can arrive at a process, completing a messageTree, with a single tree in the forest, with tqroot = false, non-empty fmc (process sends token to all neighbors)*
$(p_i,\ (s_i,\ False,\ lmc_i,\ \{((id,\ p_i,\ n'),\ (t_1,\ \ldots,\ t_k))\},\ False,\ fmc_i,\ ter_i,\ rec_i)) \in \mathcal{A}$,
$fmc_i \neq ()$,
$\mathcal{A}' = \mathcal{A} \cup \{(p_i,\ \hat{a}_i[\emptyset]_f)\} \setminus \{(p_i, \hat{a}_i)\}$,
$((id,\ p_i,\ n'),\ (t_1,\ \ldots,\ t_k)) + ((id,\ p_i,\ n'),\ mc)$ *is a complete messageTree,*
$\exists((Conf,\ ((id,\ p_i,\ n'),\ mc),\ p_i),\ p_i) \in \mathcal{M}_{arr}$,

$$\frac{\begin{array}{l}\mathcal{M}'_{arr} = \mathcal{M}_{arr} \backslash \{((Conf,\ ((id,\ p_i,\ n'),\ mc),\ p_i),\ p_i)\},\\ \mathcal{M}'_{trans} = \mathcal{M}_{trans} \cup \{((Tok,\ fmc_i ++ (id', p_i, n''),\ q),\ (p_i,\ q)) : (p_i,\ q) \in \mathbb{E}\},\\ n'' = |\{(p_i,\ q) : (p_i,\ q) \in \mathbb{E}\}|\end{array}}{(\mathbb{P},\ \mathbb{E},\ \mathcal{A},\ \mathcal{M}_{buff},\ \mathcal{M}_{trans},\ \mathcal{M}_{arr}) \rightarrow_{BTTF} (\mathbb{P},\ \mathbb{E},\ \mathcal{A}',\ \mathcal{M}_{buff},\ \mathcal{M}'_{trans},\ \mathcal{M}'_{arr})}$$

(e) % a confirmation message can arrive at a process, completing a messageTree, with a single tree in the forest, and with tqroot = true (nonInitiatorBecomesEmpty doesn't do anything)

$$\frac{\begin{array}{l}(p_i,\ (s_i,\ True,\ lmc_i,\ \{((id,\ p_i,\ n'),\ (t_1,\ \ldots,\ t_k))\},\ init_i,\ fmc_i,\ ter_i,\ rec_i)) \in \mathcal{A},\\ \mathcal{A}' = \mathcal{A} \cup \{(p_i,\ \hat{a}_i[\emptyset]_f)\} \setminus \{(p_i, \hat{a}_i)\},\\ ((id,\ p_i,\ n'),\ (t_1,\ \ldots,\ t_k)) + ((id,\ p_i,\ n'),\ mc) \text{ is a complete messageTree,}\\ \exists((Conf,\ ((id,\ p_i,\ n'),\ mc),\ p_i),\ p_i) \in \mathcal{M}_{arr},\\ \mathcal{M}'_{arr} = \mathcal{M}_{arr} \backslash \{((Conf,\ ((id,\ p_i,\ n'),\ mc),\ p_i),\ p_i)\}\end{array}}{(\mathbb{P},\ \mathbb{E},\ \mathcal{A},\ \mathcal{M}_{buff},\ \mathcal{M}_{trans},\ \mathcal{M}_{arr}) \rightarrow_{BTTF} (\mathbb{P},\ \mathbb{E},\ \mathcal{A}',\ \mathcal{M}_{buff},\ \mathcal{M}_{trans},\ \mathcal{M}'_{arr})}$$

(f) % a confirmation message can arrive at a process, completing a messageTree, with a single tree in the forest, with tqroot = false, empty fmc, true init, pending ter (declare termination!)

$$\frac{\begin{array}{l}(p_i,\ (s_i,\ False,\ lmc_i,\ \{((id,\ p_i,\ n'),\ (t_1,\ \ldots,\ t_k))\},\ True,\ (),\ Pending,\ rec_i)) \in \mathcal{A},\\ \mathcal{A}' = \mathcal{A} \cup \{(p_i,\ \hat{a}_i[\emptyset]_f[True]_{ter})\} \setminus \{(p_i, \hat{a}_i)\},\\ ((id,\ p_i,\ n'),\ (t_1,\ \ldots,\ t_k)) + ((id,\ p_i,\ n'),\ mc) \text{ is a complete messageTree,}\\ \exists((Conf,\ ((id,\ p_i,\ n'),\ mc),\ p_i),\ p_i) \in \mathcal{M}_{arr},\\ \mathcal{M}'_{arr} = \mathcal{M}_{arr} \backslash \{((Conf,\ ((id,\ p_i,\ n'),\ mc),\ p_i),\ p_i)\}\end{array}}{(\mathbb{P},\ \mathbb{E},\ \mathcal{A},\ \mathcal{M}_{buff},\ \mathcal{M}_{trans},\ \mathcal{M}_{arr}) \rightarrow_{BTTF} (\mathbb{P},\ \mathbb{E},\ \mathcal{A}',\ \mathcal{M}_{buff},\ \mathcal{M}_{trans},\ \mathcal{M}'_{arr})}$$

This definition handles confirmation messages that arrive at the process that holds the appropriate messageTree. The main thing that needs to happen then is that the messageChain is added to the appropriate messageTree. However, if this causes the messageTree to become complete, the messageTree is removed from the messageForest. If this causes the messageForest to become empty, it may be necessary to perform some steps of the YAWA part of the algorithm. This again causes a large number of possible rules. We again would like to highlight the similarities with definition RECEIVE. Both rule (a1) and rule (a2) don't perform any steps of the YAWA algorithm as there are still trees in the messageForest of the process, just as is the case in rule (a) of definition RECEIVE. Rules (b), (c) and (d) mirror the corresponding rules from definition RECEIVE in the same way. Rule (e) does not have a corresponding rule in definition RECEIVE, because all rules in definition RECEIVE turn *tqroot* to *False*. However, there is one rule we want to highlight. If *ter* was *Pending*, the initiator had initiated the YAWA part of the algorithm. If the messageForest of that process becomes empty, then the receipt of all tokens has been confirmed. This indicates that the entire system is passive, and that no messages are in transit, indicating termination.

Definition 28 (TOKEN) *The following rules describe the computation steps which describe what happens when a process receives a token message:*

(a) % a token can be received by a non-initiator with empty fmc with non-empty messageForest

$(p_i,\ (s_i,\ tqroot_i,\ lmc_i,\ f_i,\ False,\ (),\ ter_i,\ rec_i)) \in \mathcal{A}$ *where* $f_i \neq \emptyset$,

$\mathcal{A}' = \mathcal{A} \cup \{(p_i,\ \hat{a}_i[mc]_{fmc})\} \setminus \{(p_i, \hat{a}_i)\}$,

$\exists((Tok,\, mc,\, p_i),\, p_i) \in \mathcal{M}_{arr}$,
$\mathcal{M'}_{arr} = \mathcal{M}_{arr} \backslash \{((Tok,\, mc,\, p_i),\, p_i)\}$

$$\frac{}{(\mathbb{P},\, \mathbb{E},\, \mathcal{A},\, \mathcal{M}_{buff},\, \mathcal{M}_{trans},\, \mathcal{M}_{arr}) \rightarrow_{BTTF} (\mathbb{P},\, \mathbb{E},\, \mathcal{A}',\, \mathcal{M}_{buff},\, \mathcal{M}_{trans},\, \mathcal{M'}_{arr})}$$

(b) % a token can be received by a non-initiator with non-empty fmc
$(p_i,\, (s_i,\, tqroot_i,\, lmc_i,\, f_i,\, False,\, fmc_i,\, ter_i,\, rec_i)) \in \mathcal{A}$,
$fmc_i \neq ()$,
$\exists((Tok,\, ((id,\, q,\, n'),\, lmc_i'),\, p_i),\, p_i) \in \mathcal{M}_{arr}$,
$\mathcal{M'}_{arr} = \mathcal{M}_{arr} \backslash \{((Tok,\, ((id,\, q,\, n'),\, lmc_i'),\, p_i),\, p_i)\}$,
$\mathcal{M'}_{trans} = \mathcal{M}_{trans} \cup \{((Conf,\, ((id,\, q,\, n'),\, lmc_i'),\, q),\, (p_i,\, q'))\}$,
q' *is the first node on a path to* q

$$\frac{}{(\mathbb{P},\, \mathbb{E},\, \mathcal{A},\, \mathcal{M}_{buff},\, \mathcal{M}_{trans},\, \mathcal{M}_{arr}) \rightarrow_{BTTF} (\mathbb{P},\, \mathbb{E},\, \mathcal{A},\, \mathcal{M}_{buff},\, \mathcal{M'}_{trans},\, \mathcal{M'}_{arr})}$$

(c) % a token can be received by the initiator
$(p_i,\, (s_i,\, False,\, (),\, f_i,\, True,\, (),\, Pending,\, rec_i)) \in \mathcal{A}$,
$\exists((Tok,\, mc,\, p_i),\, p_i) \in \mathcal{M}_{arr}$,
$\mathcal{M'}_{arr} = \mathcal{M}_{arr} \backslash \{((Tok,\, mc,\, p_i),\, p_i)\} \cup \{((Conf,\, mc,\, p_i),\, p_i)\}$

$$\frac{}{(\mathbb{P},\, \mathbb{E},\, \mathcal{A},\, \mathcal{M}_{buff},\, \mathcal{M}_{trans},\, \mathcal{M}_{arr}) \rightarrow_{BTTF} (\mathbb{P},\, \mathbb{E},\, \mathcal{A},\, \mathcal{M}_{buff},\, \mathcal{M}_{trans},\, \mathcal{M'}_{arr})}$$

(d) % a token can be received by a non-initiator with empty fmc, empty messageForest and tqroot = False
$(p_i,\, (s_i,\, False,\, lmc_i,\, \emptyset,\, False,\, (),\, ter_i,\, rec_i)) \in \mathcal{A}$,
$\mathcal{A}' = \mathcal{A} \cup \{(p_i,\, \hat{a}_i[mc]_{fmc})\} \setminus \{(p_i, \hat{a}_i)\}$,
$\exists((Tok,\, mc,\, p_i),\, p_i) \in \mathcal{M}_{arr}$,
$\mathcal{M'}_{arr} = \mathcal{M}_{arr} \backslash \{((Tok,\, mc,\, p_i),\, p_i)\}$,
$\mathcal{M'}_{trans} = \mathcal{M}_{trans} \cup \{((Tok,\, mc + +(id,\, p_i,\, n'),\, q),\, (p_i,\, q)) : (p_i,\, q) \in \mathbb{E}\}$,
$n' = |\{(p_i,\, q) : (p_i,\, q) \in \mathbb{E}\}|$

$$\frac{}{(\mathbb{P},\, \mathbb{E},\, \mathcal{A},\, \mathcal{M}_{buff},\, \mathcal{M}_{trans},\, \mathcal{M}_{arr}) \rightarrow_{BTTF} (\mathbb{P},\, \mathbb{E},\, \mathcal{A}',\, \mathcal{M}_{buff},\, \mathcal{M'}_{trans},\, \mathcal{M'}_{arr})}$$

(e) % a token can be received by a non-initiator with empty fmc with tqroot = True
$(p_i,\, (s_i,\, True,\, lmc_i,\, f_i,\, False,\, (),\, ter_i,\, rec_i)) \in \mathcal{A}$,
$\mathcal{A}' = \mathcal{A} \cup \{(p_i,\, \hat{a}_i[mc]_{fmc})\} \setminus \{(p_i, \hat{a}_i)\}$,
$\exists((Tok,\, mc,\, p_i),\, p_i) \in mathcalM_{arr}$,
$\mathcal{M'}_{arr} = \mathcal{M}_{arr} \backslash \{((Tok,\, mc,\, p_i),\, p_i)\}$

$$\frac{}{(\mathbb{P},\, \mathbb{E},\, \mathcal{A},\, \mathcal{M}_{buff},\, \mathcal{M}_{trans},\, \mathcal{M}_{arr}) \rightarrow_{BTTF} (\mathbb{P},\, \mathbb{E},\, \mathcal{A}',\, \mathcal{M}_{buff},\, \mathcal{M}_{trans},\, \mathcal{M'}_{arr})}$$

The next definition deals with the receipt of a token by a process. Again, there is a bit of similarity to definitions RECEIVE, PASSIVE and CONFIRMATION. The behavior in rules (a) and (d) is virtually the same in each of these definitions. Furthermore, the behavior of rule (e) is the same in this definition as in definition CONFIRMATION. In all these cases, if the process is still aware of activity (because its messageForest isn't empty or because it is a TQRoot), the token is stored in the variable first messageChain, to be processed when the process is no longer responsible for any activity in the system. On the other hand, when the first token arrives at a process that is not responsible for any activity in the system, this process forwards that token to its neighbors.

The similarities are not complete, however. In the other three definitions, rule (b) dealt with the behavior of a process that had not yet been reached by a token message. For obvious reasons, there is no need to consider that possibility here. Instead, this rule will deal with the behavior of a process that had been reached by a token message before. This process either already forwarded that token, or it will do so in the future. The newly arrived token can therefore just be confirmed. Similarly, rule (c) deals with the behavior of the initiator. A token arriving at the initiator is simply transformed into a confirmation message, as the presence of this token confirms to the initiator that all tokens that caused this token have been received by their destination.

Definition 29 (INITIATION) *The following rule describes the computation step that allows the initiator to initiate YAWA if it was never a TQRoot:*

(a) % The initiator, which was never a TQRoot, can initiate YAWA

$$\frac{\begin{array}{l}(p_i,\ (s_i,\ False,\ lmc_i,\ \emptyset,\ True,\ (),\ False,\ rec_i)) \in \mathcal{A},\\ \mathcal{A}' = \mathcal{A} \cup \{(p_i,\ \hat{a}_i[\{((id,\ p_i,\ n'),\ ())\}]_f[Pending]_{ter})\} \setminus \{(p_i, \hat{a}_i)\},\\ \mathcal{M'}_{trans} = \mathcal{M}_{trans} \cup \{((Tok,\ ((id,\ p_i,\ n'),\ ()),\ q),\ (p_i,\ q)) : (p_i,\ q) \in \mathbb{E}\},\\ n' = |\{(p_i,\ q) : (p_i,\ q) \in \mathbb{E}\}|\end{array}}{(\mathbb{P},\ \mathbb{E},\ \mathcal{A},\ \mathcal{M}_{buff},\ \mathcal{M}_{trans},\ \mathcal{M}_{arr}) \rightarrow_{BTTF} (\mathbb{P},\ \mathbb{E},\ \mathcal{A}',\ \mathcal{M}_{buff},\ \mathcal{M'}_{trans},\ \mathcal{M}_{arr})}$$

This leaves one more transition to be discussed. It is possible that the initiator never was a TQRoot. In that case, it will not initiate the YAWA part of the algorithm when a TQRoot turns passive, nor will it initiate the YAWA algorithm when a TQRoot receives a message. To allow the initiator to initiate the YAWA algorithm, a special rule is needed, which is introduced in this definition.

In the rest of the text, we will refer to rules by the number of their definition, followed by the letter of the rule.

Definition 30 *A computation is defined as a (potentially infinite) series of configurations* $(c_1,\ c_2,\ \ldots)$, *where*
$\forall i : c_i \in \mathbb{C}$, *and*
c_1 *is a potential initial configuration, and*
$\forall i : c_i \rightarrow_{BTTF} c_{i+1}$ *if both* c_i *and* c_{i+1} *exist, and*
if the computation $(c_1,\ c_2,\ \ldots,\ c_n)$ *is finite,* $\neg\exists c \in \mathbb{C} : c_n \rightarrow_{BTTF} c$.

The notion of a computation step is used to define a computation as simply a series of transitions made from an initial configuration. Furthermore, we demand that a computation takes as many steps as possible.

Definition 31 *A computation is called a terminating computation if it is a (potentially infinite) computation* $(c_1,\ c_2,\ \ldots,\ c_n,\ \ldots)$, *where* c_n *is a terminated configuration.*

The notion of a terminating computation is somewhat interesting, in the sense that the terminated configuration is not necessary the last configuration of the computation. The reason for this is that even though the basic computation may have terminated, the BTTF Wave algorithm may still execute steps. Of course, we will see later that these steps of the BTTF Wave algorithm do not change the configuration from a terminated to a non-terminated one. Further, we will see that a terminating computation is always finite. However, these properties will be proven later, and are not part of the definition.

5 *Correctness Proof*

In this section, we will prove that the BTTF Wave algorithm as has been described previously is a correct implementation of a termination detection algorithm. First, in subsection 5.1, we will prove a number of auxiliary Lemmas which will be useful in later sections. Next, in subsection 5.2, we will prove that the formal description correctly implements all possible transitions in a basic computation, which implies that the main computation is not altered by running the BTTF Wave algorithm. We continue by proving the safety of the algorithm in subsection 5.3, where we demonstrate that if BTTF Wave indicates termination, the main computation has actually terminated. Finally, in subsection 5.4, we prove that if the main computation terminates, the BTTF Wave algorithm will eventually indicate this.

5.1 Auxiliary Lemmas

As mentioned before, we will first prove a number of auxiliary Lemmas. The first three Lemmas deal with messageChains, showing that they are non-empty (Lemma 32), that all messages in the system are accompanied by a messageChain (Lemma 33), and that every messageChain is part of a messageTree (Lemma 34). Lemma 35 shows that once a terminated configuration has been reached, all further configurations in a computation will be terminated configurations. If this wasn't true, termination wouldn't be a stable property and termination detection wouldn't make sense. Finally, Lemma 36 shows that once a terminated configuration has been reached, the algorithm will only perform a finite number of steps, which forms an essential part of proving the liveliness of the algorithm.

Lemma 32 *In a terminating computation* $(c_1, c_2, \ldots, c_n, \ldots)$,
$\forall c_i = (\mathbb{P}, \mathbb{E}, \mathcal{A}, \mathcal{M}_{buff}, \mathcal{M}_{trans}, \mathcal{M}_{arr}) \in \mathbb{C}$:
$\forall((m, mc, dest), p) \in \mathcal{M}_{arr}, ((m, mc, dest), (p, q)) \in \mathcal{M}_{trans}$:
$mc \neq ()$

Proof.

1. As $c_1 = (\mathbb{P}, \mathbb{E}, \mathcal{A}', \mathcal{M}'_{buff}, \mathcal{M}'_{trans}, \mathcal{M}'_{arr})$ is a potential initial configuration, $\mathcal{M}'_{arr} = \emptyset$ and $\mathcal{M}'_{trans} = \emptyset$, which implies $\neg\exists((m, mc, dest), p) \in \mathcal{M}'_{arr} \wedge \neg\exists((m, mc, dest), (p, q)) \in \mathcal{M}'_{trans}$. This implies that there must have been a computation step that created $((m, mc, p), \_)$ at some process or channel, when such a message did not exist before.

2. There are only a few rules where a messageChain can be created. A TQRoot can create a messageChain in rule SEND(a), and a non-TQRoot can create a messageChain when it flushes the buffer in rule (f) of definition RECEIVE or PASSIVE.

3. In all these rules, a messageInfo is added to the messageChain, which implies that $mc \neq ()$. □

In this Lemma, we prove that if a messageChain accompanies a message, such a messageChain is not empty.

Lemma 33 *In a terminating computation* $(c_1, c_2, \ldots, c_n, \ldots)$,
$\forall c_i = (\mathbb{P}, \mathbb{E}, \mathcal{A}, \mathcal{M}_{buff}, \mathcal{M}_{trans}, \mathcal{M}_{arr}) \in \mathbb{C}$:
$((m, dest), p) \in \mathcal{M}_{buff} \rightarrow (p, (active, False, lmc, \_, \_, \_, \_, False)) \in \mathcal{A}$ *with* $lmc \neq ()$

Proof.

1. Let $c_i = (\mathbb{P}, \mathbb{E}, \mathcal{A}, \mathcal{M}_{buff}, \mathcal{M}_{trans}, \mathcal{M}_{arr}) \in \mathbb{C}$ and let $((m, dest), p) \in \mathcal{M}_{buff}$.

2. Then there existed a series of computation steps that changed c_1 into c_i.

3. Let us consider those computation steps that made changes to process p.
4. As $c_1 = (\mathbb{P}, \mathbb{E}, \mathcal{A}', \mathcal{M}'_{buff}, \mathcal{M}'_{trans}, \mathcal{M}'_{arr})$ is a potential initial configuration, $\mathcal{M}'_{buff} = \emptyset$, which implies $\neg\exists((m, \mathit{dest}), p) \in \mathcal{M}'_{buff}$.
5. This means that there must have been a computation step that created $((m, \mathit{dest}), p)$. Let us say that this computation step changed c_k to c_l.
6. This must have been a computation step that executed rule SEND(b), because all other rules either don't create messages, create messages in a channel, or create messages accompanied by a messageChain.
7. After executing rule SEND(b), $(p, \mathit{active}, \mathit{False}, \mathit{lmc}, f, \mathit{init}, \mathit{fmc}, \mathit{ter}, \mathit{False})$ was in the set of attributes.
8. After that rule was executed, all rules either were inapplicable on this process because the required attributes did not match, were inapplicable because their application would have removed $((m, \mathit{dest}), p)$, or could be applied but would not have changed the relevant attributes.
9. This implies that $(p, (\mathit{active}, \mathit{False}, \mathit{lmc}, f, \mathit{init}, \mathit{fmc}, \mathit{ter}, \mathit{False})) \in \mathcal{A}$.
10. Now all that remains to be shown is that $\mathit{lmc} \neq ()$.
11. As rule SEND(b) was applied on configuration $c_k = (\mathbb{P}, \mathbb{E}, \mathcal{A}'', \mathcal{M}''_{buff}, \mathcal{M}''_{trans}, \mathcal{M}''_{arr})$, $(p, (\mathit{active}, \mathit{False}, \mathit{lmc}, f, \mathit{init}, \mathit{fmc}, \mathit{ter}, \mathit{False})) \in \mathcal{A}''$.
12. As c_1 is a potential initial configuration, either $(p, (\mathit{active}, \mathit{True}, \_, \_, \_, \_, \_, \mathit{False})) \in \mathcal{A}'$ or $(p, (\mathit{passive}, \mathit{False}, \_, \_, \_, \_, \_, \mathit{False})) \in \mathcal{A}'$. This means that there must have been a computation step that changed the attributes of p to $(p, (\mathit{active}, \mathit{False}, \_, \_, \_, \_, \_, \mathit{False}))$.
13. The only computation step where $(p, (\mathit{active}, \mathit{False}, \_, \_, \_, \_, \_, \mathit{False}))$ is in the result of the rule but not in the requirements of the rule, is rule RECEIVECOMPLETES(a). When this rule was executed, there existed a message $((m, mc, p), p) \in \mathcal{M}_{arr}$ where $m \neq \mathit{Tok} \vee \mathit{Conf}$.
14. According to Lemma 32, $mc \neq ()$, which implies that $mc \neq ()$ and by extension that $\mathit{lmc} \neq ()$. □

This Lemma shows that any message that is not accompanied by a messageChain is located at a process with a non-empty messageChain.

Lemma 34 *In a terminating computation* $(c_1, c_2, \ldots, c_n, \ldots)$,
$\forall c_i = (\mathbb{P}, \mathbb{E}, \mathcal{A}, \mathcal{M}_{buff}, \mathcal{M}_{trans}, \mathcal{M}_{arr}) \in \mathbb{C}:$
$(\exists(p, (\_, \_, mc, \_, \_, \_, \_, \_)) \in \mathcal{A} \text{ with } mc \neq () \vee$
$\exists((m, mc, dest), (p, p')) \in \mathcal{M}_{trans} \vee$
$\exists((m, mc, dest), p) \in \mathcal{M}_{arr})$
$\rightarrow \exists(q, (\_, \_, \_, mf, \_, \_, \_, \_)) \in \mathcal{A}:$
$\exists t \in mf : t \text{ is not a complete messageTree} \wedge$
$mc = (MI, mc') \wedge$
$t = (MI, (T_1, \ldots, T_{n'}))$.

Proof.

1. Let $c_i = (\mathbb{P}, \mathbb{E}, \mathcal{A}, \mathcal{M}_{buff}, \mathcal{M}_{trans}, \mathcal{M}_{arr}) \in \mathbb{C}$.
2. Assume $(\exists(p, (\_, \_, mc, \_, \_, \_, \_, \_)) \in \mathcal{A}$ with $mc \neq () \vee$
$\exists((m, mc, dest), (p, p')) \in \mathcal{M}_{trans} \vee$
$\exists((m, mc, dest), p) \in \mathcal{M}_{arr})$

3. If $\exists((m, mc, dest), (p, p')) \in \mathcal{M}_{trans} \vee \exists((m, mc, dest), p) \in \mathcal{M}_{arr})$ then by Lemma 32, $mc \neq ()$.

4. If $(\exists(p, (\_, \_, mc, \_, \_, \_, \_, \_)) \in \mathcal{A}$ with $mc \neq ()$ then $mc \neq ()$ by assumption.

5. Therefore, $mc \neq ()$. So let us say that $mc = (MI, mc')$ (which follows from the definition of a messageChain).

6. As $c_1 = (\mathbb{P}, \mathbb{E}, \mathcal{A}', \mathcal{M}'_{buff}, \mathcal{M}'_{trans}, \mathcal{M}'_{arr})$ is a potential initial configuration, $\forall p_i \in \mathbb{P} : mc = () \wedge \neg\exists((m, mc, dest), (p, p')) \in \mathcal{M}'_{trans} \wedge \exists((m, mc, dest), p) \in \mathcal{M}'_{arr}$. This implies that there must exists at least one computation step that manipulated mc.

7. Since the ++ operation always adds a messageInfo to the end of the messageChain, MI can only have been added to this chain when either $MI + +()$ was executed or when a messageChain with MI as first element was created directly when it did not exist before.

8. All computation steps that execute the ++ operation, do this on a messageChain that was accompanying a message (which is not empty according to Lemma 32), unless the executed rule is rule (f) of definition RECEIVE or PASSIVE. However, these rules require the presence of a message at the process, which means that Lemma 33 can be applied. From there, it follows that the variable lmc, on which they execute the ++ operation, is not empty. We must conclude that $MI + +()$ was not executed in order to add MI to the messageChain.

9. All rules that create a messageChain with MI as first element when it did not exist before, also add a tree to a forest in some process with MI at its root. We will look at this tree more in depth.

10. Let $c_j = (\mathbb{P}, \mathbb{E}, \mathcal{A}'', \mathcal{M}''_{buff}, \mathcal{M}''_{trans}, \mathcal{M}''_{arr})$ with $j =< i$ be the last configuration in which a tree with this root existed. Let $T = (MI, (T_1, \ldots T_n))$ be that tree as it existed at that time.

11. Now $mc = (MI, mc')$ and $T = (MI, (T_1, \ldots T_n))$ have the same messageInfo MI as the first element of their list. Now either $\exists i$ such that T_i and mc' have the same messageInfo as the first element of their list, or $\neg\exists i$ such that T_i and mc' have the same messageInfo as the first element of their list.

12. if $\exists i$ such that T_i and mc' have the same messageInfo as the first element of their list, this process can be repeated, until eventually a messageChain mc'' and a tree T' are found such that T' and mc'' have the same messageInfo as first element, but none of the children of T' has as its first element the same messageInfo as the second messageInfo in mc''. This happens because the length of mc' is finite, as only a finite number of computation steps have been performed. Let $MI' = (id, r, n)$ be this last matching messageInfo.

13. Note that $n \neq 0$, because all rules that add MI' to mc either explicitly state this, because it is part of the requirements for applying the rule, or because the rule creates a messageChain for each channel departing from the creating process (the number of which is greater then 0 due to the fact that the network is strongly connected). Also note how the rules that add MI' to mc always create a number of different messageChains equal to n.

14. Also note that there will be no further creation of messageChains ending on this same messageInfo, because new calls to these rules will create a new identifier. This implies that if children will be added to the node MI', it will be through $Conf$ messages that resulted from one of these n messageChains.

15. Each of the messageChains created in these rules will eventually arrive at a process. The arrival of a messageChain at a process may result in the application of the rule from definition RECEIVECOMPLETES. In that case, either a $Conf$ message will be created later with the same

messageChain, or a single node will be added as a child to MI' in the messageChain. Similarly, the arrival of a messageChain at a process may be the arrival of a Tok message. Again, either a $Conf$ message will be created later with the same messageChain, or a single node will be added as a child to MI' in the messageChain. If a $Conf$ message was created, this $Conf$ message will eventually arrive at the process which holds the messageTree, where a single child $((Comp, p, 0), ())$ is added to MI'. Alternatively, if a single node was added as a child to MI' in the messageChain, this can result in at most a single child for MI' in the messageTree.

16. There now are two possibilities. Either mc' (the list following MI' in our messageChain) is non-empty, or it is empty.

17. If mc' is non-empty, then we know that none of the messageChains that share the corresponding messageInfo arrived at the process holding the messageTree, because we have assumed earlier that there was no tree in the list of messageTrees that shared a messageInfo with mc'. This in turn implies that the messageTree is not complete. As only complete messageTrees are removed from a messageForest, and because c_j is the last configuration in which a tree with this root existed, $c_j = c_i$.

18. If mc' is empty, then mc is the exact chain that was created when MI' was created. Because there is no way to duplicate trees, this means that node MI' in the messageTree does not have a number of children equal to n, which implies that the messageTree is not complete. As only complete messageTrees are removed from a messageForest, and because c_j is the last configuration in which a tree with this root existed, $c_j = c_i$. $\square$

This Lemma shows that whenever there is a messageChain in the system somewhere, there will be a messageTree in the system that is not yet complete. As all messages are accompanied by messageChains, this will be used to detect the presence of messages in the system.

Lemma 35 *In a terminating computation* $(c_1, c_2, \ldots, c_n, \ldots)$,
$\forall c_n$ *with* c_n *a terminated configuration* : $\forall c_i : i > n : c_i$ *is a terminated configuration.*

Proof.

1. Assume that c_n is a terminated configuration.

2. $\forall c_i : i > n$ there exists a series of computation steps that changes c_n to c_i.

3. A terminated configuration can be falsified in two ways: either a process turns active, or a process creates a message that is not a $Conf$ or Tok message.

4. The only computation step that can change the state of a process from passive to active is the rule from definition RECEIVECOMPLETES. This rule requires the presence of a message.

5. The only two computation steps that can create messages that are not $Conf$ or Tok messages are the rules from definition SEND. These rules require the process that create the message to be active.

6. According to the definition of a terminated configuration, all processes in c_m are passive, and the only existing messages are either $Conf$ or Tok messages.

7. This implies that if c_j is a terminated configuration, and $c_j \to BTTFc_k$, then c_k is not reached by executing rules from definition SEND or definition RECEIVECOMPLETES.

8. This implies that in c_k, all processes are passive and the only messages are $Conf$ or Tok messages, which means that c_k is a terminated configuration.

9. By induction, $\forall c_i : i > m : c_i$ is a terminated configuration.

This Lemma proves that once a terminated configuration has been reached in a computation, all configurations that are reached afterward will also be terminated configurations.

Lemma 36 *In a terminating computation* $(c_1, c_2, \ldots, c_n, \ldots)$,
$\exists c_i \in \mathbb{C} : \neg\exists c \in \mathbb{C} : c_i \rightarrow_{BTTF} c$

Proof.

1. By definition 31, we know that $\exists c_i$ such that c_i is a terminated configuration. By Lemma 35, we know that $\forall c_j : j > i : c_j$ is a terminated configuration. By definition 20, we know that all processes are passive and only $Conf$ and Tok messages are in transit.

2. We will show that there is an upper limit to the number of computation steps that can be performed.

3. The rules from definitions SEND, PASSIVE and RECEIVECOMPLETES can never be executed because they either require the presence of an active process, or the presence of a basic message. The same is true for rules (e) and (f) from definition RECEIVE.

4. The other rules from definition RECEIVE can only be executed at most once per process, as they require *rec* in the process to be False, turn this variable to True, and the only definition that can turn this variable back to False (definition RECEIVECOMPLETES) can not be executed as we've seen before.

5. All rules that are not part of one of these definitions and that can create token messages (rules (c) and (d) from definition CONFIRMATION, rule (d) from definition TOKEN and the rule from definition INITIATION) can only be executed at most once per process, either because they turn *ter* from $False$ to $Pending$, because they change fmc from empty to non-empty, or because they change the messageForest to empty without a way for the messageForest to become non-empty again.

6. Now assume that these rules each are executed exactly once per process, and let $adTok$ be the number of tokens created by the execution of these rules. Furthermore, let $terTok$ be the number of Tok messages in the system in c_i, and let $totTok = terTok + adTok$. As there are no other rules that can be executed that create Tok messages, $totTok$ is the maximum number of times a Tok message can be removed. Because all other rules from definition TOKEN reduce the number of Tok messages, these rules can only be executed a finite number of times.

7. Assume that the rules from definition TOKEN are executed a finite number of times, and let $adCon$ be the number of confirmation messages created this way. Furthermore, let $terCon$ be the number of $Conf$ messages in the system in c_i, and let $totCon = terCon + adCon$. As there are no other rules that can be executed that create $Conf$ messages, $totCon$ is the maximum number of times a $Conf$ message can be removed. Because rules from definition CONFIRMATION all reduce the number of $Conf$ messages, these rules can only be executed a finite number of times.

8. Rules from definition DELIVERY either reduce the number of channels or the number of processes on the path over the least nodes of a message to its destination, so can only be executed a finite number of times for each message.

9. Since all rules can only be executed a finite number of times, after a finite number of computation steps, no rule will be applicable. □

This Lemma shows that each terminating computation is finite, which will be helpful to prove that our algorithm terminates.

5.2 Basic Computation

In this subsection, we show that the basic computation is not altered by running the algorithm. We do this by showing that all transitions in a basic computation are possible in our formal definition of a computation.

Lemma 37 $\forall(\mathbb{P}, \mathbb{E}, \mathcal{A}, \mathcal{M}_{buff}, \mathcal{M}_{trans}, \mathcal{M}_{arr}) \in \mathbb{C}$:
$\forall p, q \in \mathbb{P}$ *such that* $(p, (active, \_, \_, \_, \_, \_, \_, False)) \in \mathcal{A}$,
$\forall m \in \mathbb{M} \backslash \{Conf, Tok\}$:
$\exists(\mathbb{P}, \mathbb{E}, \mathcal{A}', \mathcal{M}'_{buff}, \mathcal{M}'_{trans}, \mathcal{M}'_{arr}) \in \mathbb{C}$:
$(\mathbb{P}, \mathbb{E}, \mathcal{A}, \mathcal{M}_{buff}, \mathcal{M}_{trans}, \mathcal{M}_{arr}) \rightarrow_{BTTF} (\mathbb{P}, \mathbb{E}, \mathcal{A}', \mathcal{M}'_{buff}, \mathcal{M}'_{trans}, \mathcal{M}'_{arr}) \wedge$
$(((m, q), p) \in \mathcal{M}'_{buff} \vee \exists q' \in \mathbb{P} : ((m, mc, q), (p, q')) \in \mathcal{M}'_{trans}$

Proof.

1. Let $(\mathbb{P}, \mathbb{E}, \mathcal{A}, \mathcal{M}_{buff}, \mathcal{M}_{trans}, \mathcal{M}_{arr}) \in \mathbb{C}$, let $p, q \in \mathbb{P}$, with $(p, (active, tqroot, \_, \_, \_, \_, \_, False)) \in \mathcal{A}$, let $m \in \mathbb{M} \backslash \{Conf, Tok\}$.

2. According to definition 15, $tqroot \in \mathbb{B}$, and therefore $tqroot = True \vee tqroot = False$.

3. if $tqroot = True$, then rule (a) from definition SEND is applicable. So according to definition 21, $\exists(\mathbb{P}, \mathbb{E}, \mathcal{A}', \mathcal{M}'_{buff}, \mathcal{M}'_{trans}, \mathcal{M}'_{arr}) \in \mathbb{C}$: $(\mathbb{P}, \mathbb{E}, \mathcal{A}, \mathcal{M}_{buff}, \mathcal{M}_{trans}, \mathcal{M}_{arr}) \rightarrow_{BTTF} (\mathbb{P}, \mathbb{E}, \mathcal{A}', \mathcal{M}'_{buff}, \mathcal{M}'_{trans}, \mathcal{M}'_{arr})$. Furthermore, according to that same definition, $((m, ((id, p, 1), ()), q), (p, q')) \in \mathcal{M}'_{trans}$.

4. if $tqroot = False$, then rule (b) from definition SEND of a computation step is applicable. Again according to definition 21, $\exists(\mathbb{P}, \mathbb{E}, \mathcal{A}', \mathcal{M}'_{buff}, \mathcal{M}'_{trans}, \mathcal{M}'_{arr}) \in \mathbb{C}$: $(\mathbb{P}, \mathbb{E}, \mathcal{A}, \mathcal{M}_{buff}, \mathcal{M}_{trans}, \mathcal{M}_{arr}) \rightarrow_{BTTF} (\mathbb{P}, \mathbb{E}, \mathcal{A}', \mathcal{M}'_{buff}, \mathcal{M}'_{trans}, \mathcal{M}'_{arr})$. Still following that same definition, $((m, q)\ p) \in \mathcal{M}'_{buff}$. □

This Lemma proves that any active process that is not blocking to receive can send any basic message to any process in the system. Such a message will either be located in a channel between two processes on the path over the least nodes between p and q, or it will be located at the sending process, buffered until a batch of messages departs.

Lemma 38 *In a terminating computation* $(c_1, c_2, \ldots, c_n, \ldots)$,
if $c = (\mathbb{P}, \mathbb{E}, \mathcal{A}, \mathcal{M}_{buff}, \mathcal{M}_{trans}, \mathcal{M}_{arr}) \rightarrow_{BTTF} c' = (\mathbb{P}, \mathbb{E}, \mathcal{A}', \mathcal{M}'_{buff}, \mathcal{M}'_{trans}, \mathcal{M}'_{arr}) \wedge$
$(\exists(((m, q), p) \in \mathcal{M}_{buff} \vee \exists((m, mc, q), (p, q')) \in \mathcal{M}_{trans}) \vee \exists((m, mc, q), p)$ *with* $p \neq q \in \mathcal{M}_{arr}) \wedge$
$\neg\exists((m, mc, q), q) \in \mathcal{M}'_{arr}$ *then*
$(\exists(((m, q), p) \in \mathcal{M}'_{buff} \vee \exists((m, mc, q), (p, q')) \in \mathcal{M}'_{trans}) \vee \exists((m, mc, q), p)$ *with* $p \neq q \in \mathcal{M}'_{arr}) \wedge$
$\exists c'' \in \mathbb{C} : c' \rightarrow_{BTTF} c''$.

Proof.

1. Let $c = (\mathbb{P}, \mathbb{E}, \mathcal{A}, \mathcal{M}_{buff}, \mathcal{M}_{trans}, \mathcal{M}_{arr}) \rightarrow_{BTTF} c' = (\mathbb{P}, \mathbb{E}, \mathcal{A}', \mathcal{M}'_{buff}, \mathcal{M}'_{trans}, \mathcal{M}'_{arr})$
and assume that $(\exists(((m, q), p) \in \mathcal{M}_{buff} \vee \exists((m, mc, q), (p, q')) \in \mathcal{M}_{trans}) \vee \exists((m, mc, q), p)$ with $p \neq q \in \mathcal{M}_{arr}) \wedge$
$\neg\exists((m, mc, q), q) \in \mathcal{M}'_{arr}$

2. The computation step that is applied is one of the rules from the definition of a computation step. We will look at each of these rules.

3. The rules from definitions SEND and INITIATION and the rules other then rule (f) from definitions RECEIVE and PASSIVE leave $\mathcal{M}$ unchanged, or only add to $\mathcal{M}$. This means that if $(\exists(((m,\ q),\ p) \in \mathcal{M}_{buff} \vee \exists((m,\ mc,\ q), (p,\ q')) \in \mathcal{M}_{trans}) \vee \exists((m,\ mc,\ q),\ p)$ with $p \neq q \in \mathcal{M}_{arr})$ then $(\exists(((m,\ q),\ p) \in \mathcal{M}'_{buff} \vee \exists((m,\ mc,\ q), (p,\ q')) \in \mathcal{M}'_{trans}) \vee \exists((m,\ mc,\ q),\ p)$ with $p \neq q \in \mathcal{M}'_{arr})$.

4. If rule (f) from definitions RECEIVE or PASSIVE removes a message from $\mathcal{M}$, that message was an element of $\mathcal{M}_{buff}$. This rule will then always add a message $((m, mc, q), (p, q'))$ to $\mathcal{M}'_{trans}$. This means that $\exists((m, mc, q), (p, q')) \in \mathcal{M}'_{trans}$ which in turn means that $(\exists(((m,\ q),\ p) \in \mathcal{M}'_{buff} \vee \exists((m,\ mc,\ q),\ (p,\ q')) \in \mathcal{M}'_{trans}) \vee \exists((m,\ mc,\ q),\ p)$ with $p \neq q \in \mathcal{M}'_{arr})$.

5. The rules from definitions RECEIVECOMPLETES, CONFIRMATION and TOKEN only remove a message from $\mathcal{M}$ if that message is a member of $\mathcal{M}_{arr}$ and was located at the target process q. Because this possibility is explicitly excluded, this also doesn't make a relevant change to $\mathcal{M}$, and it remains true that if $(\exists(((m,\ q),\ p) \in \mathcal{M}_{buff} \vee \exists((m,\ mc,\ q),\ (p,\ q')) \in \mathcal{M}_{trans}) \vee \exists((m,\ mc,\ q),\ p)$ with $p \neq q \in \mathcal{M}_{arr})$ then $(\exists(((m,\ q),\ p) \in \mathcal{M}'_{buff} \vee \exists((m,\ mc,\ q),\ (p,\ q')) \in \mathcal{M}'_{trans}) \vee \exists((m,\ mc,\ q),\ p)$ with $p \neq q \in \mathcal{M}'_{arr})$.

6. Rule a from definition DELIVERY removes a message $((m, mc, q), (p, q'))$ from $\mathcal{M}_{trans}$, but always adds a message $((m, mc, q), q')$ to $\mathcal{M}'_{arr}$. Because we know that $\neg\exists((m, mc, q), q) \in \mathcal{M}'_{arr}$, we can conclude that $\exists((m, mc, q), p)$ with $p \neq q \in \mathcal{M}'_{arr}$ and therefore that $(\exists(((m,\ q),\ p) \in \mathcal{M}'_{buff} \vee \exists((m,\ mc,\ q),\ (p,\ q')) \in \mathcal{M}'_{trans}) \vee \exists((m,\ mc,\ q),\ p)$ with $p \neq q \in \mathcal{M}'_{arr})$.

7. Finally, rule b from definition DELIVERY always adds a message $((m,\ mc,\ q),\ (p,\ q'))$ to $\mathcal{M}'_{trans}$. This means that $\exists((m,\ mc,\ q),\ (p,\ q')) \in \mathcal{M}'_{trans}$ which in turn means that $(\exists(((m,\ q),\ p) \in \mathcal{M}'_{buff} \vee \exists((m,\ mc,\ q),\ (p,\ q')) \in \mathcal{M}'_{trans}) \vee \exists((m,\ mc,\ q),\ p)$ with $p \neq q \in \mathcal{M}'_{arr})$.

8. Regardless of which rule was applied, $(\exists(((m, q), p) \in \mathcal{M}'_{buff} \vee \exists((m, mc, q), (p, q')) \in \mathcal{M}'_{trans}) \vee \exists((m, mc, q), p)$ with $p \neq q \in \mathcal{M}'_{arr})$

9. If $\exists(((m, q), p) \in \mathcal{M}'_{buff}$, then according to Lemma 33, $(p,\ (active,\ False,\ lmc,\ \_,\ \_,\ \_,\ \_,\ False)) \in \mathcal{A}'$ with $lmc \neq ()$. This implies that rule (f) from definition PASSIVE can be applied, which means that $\exists c'' \in \mathbb{C} : c' \rightarrow_{BTTF} c''$.

10. If $\exists((m, mc, q), (p, q')) \in \mathcal{M}'_{trans})$, then rule a from definition DELIVERY can be applied, which means that $\exists c'' \in \mathbb{C} : c' \rightarrow_{BTTF} c''$.

11. If $\exists((m, mc, q), p)$ with $p \neq q \in \mathcal{M}'_{arr}$ then rule b from definition DELIVERY can be applied, which means that $\exists c'' \in \mathbb{C} : c' \rightarrow_{BTTF} c''$.

12. From points 9, 10 and 11, it follows that for $c' = (\mathbb{P}, \mathbb{E}, \mathcal{A}', \mathcal{M}'_{buff}, \mathcal{M}'_{trans}, \mathcal{M}'_{arr})$, $(\exists(((m, q), p) \in \mathcal{M}'_{buff} \vee \exists((m, mc, q), (p, q')) \in \mathcal{M}'_{trans}) \vee \exists((m, mc, q), p)$ with $p \neq q \in \mathcal{M}'_{arr}) \rightarrow \exists c'' \in \mathbb{C} : c' \rightarrow_{BTTF} c''$

13. From points 8 and 12, the result follows. □

This is an auxiliary Lemma for the upcoming Lemma. It shows that as long as a message has not arrived at its destination, a computation step will be applicable.

Lemma 39 *In a terminating computation* $(c_1,\ c_2,\ \ldots,\ c_n,\ \ldots)$,
$(\exists c_i = (\mathbb{P},\ \mathbb{E},\ \mathcal{A},\ \mathcal{M}_{buff},\ \mathcal{M}_{trans},\ \mathcal{M}_{arr}) :$
$\exists(((m,\ q),\ p) \in \mathcal{M}_{buff} \vee \exists((m,\ mc,\ q),\ (p,\ q')) \in \mathcal{M}_{trans}) \rightarrow$

$\exists c_j = (\mathbb{P}, \mathbb{E}, \mathcal{A}', \mathcal{M}'_{buff}, \mathcal{M}'_{trans}, \mathcal{M}'_{arr}) :$
$j > i \wedge ((m, mc, q), q) \in \mathcal{M}'_{arr}$

Proof.

1. this is proven by contradiction. Assume $(\exists c_i = (\mathbb{P}, \mathbb{E}, \mathcal{A}, \mathcal{M}_{buff}, \mathcal{M}_{trans}, \mathcal{M}_{arr}) : \exists(((m,q),p) \in \mathcal{M}_{buff} \vee \exists((m,mc,q),(p,q')) \in \mathcal{M}_{trans})$ and assume $\neg\exists c_j = (\mathbb{P}, \mathbb{E}, \mathcal{A}', \mathcal{M}'_{buff}, \mathcal{M}'_{trans}, \mathcal{M}'_{arr}) : j > i \wedge ((m,mc,q),q) \in \mathcal{M}'_{arr}$.
2. Let $c_k \in (c_1, c_2, \ldots, c_n, \ldots)$ with $k < i$. Then, because c_i exists, $\exists c_{k+1}$. Therefore, according to definition 30, $c_k \rightarrow_{BTTF} c_{k+1}$. This proves that $\forall c_k \in (c_1, c_2, \ldots, c_n, \ldots)$ with $k < i : \exists c \in \mathbb{C} : c_k \rightarrow_{BTTF} c$
3. $\exists(((m,q),p) \in \mathcal{M}_{buff} \vee \exists((m,mc,q),(p,q')) \in \mathcal{M}_{trans})$
4. if $\exists(((m,q),p) \in \mathcal{M}_{buff}$ then according to Lemma 33,
$(p, (active, False, lmc, \_, \_, \_, \_, False)) \in \mathcal{A}$ with $lmc \neq ()$. This implies that rule (f) from definition PASSIVE can be applied, which means that $\exists c \in \mathbb{C} : c_i \rightarrow_{BTTF} c$.
5. if $\exists((m,mc,q),(p,q')) \in \mathcal{M}_{trans}$ then rule (a) from definition DELIVERY can be applied, which means that $\exists c \in \mathbb{C} : c_i \rightarrow_{BTTF} c$.
6. From points 3, 4 and 5, it follows that $\exists c \in \mathbb{C} : c_i \rightarrow_{BTTF} c$
7. From point 1, it follows that $c_i = (\mathbb{P}, \mathbb{E}, \mathcal{A}, \mathcal{M}_{buff}, \mathcal{M}_{trans}, \mathcal{M}_{arr})$ with $(\exists(((m,q),p) \in \mathcal{M}_{buff} \vee \exists((m,mc,q),(p,q')) \in \mathcal{M}_{trans}) \vee \exists((m,mc,q),p)$ with $p \neq q \in \mathcal{M}_{arr})$ (by introducing a third term in the disjunction).
8. From point 1, it follows that $\forall c_j = (\mathbb{P}, \mathbb{E}, \mathcal{A}'', \mathcal{M}''_{buff}, \mathcal{M}''_{trans}, \mathcal{M}''_{arr}) \in (c_1, c_2, \ldots, c_n, \ldots)$ with $j > i : \neg\exists((m,mc,q),q) \in \mathcal{M}''_{arr}$
9. By induction, it follows from points 7, 8 and Lemma 38 that $\forall c_j \in (c_1, c_2, \ldots, c_n, \ldots)$ with $j > i : \exists c \in \mathbb{C} : c_j \rightarrow_{BTTF} c$
10. From points 2, 6 and 9, it follows that $\forall c_j \in (c_1, c_2, \ldots, c_n, \ldots) : \exists c \in \mathbb{C} : c_j \rightarrow_{BTTF} c$
11. According to Lemma 36, in a terminating computation, $\exists c_j \in \mathbb{C} : \neg\exists c \in \mathbb{C} : c_j \rightarrow_{BTTF} c$
12. These last two lines form a contradiction. □

This Lemma shows that any message in the system will eventually arrive at its destination.

Lemma 40 $\forall(\mathbb{P}, \mathbb{E}, \mathcal{A}, \mathcal{M}_{buff}, \mathcal{M}_{trans}, \mathcal{M}_{arr}) \in \mathbb{C} :$
$\forall p \in \mathbb{P}$ *such that* $(p, (active, \_, \_, \_, \_, \_, \_, False)) \in \mathcal{A} :$
$\exists(\mathbb{P}, \mathbb{E}, \mathcal{A}', \mathcal{M}'_{buff}, \mathcal{M}'_{trans}, \mathcal{M}'_{arr}) \in \mathbb{C} :$
$(\mathbb{P}, \mathbb{E}, \mathcal{A}, \mathcal{M}_{buff}, \mathcal{M}_{trans}, \mathcal{M}_{arr}) \rightarrow_{BTTF} (\mathbb{P}, \mathbb{E}, \mathcal{A}', \mathcal{M}'_{buff}, \mathcal{M}'_{trans}, \mathcal{M}'_{arr}) \wedge$
$(p, (passive, \_, \_, \_, \_, \_, \_, False)) \in \mathcal{A}'$

Proof.

1. Let $c = (\mathbb{P}, \mathbb{E}, \mathcal{A}, \mathcal{M}_{buff}, \mathcal{M}_{trans}, \mathcal{M}_{arr}) \in \mathbb{C}$, and let $p \in \mathbb{P}$ such that
$(p, (active, tqroot, lmc, f, init, fmc, ter, False)) \in \mathcal{A}$
2. According to definition 15, $tqroot \in \mathbb{B}$, and therefore $tqroot = True \vee tqroot = False$. We will look at each possibility in turn, and will show that a rule from definition PASSIVE can be applied.

3. Assume $tqroot = True$.
4. Because f is a set, $f = \emptyset \vee f \neq \emptyset$.
5. Assume $f \neq \emptyset$
6. Under the given assumptions, rule (a) can be applied, and if you do, $(p,\ (passive,\ \_,\ \_,\ \_,\ \_,\ \_,\ \_,\ False)) \in \mathcal{A}'$.
7. Drop the assumption that $f \neq \emptyset$ and assume $f = \emptyset$
8. $fmc = () \vee fmc \neq ()$
9. Assume $fmc = ()$
10. Because $init \in \mathbb{B}$, $init = True \vee init = False$.
11. Assume $init = False$.
12. Under the given assumptions, rule (b) can be applied, and if you do, $(p,\ (passive,\ \_,\ \_,\ \_,\ \_,\ \_,\ \_,\ False)) \in \mathcal{A}'$.
13. Drop the assumption that $init = False$ and assume $init = True$
14. Under the given assumptions, rule (c) can be applied, and if you do, $(p,\ (passive,\ \_,\ \_,\ \_,\ \_,\ \_,\ \_,\ False)) \in \mathcal{A}'$.
15. Drop the assumption that $fmc = ()$ and assume $fmc \neq ()$
16. Under the given assumptions, rule (d) can be applied, and if you do, $(p,\ (passive,\ \_,\ \_,\ \_,\ \_,\ \_,\ \_,\ False)) \in \mathcal{A}'$.
17. Drop the assumption that $tqroot = True$ and assume $tqroot = False$.
18. $(\exists((m, q), p) \in \mathcal{M}_{buff}) \vee (\neg\exists((m, q), p) \in \mathcal{M}_{buff})$
19. Assume $\neg\exists((m, q), p) \in \mathcal{M}_{buff}$
20. Under the given assumptions, rule (e) can be applied, and if you do, $(p,\ (passive,\ \_,\ \_,\ \_,\ \_,\ \_,\ \_,\ False)) \in \mathcal{A}'$.
21. Drop the assumption that $\neg\exists((m,\ q),\ p) \in \mathcal{M}_{buff}$ and assume $\exists((m,\ q),\ p) \in \mathcal{M}_{buff}$
22. Under the given assumptions, rule (f) can be applied, and if you do, $(p,\ (passive,\ \_,\ \_,\ \_,\ \_,\ \_,\ \_,\ False)) \in \mathcal{A}'$. $\square$

This Lemma shows that when an active process is not blocking to receive, it can turn passive at any time.

Lemma 41 *In a terminating computation* $(c_1,\ c_2,\ \ldots,\ c_n,\ \ldots)$,
$\forall c_i = (\mathbb{P},\ \mathbb{E},\ \mathcal{A},\ \mathcal{M}_{buff},\ \mathcal{M}_{trans},\ \mathcal{M}_{arr}) \in \mathbb{C}:$
$\forall p \in \mathbb{P}$ *such that* $(p,\ (\_,\ \_,\ \_,\ \_,\ \_,\ \_,\ \_,\ False)) \in \mathcal{A}:$
$((\exists c' = (\mathbb{P},\ \mathbb{E},\ \mathcal{A}',\ \mathcal{M}'_{buff},\ \mathcal{M}'_{trans},\ \mathcal{M}'_{arr}) \in \mathbb{C}:$
$(p,\ (\_,\ \_,\ \_,\ \_,\ \_,\ \_,\ \_,\ True)) \in \mathcal{A}' \wedge c_i \rightarrow_{BTTF} c')$.

Proof.

1. Let $c_i = (\mathbb{P}, \mathbb{E}, \mathcal{A}, \mathcal{M}_{buff}, \mathcal{M}_{trans}, \mathcal{M}_{arr}) \in \mathbb{C}$ and let $p \in \mathbb{P}$ such that $(p, (state, tqroot, lmc, f, init, fmc, ter, False)) \in \mathcal{A}$
2. According to definition 15, $tqroot \in \mathbb{B}$, and therefore $tqroot = True \vee tqroot = False$. We will look at each possibility in turn, and will show that a rule from definition RECEIVE can be applied.
3. Assume $tqroot = True$.
4. Either $f = \emptyset \vee f \neq \emptyset$.
5. Assume $f \neq \emptyset$
6. Under the given assumptions, rule (a) of the definition of a computation step can be applied, which would result in $c' = (\mathbb{P}, \mathbb{E}, \mathcal{A}', \mathcal{M}'_{buff}, \mathcal{M}'_{trans}, \mathcal{M}'_{arr}) \in \mathbb{C} : (p, (\_, \_, \_, \_, \_, \_, \_, True)) \in \mathcal{A}' \wedge c_i \rightarrow_{BTTF} c'$.
7. Drop the assumption that $f \neq \emptyset$ and assume $f = \emptyset$
8. Either $fmc = () \vee fmc \neq ()$.
9. Assume $fmc = ()$.
10. As $init \in \mathbb{B}$, $init = False \vee init = True$.
11. Assume $init = False$.
12. Under the given assumptions, rule (b) of the definition of a computation step can be applied, which would result in $c' = (\mathbb{P}, \mathbb{E}, \mathcal{A}', \mathcal{M}'_{buff}, \mathcal{M}'_{trans}, \mathcal{M}'_{arr}) \in \mathbb{C} : (p, (\_, \_, \_, \_, \_, \_, \_, True)) \in \mathcal{A}' \wedge c_i \rightarrow_{BTTF} c'$.
13. Drop the assumption that $init = False$ and assume $init = True$.
14. Under the given assumptions, rule (c) of the definition of a computation step can be applied, which would result in $c' = (\mathbb{P}, \mathbb{E}, \mathcal{A}', \mathcal{M}'_{buff}, \mathcal{M}'_{trans}, \mathcal{M}'_{arr}) \in \mathbb{C} : (p, (\_, \_, \_, \_, \_, \_, \_, True)) \in \mathcal{A}' \wedge c_i \rightarrow_{BTTF} c'$.
15. Drop the assumption that $fmc = ()$ and assume $fmc \neq ()$.
16. Under the given assumptions, rule (d) of the definition of a computation step can be applied, which would result in $c' = (\mathbb{P}, \mathbb{E}, \mathcal{A}', \mathcal{M}'_{buff}, \mathcal{M}'_{trans}, \mathcal{M}'_{arr}) \in \mathbb{C} : (p, (\_, \_, \_, \_, \_, \_, \_, True)) \in \mathcal{A}' \wedge c_i \rightarrow_{BTTF} c'$.
17. Drop the assumption that $tqroot = True$ and assume $tqroot = False$.
18. As $state \in \mathbb{S}$, either $state = active \vee state = passive$
19. Assume $state = active$.
20. Either $\neg\exists((m, dest), p) \in \mathcal{M}_{buff} \vee \exists((m, dest), p) \in \mathcal{M}_{buff}$.
21. Assume $\neg\exists((m, dest), p) \in \mathcal{M}_{buff}$.
22. Under the given assumptions, rule (e) of the definition of a computation step can be applied, which would result in $c' = (\mathbb{P}, \mathbb{E}, \mathcal{A}', \mathcal{M}'_{buff}, \mathcal{M}'_{trans}, \mathcal{M}'_{arr}) \in \mathbb{C} : (p, (\_, \_, \_, \_, \_, \_, \_, True)) \in \mathcal{A}' \wedge c_i \rightarrow_{BTTF} c'$.
23. Drop the assumption $\neg\exists((m, dest), p) \in \mathcal{M}_{buff}$ and assume $\exists((m, dest), p) \in \mathcal{M}_{buff}$.

24. Under the given assumptions, rule (f) of the definition of a computation step can be applied, which would result in $c' = (\mathbb{P}, \mathbb{E}, \mathcal{A}', \mathcal{M}'_{buff}, \mathcal{M}'_{trans}, \mathcal{M}'_{arr}) \in \mathbb{C} : (p, (\_, \_, \_, \_, \_, \_, \_, True)) \in \mathcal{A}' \wedge c_i \rightarrow_{BTTF} c'$.

25. Drop the assumption that $state = active$ and assume $state = passive$.

26. Under the given assumptions, rule (g) of the definition of a computation step can be applied, which would result in $c' = (\mathbb{P}, \mathbb{E}, \mathcal{A}', \mathcal{M}'_{buff}, \mathcal{M}'_{trans}, \mathcal{M}'_{arr}) \in \mathbb{C} : (p, (\_, \_, \_, \_, \_, \_, \_, True)) \in \mathcal{A}' \wedge c_i \rightarrow_{BTTF} c'$.

27. From points 6, 12, 14, 16, 22, 24 and 26, it follows that $\exists c' = (\mathbb{P}, \mathbb{E}, \mathcal{A}', \mathcal{M}'_{buff}, \mathcal{M}'_{trans}, \mathcal{M}'_{arr}) \in \mathbb{C} : (p, (\_, \_, \_, \_, \_, \_, \_, True)) \in \mathcal{A}' \wedge c_i \rightarrow_{BTTF} c'$. $\square$

This Lemma shows how any process that is not blocking to receive can perform a receive.

Lemma 42 *In a terminating computation* $(c_1, c_2, \ldots, c_n, \ldots)$,
$(\exists c_i = (\mathbb{P}, \mathbb{E}, \mathcal{A}, \mathcal{M}_{buff}, \mathcal{M}_{trans}, \mathcal{M}_{arr}) \in \mathbb{C}$:
$\exists p \in \mathbb{P}$ *such that* $(p, (\_, \_, \_, \_, \_, \_, \_, True)) \in \mathcal{A} \wedge$
$\exists c_j = (\mathbb{P}, \mathbb{E}, \mathcal{A}', \mathcal{M}'_{buff}, \mathcal{M}'_{trans}, \mathcal{M}'_{arr}) \in \mathbb{C} : j \geq i \wedge$
$(\exists((m, mc, p), (p', q) \in \mathcal{M}'_{trans}) \vee \exists((m, p), q) \in \mathcal{M}'_{buff}) \wedge m \neq Conf \wedge m \neq Tok) \rightarrow$
$\exists c_l = (\mathbb{P}, \mathbb{E}, \mathcal{A}'', \mathcal{M}''_{buff}, \mathcal{M}''_{trans}, \mathcal{M}''_{arr}) \in \mathbb{C}$:
$l > i \wedge (p, (active, \_, \_, \_, \_, \_, \_, False)) \in \mathcal{A}''))$

Proof.

1. Assume $\exists c_i = (\mathbb{P}, \mathbb{E}, \mathcal{A}, \mathcal{M}_{buff}, \mathcal{M}_{trans}, \mathcal{M}_{arr}) \in \mathbb{C} : \exists p \in \mathbb{P}$ such that $(p, (\_, \_, \_, \_, \_, \_, \_, True)) \in \mathcal{A}$.

2. Assume $\exists c_j = (\mathbb{P}, \mathbb{E}, \mathcal{A}', \mathcal{M}'_{buff}, \mathcal{M}'_{trans}, \mathcal{M}'_{arr}) \in \mathbb{C} : j \geq i \wedge (\exists((m, mc, p), (p', q) \in \mathcal{M}'_{trans}) \vee \exists((m, p), q) \in \mathcal{M}'_{buff}) \wedge m \neq Conf \wedge m \neq Tok$.

3. As we have point 2, we can apply Lemma 39, which tells us $\exists c_k = (\mathbb{P}, \mathbb{E}, \mathcal{A}''', \mathcal{M}'''_{buff}, \mathcal{M}'''_{trans}, \mathcal{M}'''_{arr}) : k > j \wedge ((m, mc, p), p) \in \mathcal{M}'''_{arr}$

4. Since $((m, mc, p), p) \in \mathcal{M}'''_{arr}$, c_k is not a terminated configuration.

5. According to Lemma 35, $\exists c_n : \forall c_m$ with $m > n$ c_m is a terminated configuration.

6. Form these last two points, it follows that $n > k$. It also follows that there exists a series of computation steps that transforms c_k into c_n.

7. As c_n is a terminated configuration, one of these computation steps must remove $((m, mc, p), p)$ from the set of messages.

8. The only computation step that can do that is the computation step from definition RECEIVECOMPLETES. This implies that one of the computation steps between c_k and c_n is the computation step from that definition applied to process p.

9. As a result of applying that computation step, $(active, False, mc, f, init, fmc, ter\ False)$ will be the set of attributes of p.

10. Therefore, $\exists c_l = (\mathbb{P}, \mathbb{E}, \mathcal{A}'', \mathcal{M}''_{buff}, \mathcal{M}''_{trans}, \mathcal{M}''_{arr}) \in \mathbb{C} : l > i \wedge (p, (active, \_, \_, \_, \_, \_, \_, False)) \in \mathcal{A}''))$. $\square$

This Lemma shows that if a process is blocking to receive, and a message for that process exists somewhere in the system, the receive call will complete, and the process will end up active.

Theorem 43 *A terminating computation is a correct implementation of a basic computation.*

Proof. In a terminating computation:

1. An active process can send messages as can be seen in Lemma 37.
2. An active process can receive messages, as can be seen in Lemma 41, and such a receive will complete provided a message for that process exists, according to Lemma 42.
3. An active process can change it state to passive at any time, as can be seen in Lemma 40.
4. A passive process can receive a message according to Lemma 41, which will complete and change its state to active provided a message for that process exists, according to Lemma 42.
5. Messages will arrive at their destination in finite time, according to Lemma 39.

Therefore, all requirements of a basic computation have been met. □

5.3 Safety

In this subsection, we prove the safety of our algorithm. In other words, we show that when our algorithm indicates termination, the basic computation has actually terminated. This is done in two steps.

First, we show that when all messageForests are empty and all processes have *tqroot* = $False$, the computation has terminated (Lemma 47). As a terminated configuration requires both that there are no messages in transit and that all processes are passive, we do this by showing that both of those statements follow individually from the fact that all messageForests are empty and that all processes have *tqroot* = $False$ (Lemmas 44 and 46).

The second step is to show that when our algorithm indicates termination, all messageForests of all processes are empty and all processes have *tqroot* = $False$ (Lemma 53). In order to show this, we show that when our algorithm indicates termination, all processes have forwarded tokens to their neighbors (Lemma 50). Furthermore, we show that with one exception a process only forwards tokens when its messageForest is empty and *tqroot* = $False$ (Lemma 52). Finally (though due to dependencies not last), we show that when a process has an empty messageForest and *tqroot* = $False$, this will remain the case (Lemma 51).

All other Lemmas in this section are used to support these main points.

Lemma 44 *In a terminating computation* $(c_1, c_2, \ldots, c_n, \ldots)$,
$\forall c_i = (\mathbb{P}, \mathbb{E}, \mathcal{A}, \mathcal{M}_{buff}, \mathcal{M}_{trans}, \mathcal{M}_{arr}) \in \mathbb{C}$:
$(\forall p \in \mathbb{P} : ((p, (\_, False, \_, \emptyset, False, \_, \_, \_)) \vee$
$(p, (\_, \_, \_, \_, True, \_, Pending, \_)) \vee$
$(p, (\_, \_, \_, \_, True, \_, True, \_))) \in \mathcal{A}) \rightarrow$
$(\forall m \in \mathcal{M} : m = Tok \vee m = Conf)$

Proof.

1. This is proven by contradiction.
2. Let $c_i = (\mathbb{P}, \mathbb{E}, \mathcal{A}, \mathcal{M}_{buff}, \mathcal{M}_{trans}, \mathcal{M}_{arr})$. Assume $\forall p \in \mathbb{P}$: $((p, (\_, False, \_, \emptyset, False, \_, \_, \_)) \vee (p, (\_, \_, \_, \_, True, \_, Pending, \_)) \vee (p, (\_, \_, \_, \_, True, \_, True, \_))) \in \mathcal{A}$ and assume $\exists m \in \mathcal{M} : m \neq Tok \wedge m \neq Conf$

3. These statements imply $\exists((m,p),q) \in \mathcal{M}_{buff} \vee \exists((m,mc,p),q) \in \mathcal{M}_{arr} \vee \exists((m,mc,p),(q,q')) \in \mathcal{M}_{trans}$ with $m \neq Tok$ and $m \neq Conf$, due to the way $\mathcal{M}$ is defined.

4. If $\exists((m,p),q) \in \mathcal{M}_{buff}$, then according to Lemma 33, $(q,\ (active,\ False,\ lmc,\ \_,\ \_,\ \_,\ \_,\ False)) \in \mathcal{A}$ with $lmc \neq ()$. In that case, Lemma 34 can be applied. If any of the other two rules of the disjunction are the case, Lemma 34 can be applied as well.

5. Therefore, we apply Lemma 34. According to that Lemma, $\exists(q',\ (\_,\ \_,\ \_,\ mf,\ \_,\ \_,\ \_,\ \_)) \in \mathcal{A}$: $\exists tree \in mf : tree$ is not a complete messageTree $\wedge mc = (MI,\ mc') \wedge tree = (MI, (T_1,\ \ldots,\ T_{n'}))$.

6. Due to our assumptions, $((q',\ (\_,\ False,\ \_,\ \emptyset,\ False,\ \_,\ \_,\ \_)) \vee (q',\ (\_,\ \_,\ \_,\ \_,\ True,\ \_,\ Pending,\ \_)) \vee (q',\ (\_,\ \_,\ \_,\ \_,\ True,\ \_,\ True,\ \_))) \in \mathcal{A}$. We will show that each of these leads to a contradiction.

7. $(q', (\_, False, \_, \emptyset, False, \_, \_, \_))$ directly contradicts with our conclusion that an incomplete messageTree exists in the messageForest of q'.

8. If $(q', (\_, \_, \_, \_, True, \_, Pending, \_))$, then rule (c) from definitions RECEIVE, PASSIVE or CONFIRMATION or the rule from definition INITIATION must have been executed, since these are the only rules in the definition of a computation step that allow *ter* to change from *False* to *Pending*. We will investigate further.

   (a) After executing rule (c) from definitions RECEIVE, PASSIVE or CONFIRMATION or after executing the rule from definition INITIATION, the only messageTree in mf is a tree starting with an identifier that accompanied a number of Tok messages. Furthermore, after executing these rules *tqroot* becomes false, which means that no rules can be executed to add other trees to mf.

   (b) As *mc* starts with the same messageInfo, *mc* must have been the result of the execution of a number of computation steps on a messageChain accompanying a Tok message.

   (c) However, such computation steps can only generate Tok or $Conf$ messages.

   (d) This is in direct contradiction with our conclusion that $m \neq Tok$ and $m \neq Conf$.

9. Finally, if $(q', (\_, \_, \_, \_, True, \_, True, \_)) \in \mathcal{A}$, rule (f) of definition CONFIRMATION must have been executed. However, after this computation step, $mf = \emptyset \wedge tqroot = False$. This means that no computation step can be executed to change the fact that $mf = \emptyset$ (because all such steps either require $init = False$ or $tqroot = True$). This directly contradicts with our conclusion that an incomplete messageTree exists in the messageForest of q'. $\square$

This Lemma shows that if the initiator has started its part of the algorithm and for all other processes *tqroot* is false and their messageForest is empty, there are no basic messages in transit.

Lemma 45 *In a terminating computation* $(c_1,\ c_2,\ \ldots,\ c_n,\ \ldots)$,
$\forall c_i = (\mathbb{P},\ \mathbb{E},\ \mathcal{A},\ \mathcal{M}_{buff},\ \mathcal{M}_{trans},\ \mathcal{M}_{arr}) \in \mathbb{C}$:
$((\forall p \in \mathbb{P} : (p,\ (\_,\ False,\ \_,\ \emptyset,\ \_,\ \_,\ \_,\ \_)) \vee$
$(p,\ (\_,\ \_,\ \_,\ \_,\ True,\ \_,\ Pending,\ \_)) \vee$
$(p,\ (\_,\ \_,\ \_,\ \_,\ True,\ \_,\ True,\ \_))) \in \mathcal{A}) \rightarrow$
$(\forall p \in \mathbb{P} : (p,\ (passive,\ \_,\ \_,\ \_,\ \_,\ \_,\ \_,\ \_)) \in \mathcal{A} \vee (p,\ (\_,\ \_,\ \_,\ \_,\ \_,\ \_,\ \_,\ True)) \in \mathcal{A})$

Proof.

1. Let $c_i = (\mathbb{P},\ \mathbb{E},\ \mathcal{A},\ \mathcal{M}_{buff},\ \mathcal{M}_{trans},\ \mathcal{M}_{arr})$. Assume $\forall p \in \mathbb{P} : ((p,\ (\_,\ False,\ \_,\ \emptyset,\ \_,\ \_,\ \_,\ \_)) \vee (p,\ (\_,\ \_,\ \_,\ \_,\ True,\ \_,\ Pending,\ \_)) \vee (p,\ (\_,\ \_,\ \_,\ \_,\ True,\ \_,\ True,\ \_))) \in \mathcal{A}$. Let $q \in \mathbb{P}$.

2. $\mathbb{S} = \{active, passive\}$. This implies $(q, (passive, \_, \_, \_, \_, \_, \_, \_)) \in \mathcal{A} \vee (q, (active, \_, \_, \_, \_, \_, \_, \_)) \in \mathcal{A}$.

3. If we assume $(q,\ (passive,\ \_,\ \_,\ \_,\ \_,\ \_,\ \_,\ \_)) \in \mathcal{A}$, then $(q,\ (passive,\ \_,\ \_,\ \_,\ \_,\ \_,\ \_,\ \_)) \in \mathcal{A} \vee (q,\ (\_,\ \_,\ \_,\ \_,\ \_,\ \_,\ \_,\ True)) \in \mathcal{A})$, and we are done.

4. Therefore, assume $(q,\ (active,\ \_,\ \_,\ \_,\ \_,\ \_,\ \_,\ \_)) \in \mathcal{A}$. We will prove by contradiction that $(q,\ (active,\ \_,\ \_,\ \_,\ \_,\ \_,\ \_,\ True)) \in \mathcal{A}$.

5. Assume $(q, (active, \_, \_, \_, \_, \_, \_, False)) \in \mathcal{A}$. Now, there are two possibilities. Either the rule from definition RECEIVECOMPLETES was executed on q during the computation, or it wasn't.

6. If the rule from definition RECEIVECOMPLETES was executed on q, then let us consider the last time this happened. After the rule was executed, $(q,\ (active,\ \_,\ lmc,\ \_,\ \_,\ \_,\ \_,\ False)) \in \mathcal{A}$, and because of Lemma 32, $lmc \neq ()$. After this, neither rule (e) nor rule (f) from definition RECEIVE were executed, because these would turn *rec* to $True$. As these are the only way to turn *lmc* empty, this implies that $lmc \neq ()$.

7. However, according to Lemma 34, $\exists(q', (\_, \_, \_, mf, \_, \_, \_, \_)) \in \mathcal{A} : \exists tree \in mf : tree$ is not a complete messageTree $\wedge lmc = (MI,\ mc') \wedge tree = (MI, (T_1, \ldots, T_{n'}))$. Our assumptions imply that this messageTree is located in a process were $init = True$ and $ter = Pending \vee ter = True$. However, all computation steps that create a messageTree in such a process create either Tok or $Conf$ messages, which can only result in further Tok or $Conf$ messages. This contradicts our assumption that the rule from definition RECEIVECOMPLETES was executed, for that rule specifically excludes Tok and $Conf$ messages.

8. Now, let us assume that the rule from definition RECEIVECOMPLETES was never executed. Then q must have been active in c_1, as there is no other way for a process to turn active. However, in that case, *tqroot* was $True$, and all possible computation steps that turn *tqroot* to $False$ can't have been executed because they turn *rec* to True.

9. It follows that $(q, (active, \_, \_, \_, \_, \_, \_, True)) \in \mathcal{A}$. $\square$

This is an auxiliary statement for the following Lemma.

Lemma 46 *In a terminating computation* $(c_1,\ c_2,\ \ldots,\ c_n,\ \ldots)$,
$\forall c_i = (\mathbb{P},\ \mathbb{E},\ \mathcal{A},\ \mathcal{M}_{buff},\ \mathcal{M}_{trans},\ \mathcal{M}_{arr}) \in \mathbb{C}$:
$((\forall p \in \mathbb{P} : (p,\ (\_,\ False,\ \_,\ \emptyset,\ \_,\ \_,\ \_,\ \_)) \vee$
$(p,\ (\_,\ \_,\ \_,\ \_,\ True,\ \_,\ Pending,\ \_)) \vee$
$(p,\ (\_,\ \_,\ \_,\ \_,\ True,\ \_,\ True,\ \_))) \in \mathcal{A}) \rightarrow$
$(\forall p \in \mathbb{P} : (p,\ (passive,\ \_,\ \_,\ \_,\ \_,\ \_,\ \_,\ \_)) \in \mathcal{A})$

Proof.

1. According to Lemma 45, $(\forall p \in \mathbb{P} : (p, (passive, \_, \_, \_, \_, \_, \_, \_)) \in \mathcal{A} \vee (p, (\_, \_, \_, \_, \_, \_, \_, True)) \in \mathcal{A})$

2. According to Lemma 44, $\forall m \in \mathcal{M} : m = Tok \vee m = Conf$.

3. We will show that $(p, (active, \_, \_, \_, \_, \_, \_, True)) \in \mathcal{A}$ leads to a contradiction. Assume $\exists p \in \mathbb{P}$: $(p, (active, \_, \_, \_, \_, \_, \_, True)) \in \mathcal{A}$.

4. Note that $\forall j < i : c_j$ is not a terminated configuration, because according to Lemma 35, all configurations reached after that are terminated configurations as well.

5. Because all processes are either passive or waiting to receive, no process can create a message that is not a Tok or $Conf$ message. Furthermore, no such messages are currently present.

6. This implies that the rule from definition RECEIVECOMPLETES can never be executed, which in turn implies that p will never turn passive. But then no terminated configuration will ever be reached, which contradicts the fact that this is a terminating configuration.

7. Therefore, $\forall p \in \mathbb{P} : (p, (passive, \_, \_, \_, \_, \_, \_, \_)) \in \mathcal{A})$ $\Box$

This Lemma proves that once all messageForests have turned empty and no processes can create new messageForests (excluding the initiator who may have a non-empty messageForest to track the progress of Tok messages), all processes must be passive.

Lemma 47 *In a terminating computation* $(c_1, c_2, \ldots, c_n, \ldots)$,
$\forall c_i = (\mathbb{P}, \mathbb{E}, \mathcal{A}, \mathcal{M}_{buff}, \mathcal{M}_{trans}, \mathcal{M}_{arr}) \in \mathbb{C} :$
$(\forall p \in \mathbb{P} : ((p, (\_, False, \_, \emptyset, \_, \_, \_, \_)) \vee$
$(p, (\_, \_, \_, \_, True, \_, Pending, \_)) \vee$
$(p, (\_, \_, \_, \_, True, \_, True, \_))) \in \mathcal{A}) \rightarrow$
c_i *is a terminated configuration*

Proof

1. This follows directly from Lemmas 44 and 46, and from the definition of a terminated configuration. $\Box$

This Lemma states that the fact that all messageForests are empty (excluding the initiator, who may track the progress of Tok messages) and the fact that no processes can create new messageTrees together can be used as an indicator that the computation has terminated.

Lemma 48 *In a terminating computation* $(c_1, c_2, \ldots, c_n, \ldots)$,
$\forall p \in \mathbb{P} : (\exists c_i = (\mathbb{P}, \mathbb{E}, \mathcal{A}, \mathcal{M}_{buff}, \mathcal{M}_{trans}, \mathcal{M}_{arr}) \in \mathbb{C} :$
$\exists p' \in \mathbb{P} : (p', (\_, \_, \_, \_, \_, \_, True, \_)) \in \mathcal{A} \wedge$
$\exists c_j = (\mathbb{P}, \mathbb{E}, \mathcal{A}', \mathcal{M}'_{buff}, \mathcal{M}'_{trans}, \mathcal{M}'_{arr}) \in \mathbb{C} :$
$((Tok, \_, p), p) \in \mathcal{M}'_{arr}) \rightarrow$
$(\exists c_k = (\mathbb{P}, \mathbb{E}, \mathcal{A}'', \mathcal{M}''_{buff}, \mathcal{M}''_{trans}, \mathcal{M}''_{arr}) \in \mathbb{C} :$
$\forall (p, q) \in \mathbb{E} : ((Tok, \_, q), (p, q)) \in \mathcal{M}''_{trans})$

Proof.

1. Since this is a terminating computation, the token message in $\mathcal{M}'_{arr}$ must be removed at some point in time. Only the rules of definition TOKEN can accomplish this. We will take a look at each of these rules.

2. If rule (c) of definition TOKEN was performed, then $(p, (\_, \_, \_, \_, True, \_, Pending, \_)) \in \mathcal{A}'$. As all processes start with $init = False$, this implies that either rule (c) of definitions RECEIVE, PASSIVE or CONFIRMATION, or the rule from definition INITIATION were executed some time earlier, from which the result follows.

3. If rule (b), rule (a) or rule (e) from definition TOKEN were executed, then after the execution $(p, (\_, \_, \_, \_, False, fmc, \_, \_))$ is in the set of attributes, with $fmc \neq ()$. This has a number of consequences:

   (a) As p' has $ter = True$, at some point this process must have executed rule (f) from definition CONFIRMATION. This implies that all messageChains resulting from token messages, including the chain belonging to fmc, must have been changed into $Conf$ messages at some point in the computation.

(b) In order to change into a confirmation message, a Tok message must arrive at a process that already has a messageChain stored in its fmc.

(c) The only way in which this can happen to a messageChain stored in fmc, is if rule (d) of definition TOKEN was performed upon arrival of the Tok, or if rule d of definitions RECEIVE, PASSIVE, or CONFIRMATION was performed at some point during the computation. In all those cases, the result follows.

4. If rule (d) of definition TOKEN was performed, the result follows immediately. □

This Lemma shows that a process that has been reached by a token message will have forwarded a token message to all its neighbors at some point during the computation, provided that the initiator indicates termination at some point during the computation.

Lemma 49 *In a terminating computation* $(c_1, c_2, \ldots, c_n, \ldots)$,
$(\exists c_i = (\mathbb{P}, \mathbb{E}, \mathcal{A}, \mathcal{M}_{buff}, \mathcal{M}_{trans}, \mathcal{M}_{arr}) \in \mathbb{C} :$
$\exists p' \in \mathbb{P} : (p', (\_, \_, \_, \_, \_, \_, True, \_)) \in \mathcal{A}) \rightarrow$
$(\forall p \in \mathbb{P} \setminus \{(p', (\_, \_, \_, \_, \_, \_, \_, \_))\} :$
$\exists c_j = (\mathbb{P}, \mathbb{E}, \mathcal{A}', \mathcal{M}'_{buff}, \mathcal{M}'_{trans}, \mathcal{M}'_{arr}) \in \mathbb{C} :$
$((Tok, \_, p), p) \in \mathcal{M}'_{arr}$

Proof.

1. This will be proven by induction on the length of the path over the least nodes from p' (which exists according to Lemma 6.

2. First, let us assume that a path of length 1 exists from p' to p.

3. In c_1, $(p', (\_, \_, \_, \_, \_, \_, False, \_))$ is in the list of attributes. At some point, ter of p' must have turned to $Pending$ (because it is $True$ in c_i). This implies that rule (c) of definitions RECEIVE, PASSIVE or CONFIRMATION, or the rule from definition INITIATION was executed at some point in time, at which moment a Tok message was sent on the channel between p' and p.

4. This message must have arrived at p, because otherwise there is no way the messageTree for the Tok messages in p' could have become a complete messageTree (which is necessary to execute rule CONFIRMATION(f) in order to turn ter to $True$).

5. It follows that $\exists c_j = (\mathbb{P}, \mathbb{E}, \mathcal{A}', \mathcal{M}'_{buff}, \mathcal{M}'_{trans}, \mathcal{M}'_{arr}) \in \mathbb{C} : ((Tok, \_, p), p) \in \mathcal{M}'_{arr}$.

6. Now, assume that the length of the path over the least nodes from p' to p is n, and assume that $\forall p'' \in \mathbb{P}$ for which the length of the path over the least nodes from p' to p'' is smaller then n and which are not p', $\exists c_j = (\mathbb{P}, \mathbb{E}, \mathcal{A}'', \mathcal{M}''_{buff}, \mathcal{M}''_{trans}, \mathcal{M}''_{arr}) \in \mathbb{C} : ((Tok, \_, p''), p'') \in \mathcal{M}''_{arr}$.

7. In particular, let us consider the process immediately before p on the path over the least nodes from p' to p, and let us call this process p'''.

8. By assumption, $\exists c_j = (\mathbb{P}, \mathbb{E}, \mathcal{A}', \mathcal{M}'_{buff}, \mathcal{M}'_{trans}, \mathcal{M}'_{arr}) \in \mathbb{C} : ((Tok, \_, p'''), p''') \in \mathcal{M}'_{arr}$.

9. Then, by Lemma 48, this process sent a Tok message to all its neighbors, including p.

10. This message must have arrived at p, because otherwise there is no way the messageTree for the Tok messages in p' could have become a complete messageTree (which is necessary to execute rule CONFIRMATION(f)).

11. It follows that $\exists c_{j'} = (\mathbb{P}, \mathbb{E}, \mathcal{A}', \mathcal{M}'_{buff}, \mathcal{M}'_{trans}, \mathcal{M}'_{arr}) \in \mathbb{C} : ((Tok, \_, p), p) \in \mathcal{M}'_{arr}$. □

This Lemma shows that if the initiator indicates termination, all processes have received a Tok message at least once.

Lemma 50 *In a terminating computation* $(c_1,\, c_2,\, \ldots,\, c_n,\, \ldots)$,
$(\exists c_i = (\mathbb{P},\, \mathbb{E},\, \mathcal{A},\, \mathcal{M}_{buff},\, \mathcal{M}_{trans},\, \mathcal{M}_{arr}) \in \mathbb{C}:$
$\exists p' \in \mathbb{P} : (p',\, (\_,\, \_,\, \_,\, \_,\, \_,\, \_,\, True,\, \_)) \in \mathcal{A}) \rightarrow$
$(\forall p \in \mathbb{P} : \exists c_j = (\mathbb{P},\, \mathbb{E},\, \mathcal{A}',\, \mathcal{M}'_{buff},\, \mathcal{M}'_{trans},\, \mathcal{M}'_{arr}) \in \mathbb{C}:$
$\forall (p,\, q) \in \mathbb{E} : ((Tok,\, \_,\, q),\, (p,\, q)) \in \mathcal{M}'_{trans})$

Proof.

1. In c_1, $(p', (\_, \_, \_, \_, \_, \_, False, \_))$ is in the list of attributes. At some point, ter of p' must have turned to $Pending$ (because it is $True$ in c_i). This implies that rule c of definitions RECEIVE, PASSIVE or CONFIRMATION, or the rule from definition INITIATION was executed at some point in time, at which moment a Tok message was sent on the channel between p' and p. It follows that the process where $init = True$ must have forwarded the token to all its neighbors.

2. For all other processes, they received a Tok message according to Lemma 49, which implies that they forwarded the message according to Lemma 48. □

This Lemma shows that in a terminating computation, if the initiator indicates termination, all processes have forwarded Tok messages to their neighbors.

Lemma 51 *In a computation* $(c_1,\, c_2,\, \ldots,\, c_n,\, \ldots)$,
$(\exists c_i = (\mathbb{P},\, \mathbb{E},\, \mathcal{A},\, \mathcal{M}_{buff},\, \mathcal{M}_{trans},\, \mathcal{M}_{arr}) \in \mathbb{C}:$
$\exists p \in \mathbb{P} : (p,\, (\_,\, False,\, \_,\, \emptyset,\, \_,\, \_,\, \_,\, \_)) \vee (p,\, (\_,\, \_,\, \_,\, \_,\, True,\, \_,\, Pending,\, \_)) \in \mathcal{A}) \rightarrow$
$\forall j > i : c_j = (\mathbb{P},\, \mathbb{E},\, \mathcal{A}',\, \mathcal{M}'_{buff},\, \mathcal{M}'_{trans},\, \mathcal{M}'_{arr})$
with $(p, (\_, False, \_, \emptyset, \_, \_, \_, \_)) \vee (p, (\_, \_, \_, \_, True, \_, Pending, \_)) \vee (p, (\_, \_, \_, \_, True, \_, True, \_)) \in \mathcal{A}')$

Proof.

1. Let us assume that $\exists c_i = (\mathbb{P},\, \mathbb{E},\, \mathcal{A},\, \mathcal{M}_{buff},\, \mathcal{M}_{trans},\, \mathcal{M}_{arr}) \in \mathbb{C} : \exists p \in \mathbb{P} : (p,\, (\_,\, False,\, \_,\, \emptyset,\, \_,\, \_,\, \_,\, \_)) \vee (p,\, (\_,\, \_,\, \_,\, \_,\, True,\, \_,\, Pending,\, \_)) \in \mathcal{A}$. We will go over each of the possibilities in turn.

2. First, let us assume that $(p, (\_, False, \_, (), \_, \_, \_, \_)) \in \mathcal{A}$.

3. There are no rules in the definition of a computation step that change the variable $tqroot$ from $False$ to $True$. This implies that $\forall j > i : c_j = (\mathbb{P},\, \mathbb{E},\, \mathcal{A}',\, \mathcal{M}'_{buff},\, \mathcal{M}'_{trans},\, \mathcal{M}'_{arr})$ with $(p,\, (\_,\, False,\, \_,\, \_,\, \_,\, \_,\, \_,\, \_)) \in \mathcal{A}'$.

4. The only way in which the messageForest of a process with $tqroot = False$ and an empty messageForest can move to a non-empty messageForest is through the execution of the rule of definition INITIATION. However, after executing that step, $(p,\, (\_,\, \_,\, \_,\, \_,\, True,\, \_,\, Pending,\, \_))$ is in the list of attributes, and our following reasoning will apply.

5. The only rule that can be applied to $(p,\, (\_,\, \_,\, \_,\, \_,\, True,\, \_,\, Pending,\, \_))$ and change these variables is rule (f) of definition CONFIRMATION, after which $(p,\, (\_,\, \_,\, \_,\, \_,\, True,\, \_,\, True,\, \_)) \in \mathcal{A}')$

6. There are no rules that can change the variables of $(p,\, (\_,\, \_,\, \_,\, \_,\, True,\, \_,\, True,\, \_))$.

7. It follows that after each computation step that is performed, $(p,\, (\_,\, False,\, \_,\, \emptyset,\, \_,\, \_,\, \_,\, \_)) \vee (p,\, (\_,\, \_,\, \_,\, \_,\, True,\, \_,\, Pending,\, \_)) \vee (p,\, (\_,\, \_,\, \_,\, \_,\, True,\, \_,\, True,\, \_)) \in \mathcal{A}'$.

8. Therefore, $\forall j > i : c_j = (\mathbb{P}, \mathbb{E}, \mathcal{A}', \mathcal{M}'_{buff}, \mathcal{M}'_{trans}, \mathcal{M}'_{arr})$ with $(p, (\_, False, \_, \emptyset, \_, \_, \_, \_)) \vee (p, (\_, \_, \_, \_, True, \_, Pending, \_)) \vee (p, (\_, \_, \_, \_, True, \_, True, \_)) \in \mathcal{A}')$. □

This Lemma shows that for a non-initiator, when $tqroot = False$ and its messageForest is empty, this will remain the case. The initiator may get a non-empty messageForest, but this messageForest will be related to the Tok messages rather then the messages from the basic computation.

Lemma 52 *In a terminating computation* $(c_1, c_2, \ldots, c_n, \ldots)$,
$(\exists c_j = (\mathbb{P}, \mathbb{E}, \mathcal{A}', \mathcal{M}'_{buff}, \mathcal{M}'_{trans}, \mathcal{M}'_{arr}) :$
$\forall (p, q) \in \mathbb{E} : ((Tok, \_, q), (p, q)) \in \mathcal{M}'_{trans}) \rightarrow$
$((p, (\_, False, \_, \emptyset, \_, \_, \_, \_)) \vee (p, (\_, \_, \_, \_, True, \_, Pending, \_)) \in \mathcal{A}')$

Proof.

1. As tokens are only created with a neighbor of the creating process as destination, the only way in which $\forall (p, q) \in \mathbb{E} : ((Tok, \_, q), (p, q)) \in \mathcal{M}'_{trans}$ can arise is if a computation step is executed on p that creates a Tok message to each neighbor.

2. This implies that either a computation step in which the initiator initiates the YAWA part of the algorithm (rule (c) of definitions RECEIVE, PASSIVE or CONFIRMATION, or the rule from definition INITIATION) or a computation step where a non-initiator forwards the Tok messages for the first time (rule d of definitions RECEIVE, PASSIVE, CONFIRMATION or TOKEN) were performed on process p.

3. In all cases, after such a step has been performed, $(p, (\_, False, \_, \emptyset, \_, \_, \_, \_)) \vee (p, (\_, \_, \_, \_, True, \_, Pending, \_))$ is in the list of attributes.

4. According to Lemma 51, this implies that $(p, (\_, False, \_, (), \_, \_, \_, \_)) \vee (p, (\_, \_, \_, \_, True, \_, Pending, \_)) \vee (p, (\_, \_, \_, \_, True, \_, True, \_)) \in \mathcal{A}')$.

5. As there are unconfirmed Tok messages, the messageTree in the initiator can not yet have been completed, so $\neg(p, (\_, \_, \_, \_, True, \_, True, \_)) \in \mathcal{A}')$.

6. If follows that $(p, (\_, False, \_, (), \_, \_, \_, \_)) \vee (p, (\_, \_, \_, \_, True, \_, Pending, \_)) \in \mathcal{A}')$. □

This Lemma shows that when a process forwards Tok messages to all its neighbors, then either the process is not a TQRoot and its messageForest is empty, or the process is the initiator with $ter = Pending$, which implies that the process is not a TQRoot and the only messageTree in its messageForest is the messageTree representing the Tok messages.

Lemma 53 *In a computation* $(c_1, c_2, \ldots, c_n, \ldots)$,
$(\exists c_i = (\mathbb{P}, \mathbb{E}, \mathcal{A}, \mathcal{M}_{buff}, \mathcal{M}_{trans}, \mathcal{M}_{arr}) \in \mathbb{C} :$
$\exists p' \in \mathbb{P} : (p', (\_, \_, \_, \_, \_, \_, True, \_)) \in \mathcal{A}) \rightarrow$
$(\forall p \in \mathbb{P} : (p, (\_, False, \_, (), \_, \_, \_, \_)) \in \mathcal{A})$

Proof.

1. According to Lemma 50, $(\forall p \in \mathbb{P} : \exists c_j = (\mathbb{P}, \mathbb{E}, \mathcal{A}', \mathcal{M}'_{buff}, \mathcal{M}'_{trans}, \mathcal{M}'_{arr}) \in \mathbb{C} : \forall (p, q) \in \mathbb{E} : ((Tok, \_, q), (p, q)) \in \mathcal{M}'_{trans})$

2. Then, through applying Lemma 52, $\forall p \in \mathbb{P} : \exists c_k = (\mathbb{P}, \mathbb{E}, \mathcal{A}', \mathcal{M}'_{buff}, \mathcal{M}'_{trans}, \mathcal{M}'_{arr}) \in \mathbb{C} : (p, (\_, False, \_, \emptyset, \_, \_, \_, \_)) \vee (p, (\_, \_, \_, \_, True, \_, Pending, \_)) \in \mathcal{A}'$. Let us consider the first such c_k.

3. Next, we can apply Lemma 51, to obtain $\forall p \in \mathbb{P} : \forall l > k : c_l = (\mathbb{P}, \mathbb{E}, \mathcal{A}'', \mathcal{M}''_{buff}, \mathcal{M}''_{trans}, \mathcal{M}''_{arr})$ with $(p, (\_, False, \_, \emptyset, \_, \_, \_, \_)) \vee (p, (\_, \_, \_, \_, True, \_, Pending, \_)) \vee (p, (\_, \_, \_, \_, True, \_, True, \_)) \in \mathcal{A}'')$
4. For each p, it is impossible for p to send Tok messages before c_k, because those computation steps that allow the process to send Tok messages to its neighbors require $(p, (\_, False, \_, \emptyset, \_, \_, \_, \_))$. This in turn implies that $i > k$.
5. It follows that $\forall p \in \mathbb{P}$ $(p, (\_, False, \_, \emptyset, \_, \_, \_, \_)) \vee (p, (\_, \_, \_, \_, True, \_, Pending, \_)) \vee (p, (\_, \_, \_, \_, True, \_, True, \_)) \in \mathcal{A})$.
6. Since p' is the only process where $init = True$ (after all, there is exactly one process where $init = True$ in c_1, and no computation steps allow a process to go from $init = False$ to $init = True$), it follows that $\neg(p, (\_, \_, \_, \_, True, \_, Pending, \_)) \in \mathcal{A}$.
7. The only way in which $(p, (\_, \_, \_, \_, True, \_, True, \_))$ can be in $\mathcal{A}$ is after the execution of rule (f) of definition CONFIRMATION. After that rule has been executed, $(p', (\_, False, \_, \emptyset, \_, \_, \_, \_)) \in \mathcal{A}$.
8. It follows that $(\forall p \in \mathbb{P} : (p, (\_, False, \_, (), \_, \_, \_, \_)) \in \mathcal{A})$. $\Box$

This Lemma proves that when the initiator indicates termination, all processes have $tqroot = False$ and an empty messageForest.

Theorem 54 *In a computation* $(c_1, c_2, \ldots, c_n, \ldots)$,
$(\exists c_i = (\mathbb{P}, \mathbb{E}, \mathcal{A}, \mathcal{M}_{buff}, \mathcal{M}_{trans}, \mathcal{M}_{arr}) \in \mathbb{C}$:
$\exists p' \in \mathbb{P} : (p', (\_, \_, \_, \_, \_, \_, True, \_)) \in \mathcal{A}) \rightarrow$
c_i *is a terminated configuration*

Proof.

1. This follows directly from Lemmas 53 and 47.

The main theorem of this section states that when the initiator indicates termination, the computation has actually terminated.

5.4 Liveliness

This subsection proves the liveliness of the algorithm. In other words, we show that when the basic computation terminates, our algorithm will eventually indicate termination. We show this by proving that any messageTree will eventually be removed (Lemma 57), and then by showing that in each situation, the only way to remove the last messageTree is by applying rule CONFIRMATION(f), after which the process on which this rule was executed indicates termination.

Lemma 55 *In a terminating computation* $(c_1, c_2, \ldots, c_n, \ldots)$,
$(\exists c_i = (\mathbb{P}, \mathbb{E}, \mathcal{A}, \mathcal{M}_{buff}, \mathcal{M}_{trans}, \mathcal{M}_{arr}) \in \mathbb{C}$:
$\exists p \in \mathbb{P} : (p, (\_, \_, \_, forest, \_, \_, False, \_)) \in \mathcal{A} : \exists tree \in forest) \rightarrow$
$(\exists c_j = (\mathbb{P}, \mathbb{E}, \mathcal{A}', \mathcal{M}'_{buff}, \mathcal{M}'_{trans}, \mathcal{M}'_{arr}) \in \mathbb{C}$
with $j > i : (p, (\_, \_, \_, forest', \_, \_, \_, \_)) \in \mathcal{A}' \wedge tree \notin forest')$

Proof.

1. Let us assume that $\exists c_i = (\mathbb{P}, \mathbb{E}, \mathcal{A}, \mathcal{M}_{buff}, \mathcal{M}_{trans}, \mathcal{M}_{arr}) \in \mathbb{C} : \exists p \in \mathbb{P} : (p, (\_, \_, \_, forest, \_, \_, False, \_)) \in \mathcal{A} : \exists tree \in forest$

2. This $tree$ must have been created in a computation step, because in c_1 all messageForests are empty. The only computation step that allows a messageTree to be created while ter remains $False$ is rule SEND(a). When that rule was executed, a messageChain was created simultaneously, accompanying a message. We will study the progress of this messageChain through the computation.

3. According to Lemma 39, this message will eventually arrive at its destination, and because this is a terminating computation (where there exists a configuration with no messages in transit), it must be received by calling rule RECEIVECOMPLETES(a). After that rule was executed, the receiving process was active, and the message chain was stored in that process' lmc, and the $tqroot$ variable of that process was $False$.

4. From that situation, there are a number of possibilities. As this is a terminating computation, this process must eventually turn passive. If it still holds the messageChain in lmc and doesn't have any messages in its buffer at that time, it must do so by calling rule PASSIVE(e), after which a $Conf$ message was sent back to the process that created the messageTree.

5. The reason why the process might no longer be holding the messageChain in lmc is if it received a message first. If the process did so when it didn't have any messages in its buffer, it must have executed rule RECEIVE(e), which caused a $Conf$ message to be sent back to the process that created the messageTree.

6. However, it is also possible that the process did have messages in its buffer when turning passive or receiving a message. This can happen if it executed rule SEND(b), which would mean that the situation is somewhat different. In that case, a number of messages were created. Still, because the computation is terminating, the process must eventually have turned passive, in which case there are two possibilities: the process received a message before turning passive, which would cause the execution of rule RECEIVE(f), or the process turned passive by executing rule PASSIVE(f).

7. In both cases, each of the messages that were sent was accompanied by a messageChain that had the same first messageInfo as the messageChain that was originally created. So for each of these messages, exactly the same reasoning as above applies.

8. Next, we should point out that each time new messages are created, there are computation steps that can be applied: either the message can be delivered to its destination, it can be stored in a lmc, rule SEND(b) can create new messages, or rule (f) from definitions RECEIVE or PASSIVE can be applied. In Lemma 36 we have seen that eventually a situation will be reached where no rule is applicable. This implies that for each of the messageChains created in this way, eventually rule SEND(b) is not applied. This means that each of the message chains created in this way is eventually sent back to the process that created the messageTree, accompanying a $Conf$ message.

9. As long as these $Conf$ messages have not reached their destination, one of the rules from definition DELIVERY will be applicable. Lemma 36 therefore implies that eventually these $Conf$ messages arrive at their destination.

10. As long as not all $Conf$ messages have been received, one of the rules from definition CONFIRMATION can be applied. Due to Lemma 36, this means that eventually one of them will be executed.

11. If any rule other then CONFIRMATION(a1) is applied, $tree$ is removed from the forest, and the statement follows. We will therefore have to prove that eventually, a rule other then CONFIRMATION(a1) must be applied. We will prove this by showing that when the last of these messageChains is added to $tree$, $tree$ is a complete messageTree.

12. The first thing that should be observed by studying the rules that have added to this messageTree (rule SEND(a), RECEIVE(f) and PASSIVE(f)), is that whenever new messageChains are created, a new messageInfo is created with a unique identifier, and with n equal to the number of new messageChains created.

13. A second observation is that when a single messageChain mc results in the creation of multiple messageChains, each of those newly created messageChains will share the same prefix $mc + info$. As a result, when these messageChains are added to the messageTree, they create a single child for the branch of the messageTree that is represented by mc.

14. Similarly, when a single messageChain mc does not result in the creation of any messageChains, the receipt of a $Conf$ message will result in a single child $((Comp, p, 0), ())$ for the branch of the messageTree that is represented by mc.

15. The above three points imply that when all messageChains have been added to the messageTree, each node in $tree$ will have a number of children exactly equal to n, and all leaf nodes of $tree$ are of the form $((Comp, p, 0), ())$.

16. This implies that $tree$ is a complete messageTree, which means that rule CONFIRMATION(a1) wasn't applied. It follows that $tree$ was removed from $forest$ upon execution of the computation step that removed the last $Conf$ message from $\mathcal{M}$. □

This Lemma shows that any tree representing basic messages will eventually be removed from the corresponding messageForest.

Lemma 56 *In a terminating computation* $(c_1, c_2, \ldots, c_n, \ldots)$,
$(\exists c_i = (\mathbb{P}, \mathbb{E}, \mathcal{A}, \mathcal{M}_{buff}, \mathcal{M}_{trans}, \mathcal{M}_{arr}) \in \mathbb{C}:$
$\exists p \in \mathbb{P} : (p, (\_, \_, \_, forest, \_, \_, Pending, \_)) \in \mathcal{A} : \exists tree \in forest) \rightarrow$
$(\exists c_j = (\mathbb{P}, \mathbb{E}, \mathcal{A}', \mathcal{M}'_{buff}, \mathcal{M}'_{trans}, \mathcal{M}'_{arr}) \in \mathbb{C}$
with $j > i : (p, (\_, \_, \_, forest', \_, \_, \_, \_)) \in \mathcal{A}' \wedge tree \notin forest')$

Proof.

1. Let us assume that $\exists c_i = (\mathbb{P}, \mathbb{E}, \mathcal{A}, \mathcal{M}_{buff}, \mathcal{M}_{trans}, \mathcal{M}_{arr}) \in \mathbb{C} : \exists p \in \mathbb{P} : (p, (\_, \_, \_, forest, \_, \_, Pending, \_)) \in \mathcal{A} : \exists tree \in forest$

2. The requirements for applying a rule that changes ter from $False$ to $Pending$ imply that $(p, (\_, False, \_, forest, True, \_, Pending, \_)) \in \mathcal{A}$. Furthermore, in the computation step where ter changed from $False$ to $Pending$, a number of messageChains were created, each accompanying a Tok message.

3. From Lemma 39, we know that these Tok messages will arrive at their destination. When this happen, a rule from definition TOKEN becomes applicable, and one of these rules will remain applicable for as long as no rule of this definition has been executed. Lemma 36 then implies that one of them will be executed eventually.

4. When a Tok message is received at its destination, there are three possibilities. The destination is the initiator, it is a non-initiator and the Tok message is the first Tok message to be received by this process, or it is a non-initiator and the process received a Tok message before.

5. If the destination is the initiator, rule TOKEN(c) will be applied, and the messageChain will be sent back to the initiator accompanying a $Conf$ message.

6. If the destination received a Tok message before, rule TOKEN(b) will be applied as a messageChain has been stored in the fmc variable of the process, and the messageChain will be sent back to the initiator accompanying a $Conf$ message.

7. On the other hand, if the destination did not receive a Tok message before, the situation is slightly more complicated. If both its messageForest is empty and its *tqroot* variable is $False$, rule TOKEN(d) will be applied, forwarding a new set of messageChains accompanying a new set of tokens to all neighbor processes, while storing the messageChain in its fmc variable. Note how the identifier of the new messageInfo in this chain is unique, and how the n in this messageInfo corresponds to the number of new messageChains sent out.

8. If the messageForest of the receiving process is not empty, but $tqroot = False$, the messageChain gets stored in the fmc variable of this process. Now, because the process storing the messageChain is not an initiator, Lemma 55 can be applied, to conclude that eventually all messageTrees must be removed from the messageForest of the process. When the last messageTree is removed from the messageForest, rule CONFIRMATION(d) must be applied, since that is the only rule whose parameters fit the given situation. This will forward a new set of messageChains accompanying a new set of tokens to all neighbor processes. Note how the identifier of the new messageInfo in this chain is unique, and how the n in this messageInfo corresponds to the number of new messageChains sent out.

9. Finally, if the process has $tqroot = True$, then regardless of whether there are trees in the messageForest of the process or not, the messageChain will be stored in the fmc variable. Now because the computation is terminating, the process must turn passive at least once. When this happens, if *tqroot* is still $True$, *tqroot* will become $False$. However, *tqroot* may have become $False$ earlier when the process tried to receive a message. Depending on the exact situation, rules (a) or (d) from definitions RECEIVE or PASSIVE were applied. If rule (a) from one of these definitions was applied, we would end up in the situation where $tqroot = False$ with a non-empty messageForest and a messageChain stored in fmc, of which we have seen in the previous point that it results in rule CONFIRMATION(d) to be applied. If, on the other hand, rule (d) from one of these definitions was applied, a new set of messageChains is forwarded, accompanying a new set of tokens to all neighbor processes. Note how the identifier of the new messageInfo in this chain is unique, and how the n in this messageInfo corresponds to the number of new messageChains sent out.

10. Each time such a Tok message arrives at a process that did not receive a Tok message before, the number of processes that has not yet received a Tok message is reduced by one. As the total number of processes is finite, this can only happen a finite number of times. We must therefore conclude that eventually all messageChains are sent back to the initiator accompanying $Conf$ messages.

11. We have seen that whenever new messageChains are created, a new messageInfo is created with a unique identifier, and with n equal to the number of new messageChains created.

12. We have also seen that that when a single messageChain mc results in the creation of multiple messageChains, each of those newly created messageChains will share the same prefix $mc + info$. As a result, when these messageChains are added to the messageTree, they create a single child for the branch of the messageTree that is represented by mc.

13. Finally, when a single messageChain mc did not result in the creation of any messageChains, the receipt of a $Conf$ message will result in a single child $((Comp, p, 0), ())$ for the branch of the messageTree that is represented by mc.

14. The above three points imply that when all messageChains have been added to the messageTree, each node in $tree$ will have a number of children exactly equal to n, and all leaf nodes of $tree$ are of the form $((Comp, p, 0), ())$.

15. This implies that $tree$ is a complete messageTree, which means that rule CONFIRMATION(a1) wasn't applied on the receipt of the last $Conf$ message. It follows that $tree$ was removed from $forest$ upon execution of the computation step that removed the last $Conf$ message from $\mathcal{M}$, which must have happened by executing rule CONFIRMATION(f). $\square$

This Lemma shows that the tree representing the Tok messages will eventually be removed from its messageForest.

Lemma 57 *In a terminating computation* $(c_1, c_2, \ldots, c_n, \ldots)$,
$(\exists c_i = (\mathbb{P}, \mathbb{E}, \mathcal{A}, \mathcal{M}_{buff}, \mathcal{M}_{trans}, \mathcal{M}_{arr}) \in \mathbb{C}:$
$\exists p \in \mathbb{P} : (p, (\_, \_, \_, forest, \_, \_, \_, \_)) \in \mathcal{A} : \exists tree \in forest) \rightarrow$
$(\exists c_j = (\mathbb{P}, \mathbb{E}, \mathcal{A}', \mathcal{M}'_{buff}, \mathcal{M}'_{trans}, \mathcal{M}'_{arr}) \in \mathbb{C}$
with $j > i : (p, (\_, \_, \_, forest', \_, \_, \_, \_)) \in \mathcal{A}' \wedge tree \notin forest')$

Proof.

1. This follows from combining Lemmas 56 and 55, as the requirements for turning ter to $True$ dictate that no more messageTrees can be created in that process.

This Lemma shows that eventually each messageTree will be removed from its corresponding messageForest.

Theorem 58 *In a terminating computation* $(c_1, c_2, \ldots, c_n, \ldots)$,
$\exists c_i = (\mathbb{P}, \mathbb{E}, \mathcal{A}, \mathcal{M}_{buff}, \mathcal{M}_{trans}, \mathcal{M}_{arr}) \in \mathbb{C}:$
$\exists p \in \mathbb{P} : (p, (\_, \_, \_, \_, \_, \_, True, \_)) \in \mathcal{A}')$

Proof.

1. Let us consider the process where $init = True$. According to the definition of a potential initial configuration, exactly one such process must exist in c_1. Furthermore, there is no computation step that turns $init$ to $False$. If follows that there is exactly one such process in each configuration of the computation. This process starts with $ter = False$ according to definition 19.

2. Also according to definition 19, either $s = active \wedge tqroot = True$ or $s = passive \wedge tqroot = False$ for the process where $init = True$. We will look at each possibility in turn.

3. First, let us assume that $s = passive \wedge tqroot = False$. For this process, the rule of definition INITIATION is applicable, as $tqroot = False$, $forest = \emptyset$, $init = True$, $fmc = ()$ and $ter = False$. Furthermore, no rules exists that can be applied to make this rule no longer applicable. Because of Lemma 36, it follows that this rule must eventually be applied. This will create a single messageTree in the messageForest of this process.

4. According to Lemma 57, this messageTree must eventually be removed from its messageForest. The only way this is possible is through applying rule CONFIRMATION(f), after which the theorem follows.

5. Next, let us assume that $s = active \wedge tqroot = True$ in c_1.

6. Since this is a terminating computation, this process must have turned passive at least once. Let us consider the first time this process turned passive. This must have happened through a rule of definition PASSIVE. Either this was the application of rule (c) of this definition, or through the application of a different rule. We will consider each possibility.

7. If rule PASSIVE(c) was applied, a single messageTree was created in the messageForest of this process. According to Lemma 57, this messageTree must eventually be removed from its messageForest. The only way this is possible is through applying rule CONFIRMATION(f), after which the theorem follows.

8. If rule PASSIVE(c) wasn't applied, it was because this rule wasn't applicable (as each rule excludes the others). there are a number of possible ways in which this could have happened. Either the process received a message before turning passive, which would have turned $tqroot = False$, or the messageForest of the process wasn't empty. Let us consider both possibilities.

9. First, let us assume that a message was received before turning passive, and let us consider the first time this happened. At that time, the process either applied rule RECEIVE(c), or it applied a different rule from definition RECEIVE.

10. If rule RECEIVE(c) was applied, a single messageTree was created in the messageForest of this process. According to Lemma 57, this messageTree must eventually be removed from its messageForest. The only way this is possible is through applying rule CONFIRMATION(f), after which the theorem follows.

11. If a different rule was applied, rule RECEIVE(c) must have been inapplicable (as each rule excludes the others). This can't have been the result of $tqroot = False$, as this is the first time the process receives a message, and the first time the process will turn passive will happen later. We must conclude that rule RECEIVE(c) was inapplicable because the messageForest of the process wasn't empty.

12. So now, let us consider the possibility that the messageForest of the process wasn't empty the first time the process either received a message or turned passive.

13. At this point in time, there must have been a finite number of messageTrees in this messageForest (as only a finite number of computation steps have been performed at that time). Furthermore, because $tqroot = False$ after the first time the process either received a message or turned passive, no further messageTrees will have been added to the messageForest of the process.

14. According to Lemma 57, eventually each of these messageTrees will be removed through the receipt of a $Conf$ message. The last tree to be removed in this way must be removed by applying rule CONFIRMATION(c), as this is the only rule that will be applicable in that situation.

15. When rule CONFIRMATION(c) was applied, a single messageTree was created in the messageForest of this process. According to Lemma 57, this messageTree must eventually be removed from its messageForest. The only way this is possible is through applying rule CONFIRMATION(f), after which the theorem follows.

16. In all possible situations, $\exists c_i = (\mathbb{P}, \mathbb{E}, \mathcal{A}, \mathcal{M}_{buff}, \mathcal{M}_{trans}, \mathcal{M}_{arr}) \in \mathbb{C} : \exists p \in \mathbb{P} : (p, (\_, \_, \_, \_, \_, \_, True, \_)) \in \mathcal{A}')$. $\square$

The final theorem proves that in a terminating computation, eventually a process will indicate that termination has occurred.

6 *Discussion*

6.1 Message Complexity

6.1.1 Transitory Quiescence Part of the BTTF Wave Algorithm

It is easy to see that the number of control messages sent by the Transitory Quiescence part of the algorithm will never exceed the number of basic messages sent by the computation. The only control messages sent by the algorithm are confirmation messages confirming the receipt of an entire message chain. This can only happen once per message chain received, and since each message is accompanied by at most one message chain, a confirmation message is sent at most once for each basic message that was received.

In the worst case, that is exactly what happens. When all messages are sent by TQRoot processes, each message gets confirmed eventually, leading to a confirmation message for each basic message sent. However, this only happens when all message chains are of length one. If message chains become longer, the number of control messages goes down in comparison to the number of basic messages, because messages earlier in the chain are confirmed implicitly.

The exact effect is dependent on the nature of the main computation. The more often messages are sent in response to other messages, the less control messages are needed. An ideal message tree from the point of view of this algorithm has a large height and a small width, because the ratio of leaf nodes (which send confirmation messages) to edges (which represent basic messages) becomes smaller. In the best case scenario, all basic messages are sent in response to a different basic message, creating a single long message chain culminating in a single control message.

As another example, if every process either sends out two messages whenever it receives a message or sends a confirmation message, a binary message tree will be created. In such a tree, the number of external nodes is 1 higher than the number of internal nodes, as has been proven in [10]. Since the root of such a tree does not have an incoming edge, the number of edges leading to external nodes is 2 higher than the number of edges leading to internal nodes. This in turn implies that the number of confirmation messages in a single tree will equal $\frac{1}{2}M + 1$.

6.1.2 YAWA Part of the BTTF Wave Algorithm

The message complexity of the YAWA part of the algorithm is easy to evaluate. Each process sends a token to each of its neighbors once. This means that all channels get traversed by tokens twice, once in each direction. Furthermore, in the worst case a confirmation message must be sent for each of these tokens. This means that in the worst case, YAWA sends $4E$ control messages. Of course, the number of confirmation messages will be somewhat reduced because control messages that spawned more tokens don't get confirmed, but that only gives a significant advantage if the number of channels per process is small, to keep the message trees small in width.

6.2 Comparison With Existing Work

6.2.1 The Shavit Francez Algorithm

The most obvious comparison for the Transitory Quiescence algorithm is the algorithm described by Shavit and Francez in [19], which is based on the algorithm of Dijkstra and Scholten [7]. Just like Transitory Quiescence, the algorithm of Shavit and Francez also creates several trees in processes in order to keep track of the activity of the computation, and gathers this information using a single wave.

The exact way this tree is constructed and what it represents differs significantly though. In their algorithm, the construction of a tree is implicit. When a process that was not yet considered to be part of a tree receives a basic message, it is considered to be a child of the process that sent the basic message.

It remains a part of the tree until it has sent a confirmation message to its parent, which it will not do until it has received confirmation messages for all the messages that it sent while it was part of the tree. In such an implicit tree, each node represents a process, and the parent-child relation represents the fact that the parent originally activated the child. In our algorithm on the other hand, the tree is constructed explicitly in one of the processes. Furthermore, each node represents a batch of messages that were sent out at the same time rather than a process, and a parent-child relation indicates the fact that one of the messages of the batch represented by the parent could be considered the direct cause of the batch represented by the child.

The main difference between the two algorithms lies in the behavior when sending confirmation messages, though. In the algorithm by Shavit and Francez, confirmation messages are sent directly to the process that sent the basic message of which they are confirming the receipt, either immediately if the receiving process was already considered part of the tree, or after a delay if the basic message caused the process to become part of the tree. This contrasts strongly with the behavior of Transitory Quiescence, which doesn't sent any confirmation messages unless there was no further activity, in which case a confirmation message is sent directly to the root of the tree.

As we have seen before, this doesn't make a difference in the worst case, because both algorithms sent out a number of confirmation messages exactly equal to the number of basic messages sent. However, in the average and best case, Transitory Quiescence requires significantly less control messages than exactly one for each basic message, giving it a better message complexity.

6.2.2 Chang's Echo Algorithm

An obvious comparison for YAWA is Chang's Echo Algorithm, first described by Chang in [4] and later refined by Segall [18] as described in [20]. This algorithm uses a method of token forwarding similar to the YAWA algorithm. The initiator forwards a token to all its neighbors, whereas other processes forward the token to all neighbors except the neighbor from which they received the first token. These tokens will function like confirmation messages for processes that had already been reached by the wave. When a process has received such a confirmation from all its neighbors, it can send a token to the process from which it received its first token. Tokens will echo back from the edges of the graph, guaranteeing that all processes have been reached when the initiator has received confirmation tokens from all its neighbors.

This algorithm traverses every edge exactly twice. Since the edges back to the initiator are traversed last, no further confirmation messages are necessary. Therefore, this algorithm sends exactly $2E$ messages, and is therefore significantly more efficient than the $4E$ bound of YAWA. However, as we'll see in 6.3.1, it is possible to optimize YAWA in such a way that it is likely to come close to this message complexity, and in the best case even surpass it.

6.2.3 Wave Based Algorithms

Rather than using them to collect information from TQRoots, waves can also be used as the basis for a termination detection algorithm in their own right. One of the best known of these algorithms is the algorithm proposed by Dijkstra, Feijen and van Gasteren [6]. Even though it assumes synchronous communication, similar principles have served as the basis for a number of algorithms based on asynchronous communication as well (see, for example, [5] or [17]). Because of this, it is still interesting to compare their algorithm to ours.

In the algorithm by Dijkstra, Feijen and van Gasteren, the processes are assumed to be organized in a ring. A token is repeatedly sent through this ring. A process only forwards the token if the process is passive, which has as idea that the token only completes a round through the ring if all processes are passive. Unfortunately, this isn't the case, as a message sent by a process may activate a process that has already been visited by the token. To detect this situation, the token and all processes start out colored white. If a process sends a message, it colors itself black. If a black process handles the token, it colors the token black and itself white. When a black token completes a lap through the ring, it is colored white again and a new round is started. If a white token completes a lap through the ring, it can be concluded that no processes sent any messages since the last time the token passed, and that all processes are passive since they forwarded the token. Given the fact that there is synchronous communication, it is therefore impossible that there are messages in transit, which means that termination can be concluded.

Since token passing in a ring is a special case of a wave algorithm, this principle can be generalized to a general wave algorithm on an arbitrary network structure. The same principle of coloring processes and tokens works perfectly fine for synchronous communication, as has been discussed in [20].

This type of algorithm has a significant downside when compared to our algorithm though. In the worst case, it needs to send a wave for each message that is exchanged in the main computation. And even though in general a large number of messages will be exchanged in between waves, the number of waves required will remain dependent on the number of messages exchanged. As a result, the message complexity of these type of solutions tends to be worse than the number of messages exchanged by our algorithm.

6.2.4 Credit Distribution

Finally, it is worth comparing the BTTF Wave algorithm to algorithms based on credit distribution. These algorithms were described by Mattern [16] and by Huang [11]. Just like BTTF Wave, they make use of piggybacking of information in order to reduce the number of control messages. They make use of a central processor, named the controller, which starts the algorithm by distributing a total weight of 1 over all processes that start out active. Whenever an active process sends a message, it sends along part of the weight that it had been assigned. Whenever a process receives a message, it adds the weight that accompanies this message to its own. Because of these two rules, all active processes and all messages are assigned a weight. Whenever a process turns passive, it sends the weight it had been assigned back to the controller. When the total weight at the controller equals one again, the controller knows that all processes are passive and there are no messages in transit, and concludes that the computation has terminated.

In the worst case, each basic message activates a process, which turns passive again without receiving more messages. If this happens, the credit distribution algorithm requires a number of control messages equal to the number of basic messages in the computation (plus a small number of initiation messages), just like our algorithm. However, in general a process will receive multiple messages before turning passive, which reduces the number of control messages required. In the best case, no process becomes activated after it has turned passive, which would mean that the number of control messages required would equal the number of processes that participate in the computation.

The advantages in message complexity are therefore extremely similar to the advantages of the BTTF Wave algorithm. However, the situation in which optimal message complexity is reached differs significantly. Where credit distribution algorithms want to receive large numbers of messages before turning passive, this behavior is completely irrelevant to the message complexity of BTTF Wave. BTTF Wave prefers that a small number of messages is consistently sent in reply to each message received, behavior that is completely irrelevant to credit distribution algorithms.

6.3 Features and Optimizations of the Original Design

There are a number of features and optimizations that were part of the original design of the BTTF Wave algorithm by Farhad Arbab, but which have thus far been ignored. As they are not essential to the core idea of the algorithm, their inclusion would have significantly complicated both the description and the correctness proofs discussed thus far. However, they do introduce a number of interesting improvements and tradeoffs. Therefore, we would like to discuss them here, along with the impact of their inclusion on the message complexity of the algorithm.

6.3.1 Improving the YAWA Algorithm

One of the possible improvements is a simple modification of the YAWA algorithm. The reason a process sends a token to all of its neighbors is to make sure that each process gets reached by a token at least once. However, a process can only send a token after it has been reached by a token at least once. Therefore, there is no need for a process to send tokens to processes from which it already received a token. There will always be at least one process that can be excluded this way, but if the process delays forwarding the token because the message forest isn't empty, it is quite likely that tokens from processes other than the first will reach it as well.

Adding such a modification would drastically reduce the number of control messages sent by YAWA. If message forests become empty at different times, it is unlikely that such token messages will cross in transit, which in turn implies that each channel would only be traversed by a control message once. These tokens would still require confirmation messages, but the average number of control messages would be closer to $2E$ than to the original $4E$, making the algorithm roughly as efficient as Chang's echo algorithm. In fact, in the best case, when all processes are organized in a single line and the initiator is located at an end of the line, the number of control messages might be as low as $E + 1$, because each edge is traversed once, and a single message confirms the entire message chain.

The assumption that control messages do not cross each other in transit is a rather big one, though, especially when considering two processes that started out with an empty forest. It might be necessary to introduce delays of random lengths before forwarding tokens, in order to allow tokens sent by other processes the time required to arrive.

6.3.2 Reduction of the Length of Message Chains

An optimization deals with the potential problems of the length of the message chains. Even though piggybacking information onto an existing message is relatively cheap, the length of the message chains may grow to such an extent that they overshadow the length of the messages and become the main factor determining the transmission delay of a message. When the length of the message chains becomes a problem, it can be reduced at the cost of additional control messages. Rather than sending the message chain on unaltered, a process can send a control message to the root of the message tree. This control message has two functions. First of all, it functions as a confirmation message for the message chain that arrived at the process. Secondly, it informs the root of the message tree that it needs to create a new message tree, to represent the subtree that is the result of the batch of messages that this process will send. The process then continues to send message chains with as first element the root of the original message chain, but with all other steps in the message chain left out.

Each such a reduction in length of a message chain increases the number of control messages sent by 1. Interestingly enough, if this procedure is used to keep the length of the message chains equal to 1, a confirmation message is sent for each message received in the system, with the only remaining difference to the Shavit Francez algorithm being the order in which confirmation messages are sent.

6.3.3 Receive with Timeout

Thus far, we have assumed that any call of receive by a process will block until a message arrives at that process. This allowed us to stop buffering messages sent by the process when receive is called, because we know that upon completion of the receive statement another message will have arrived, allowing further messages sent by the process to be considered caused by that message.

The assumption does have a drawback, though. It requires the programmer to know when a message is guaranteed to eventually arrive, because if receive is called without a message arriving, the program will block indefinitely, causing a deadlock.

The solution to this possibility is implementing a receive with timeout. When a process performs a receive with timeout, the call will block until either a message arrives, or until the specified amount of time has passed. However, allowing a receive with timeout loses the advantage of knowing that upon completion of the statement a message has arrived. This implies that if the receive statement times out without a message arriving, any messages this process (which remains active) will send in the future must be considered part of the same message tree as the messages that are buffered when the receive with timeout is called. Simply clearing the buffer is therefore not an option, as the number of siblings noted in the message chain accompanying the buffered messages would not take into account the possible presence of siblings that were created later.

The solution that has been chosen by Farhad Arbab to solve this problem is the same that allows for the reduction of the length of message chains. Rather then sending on the message chain unaltered, the buffered messages, along with any other messages this process may send before it receives a message, form the root of a new subtree, which becomes stored in the message forest of this process. The message chain that spawned that subtree is also retained at the process. As branches of this subtree complete, the process will receive confirmation messages, allowing it to fill out the various parts of the subtree. When the subtree becomes complete and is removed from the message forest, the process knows that the message chain it has retained did not spawn any activity that is still part of the system, so it can then send a confirmation message to the root of the original tree.

Doing so introduces an additional confirmation message, though, as each of the leaves of the subtree would normally send their confirmation message directly to the root of the entire tree rather then to the root of the subtree.

6.3.4 No Delay on Basic Messages

If the delivery time of a message is critical despite the network only guaranteeing delivery in finite time, it may not be acceptable to delay pushing the message onto the channel. In that case, the delaying of messages until a process turns passive or receives a message can be turned off. That decision would have some significant consequences though.

The delay was originally introduced in order to allow later messages to be added as children to the same node. If the delay is removed, this will no longer be easily possible. The reason for this is that the root of the tree would have no method available to determine whether or not it should expect additional confirmation messages, and therefore can not determine whether or not it is safe to remove the message tree.

The solution to this problem is the same as for the previous two: have the process create a subtree, and confirm the original message tree when that subtree becomes complete.

6.3.5 Fault Tolerance

Since the root of a message tree keeps track of all messages in that message tree, it is relatively easy to adapt the algorithm to make it fault tolerant, under the assumption that messages sent to a faulty process are returned to their sender in finite time. The main problem for the BTTF Wave algorithm that arises when a process fails is that messages may go unconfirmed. This can happen in one of two ways. The first possibility is that the process fails before receiving a normal message. In this case, the process that sent the message will receive it back in finite time. Even though the message was never received, the process that sent it knows that this message will not result in any more messages. It can therefore act as if the failed process received the message and failed immediately after, sending a confirmation message to the root of the tree in the failed process' stead. The other possibility is that the process receives the message, and then fails at some point afterward. In this situation, the failed process may have sent messages before failing. Unfortunately, there is no way to determine the actual presence of these messages, because this information was only available at the failed process. Duplicating this information could make the problem significantly less likely to occur, but comes at a cost of a large amount of extra control messages. We therefore consider the inability to detect messages by a failed process as collateral damage. When the root of a message tree detects that a process has failed, it assumes that the failed process did not send any messages, unless the root receives a messageChain that indicates otherwise. As a result, in case of a process failure it is possible (though not very likely) that the algorithm detects termination even though there are still messages in transit. However, in the majority of cases, these messages will result in a confirmation message to the root of the message tree before the algorithm indicates termination. It should be noted in particular though that if the root of a message tree itself fails, that entire tree is considered to be collateral damage, in which case no guarantees with regard to termination detection can be made.

Given this, the root of the tree can detect a failure by sending control messages, which will be called probing messages, to processes that received unconfirmed messages, asking if that process forwarded the corresponding message chain, and if so, to what processes. If it detects a failure in this way, it can consider the corresponding message chains confirmed. In order to limit the total number of probing messages generated this way, the frequency of these messages should be set rather low. Note that it may be convenient to change the notion of a messageInfo object to include the destination of the message that it accompanied. That way, the TQRoot can determine for which messages it already received a confirmation, thus reducing the number of probing messages it needs to send.

A process that fails does not have a significant impact on the YAWA part of the algorithm, again provided that you're willing to accept the results of messages sent by a failed process as collateral damage. If a process fails, this can be detected in the same way as the Transitory Quiescence algorithm detects this, allowing the TQRoot to ignore those branches of the message tree that involve a failed process. However, each time this happens, another wave becomes necessary. The reason for this is that there may be processes that have not failed, but which hold a token that is part of the message tree that passed through the failed process. When the failure is detected, that part of the tree is disregarded, whereas any further tokens arriving at the living process are sent back immediately.

6.4 The Importance of Clearing the Buffer

An attribute of the BTTF Wave algorithm that has to be taken into account when creating an API for this algorithm is the way it uses a receive statement by a process to indicate that messages should no longer be buffered. If this isn't taken into account when creating an API, various problems can easily arise, as can be seen from the two examples given below.

6.4.1 Clearing the Buffer when Checking for Messages

If an API contains a method that allows a programmer to check if a message has arrived without flushing any messages that have been buffered, deadlocks can occur. Consider, for example, the following program, in which checkForMessage() checks if a message has arrived without clearing the buffer.

Algorithm *Message Delay measurement*

```
Process A
% sends a message to Process B, then keeps sending messages to Process C until it receives a response.
    receive(START);
    int x = 1;
    send (CONTINUE,x) to B;
    boolean receivedMessage = false;
    while not receivedMessage do
```

```
        x += 1;
        send (CONTINUE,x) to B;
        receivedMessage = checkForMessage();
    x += 1;
    send (STOP,x) to B;
    receive(message);
```

Process B

```
% sends a response to a message from process A.
    send (START) to A;
    receive(string s, int y);
    send message to A;
    receivedMessages = 1;
    if s = STOP then
        x = y;
    else
        x = 0;
    while receivedMessages ≠ x do
        receive(string s, int y);
        receivedMessages += 1;
        if s = STOP then
            x = y;
    print x;
```

Normally, this computation terminates. Process B sends a message to A, which arrives in finite time. Upon receiving that message, Process A sends a message to Process B, which arrives in finite time. When that message arrives, Process B sends a response to Process A, which again arrives in finite time. After the response has arrived, Process A sends a STOP message, and receives the message sent by Process B. In the meantime, Process B keeps calling receive until it receives a STOP message, which tells that process exactly how many messages Process A sent. It receives this number of messages, then prints how many messages were sent by Process A.

If we apply an incorrect API for the BTTF Wave algorithm to these processes, something interesting happens in Process A. The process starts out passive, and becomes active upon receiving the START message from B. It stores the accompanying message chain in order to be able to append the messages it sends. Then, when the process calls send(CONTINUE,x), this message is buffered until the process becomes passive or calls receive. However, Process A never does either of those. It keeps sending messages to B, so it remains active. And as it can check whether messages have arrived, it never calls receive. This in turn means that the message to Process B never gets send, which means that B never sends a response, so as a result of our algorithm, the computation suddenly no longer terminates!

This deadlock can easily be avoided if a receive statement is used to check for the arrival of messages. A blocking receive statement obviously won't work, but a receive with a timeout of 0 will allow the programmer to check for message arrival while performing a receive and therefore avoiding the deadlock.

6.4.2 Clearing the Buffer in Non-Terminating Computations

The importance of clearing the buffers must also be kept in mind when applying the BTTF Wave algorithm on a non-terminating computation. A possible way in which a computation might not terminate is if a single process (after receiving a single message) continually sends messages, without ever receiving a message again and without ever turning passive. In a normal computation, these messages will eventually arrive at their destination, where the receiving process can call receive to consume them. However, when such a non-terminating process is implemented with an API of the BTTF Wave algorithm that does not clear the buffer on top, the messages sent by the process will no longer arrive! When the process first receives a message, it will stop being a TQRoot. This means that all messages that are later sent by the process will be stored in the buffer until the process either receives a message or until it becomes passive. Since neither happens, the messages will remain in the buffer and will never arrive at their destination.

Again, this problem can easily be avoided by slightly modifying the basic algorithm if the implementation is expected to run on non-terminating computations.

6.5 Empirical Results

An implementation of the algorithm designed by Farhad Arbab has been created by Kees Blom in Java, making full use of all features and optimizations described.

This implementation of the algorithm was tested on a computation where each process randomly decided if it was going to send a group of messages to a number of different processes, if it was going to send a group of messages while ignoring the delay, if it was going to receive a message (with a timeout, as the arrival of a message could not be guaranteed) or whether it would turn passive. This computation was run on a variety of different process configurations, including among others a line configuration, a star configuration, and a ring configuration. As this test algorithm worked probabilistically, termination of the computation could not be guaranteed. Instead, the computation was allowed to run until a certain threshold of basic messages was sent, and the behavior of the algorithm was observed.

It should be noted that this random behavior does not create a situation that is optimal for the BTTF algorithms. In general, message trees will be rather wide, as messages are sent to a rather large number of other processes. Furthermore, message chains will remain relatively short, as any process receiving a message will have a fifty percent chance of either calling receive or turning passive. It therefore came as no surprise that the testing results seemed to indicate that the number of control messages generally was of the same order of magnitude as the number of basic messages that were sent. Further testing seemed to show that the number of control messages was not dependent on the number of processes, again as predicted.

7 *Conclusion*

In this paper we have studied the details of the BTTF Wave algorithm, with the goal of providing a better understanding of this algorithm. We have done this by describing the core of the algorithm informally, as well as by proving both its safety (if the algorithm indicates that the computation has terminated, then the computation has actually terminated) and its liveliness (if the computation terminates, then the algorithm will eventually indicate termination).

Based on this description and proof, we have concluded that the BTTF Wave algorithm is a viable algorithm for termination detection. This conclusion led us to investigate its advantages and disadvantages when compared to different algorithms currently available. These comparisons were generally favorable, with the BTTF Wave reaching optimal message complexity in the worst case, and significant better message complexity in the best and average cases. The only other algorithm that reached comparable message complexity was the credit distribution algorithm. However, this algorithm achieved its optimal performance under significantly different circumstances, which means that the BTTF Wave algorithm adds the opportunity for improvement where that didn't exist before.

We have also seen a number of features and optimizations that were part of the design of the algorithm by Farhad Arbab, but which weren't an essential part of the core idea of the algorithm. These modifications still require a clear, formal description, which would need to be pursued during further research.

The main conclusion to draw from this paper though is that the BTTF Wave algorithm is a correct termination detection algorithm, that in the worst case performs optimally, and in the average and best case performs significantly better than the main algorithms for this problem.

8 *References*

1. Chandy, Lamport (1985) 'Distributed snapshots: Determining global states of distributed systems' *ACM Transactions on Computing Systems* 3(1):63-75

2. Chandy, Misra (1986) 'An Example of Stepwise Refinement of Distributed Programs: Quiescence Detection' *ACM Transactions on Programming Languages and Systems* 8(3):326-343

3. Chandy, Misra (1986) 'How processes learn' *Distributed Computing* Vol 1, Iss 1, 40-52

4. Chang (1982) 'Echo Algorithms: Depth Parallel Operations on General Graphs' *IEEE Transactions on Software Engineering* 8 (4): 391-401

5. Dijkstra (1987) 'Shmuel Safra's version of termination detection' *EWD-Note* 998

6. Dijkstra, Feijen and van Gasteren (1983) 'Derivation of a termination detection algorithm for a distributed computation' *Information Processing Letters* 16(5):217-221

7. Dijkstra and Scholten (1980) 'Termination detection for diffusing computations' *Information Processing Letters* 11(1):1-4

8. Eriksen (1988) 'A termination detection protocol and its formal verification' *Journal for Parallel and Distributed Computing* 5:82-91

9. Francez (1980) 'Distributed Termination' *ACM Transactions on Programming Languages and Systems* 2(1):42-55

10. Goodrich, Tamassia (2006) 'Algorithm Design' *John Wiley & Sons*

11. Huang (1989) 'Detecting termination of distributed computations by external agents' *Proceedings of the 9th international conference on Distributed Computing Systems* 79-84

12. Lifflander, Miller, Kale (2013) 'Adoption protocols for fanout-optimal fault-tolerant termination detection' *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming* 13-22

13. Lim, Chung, Gil, Suh, Yu (2013) 'An Unstructured Termination Detection Algorithm Using Gossip in Cloud Computing Environments' *Lecture Notes in Computer Science* 7767:1-12

14. Mahapabra, Dutt (2007): 'An efficient delay-optimal distributed termination detection algorithm' *Journal of Parallel and Distributed Computing* 67(10):1047-1066

15. Matocha, Camp (1998): 'A taxonomy of distributed termination detection algorithms' *Journal of Systems and Software* 43(3):207-221

16. Mattern (1989) 'Global quiescence detection based on credit distribution and recovery' *Information Processing Letters* 30:195-200

17. Misra (1983) 'Detecting termination of distributed computations using markers' *ACM SIGACT-SIGOPS Symposium on principles of distributed computing*

18. Segall (1983) 'Distributed Network Protocols' *IEEE Transactions on Information Theory* 29:23-35

19. Shavit, Francez (1986) 'A new approach to detection of locally indicative stability' *proceedings of the 13th colloquium on automata, languages and programming* 226:344-358

20. Tel (1994) 'Introduction to Distributed Algorithms' *Cambridge University Press*

A *Pseudocode*

To illustrate the informal description of the algorithm, we provide a Pseudocode description of the simplified version of the BTTF Wave algorithm. This description is not intended as the basis for an implementation of the algorithm, but rather to illuminate the reader about parts of the algorithm that might be unclear.

As a reminder, there are a number of events that can happen in the basic computation. These are:

- An active process can send a message.
- An active process can receive a message.
- An active process can turn passive.
- A passive process can receive a message, which turns the process active.

The methods in this section do not implement these events. Rather, such an event in the basic computation may trigger the execution of one of the methods listed here. The exact way in which this happens will depend on the implementation.

We will start by describing the datastructures and variables used by the entire algorithm. Next, we will describe five methods that together form a pseudocode description of the BTTF Transitory Quiescence Detection Algorithm. The last two methods describe the wave distribution part of YAWA.

Algorithm *BTTF Wave*

```
datastructure
  messageInfo     : <messageSetId, sender, numberOfTargets>
  messageChain    : list of messageInfo
  messageTree     : tree with messageInfo as its internal nodes
  messageForest   : set of messageTree

var
  state        : {active, passive}   init if p is TQRoot then active else passive;
  TQRoot       : boolean             init if p = TQRoot then T else F ;
  lastChain    : messageChain        init [ ]
  mesforest    : messageForest       init ∅
  buffer       : set of messages     init ∅
  initiator    : boolean             init if p = initiator then T else F ;
  firstChain   : messageChain        init [ ]
```

Each messageInfo object will contain information about a batch of messages that was sent simultaneously. messageSetId will contain a unique identifier for this batch of messages, sender will denote the process that sent the batch, and numberOfTargets will denote the number of different messages that were sent in this batch.

A number of messageInfo objects can be listed to form a messageChain, which identifies an apparent causal relationship between certain messages of each batch represented by the messageInfo objects in the chain. If certain messageInfo objects are listed after one another in a messageChain, this identifies that one of the messages from the batch represented by the messageInfo listed second was the last message to arrive at the process that sent the batch represented by the first messageInfo. For example, if we look at the messageChain [<P0b, P0, 5 >, <P2a, P2, 3>, <P0a, P0, 1>, <A1a, A1, 2>], then process A1 sent a batch of 2 messages identified as A1a. One of these messages arrived at process P0, after which that process sent 1 message identified by id P0a. This message reached P2, which responded by sending a batch of three messages, named P2a, to P0. After P0 received one of those three messages, P0 sent a batch of 2 messages to unknown recipients.

messageInfo objects can also be stored in messageTree objects. Such a tree represents all messages that were the result of a message sent by a process that was initially active. Each messageInfo in the tree represents a batch of messages, and will eventually receive a child for each message in that batch. During a run of the algorithm, a process storing a messageTree will eventually receive messageChain objects that it initiated, when it can be guaranteed that no further messages will be added to that chain. When this happens, the process stores this messageChain in the corresponding messageTree, making each node in the chain the child of the node before it. As a result, the entire message chain will be represented by following that particular branch of the tree. When each node in the tree has a number of children equal to the number of messages sent in the corresponding batch, it can be guaranteed that all messages that were the result of the initial batch have arrived and that no further messages will result from that batch. As a result, such a complete tree can be removed.

Finally, a messageForest of a process simply denotes all messageTree objects that were initiated by this process and that have not yet been removed. As long as there are messageTree objects in a messageForest, it can be guaranteed that there are still unsent or undelivered messages in the system.

Which brings us to the variables used by the algorithm. The variable state simply denotes the state of a process. TQRoot is a boolean that starts out true for those processes that start out active, and becomes false the first time the process becomes passive or receives a message. The variable mesforest denotes the message forest of a process. The variable buffer is used to store those messages whose sending has been delayed in order to add them to a batch. And finally, initiator is a boolean that's only true in the process that will be responsible for starting the YAWA part of the algorithm and be responsible for the detection of system wide termination.

This leaves two variables that are slightly more complicated than the others. The variable lastChain is used to store the last message chain of the Transitory Quiescence part of the algorithm received by a process. By doing so, the process can create a single batch for all messages that were sent after lastChain was received but before any other messages were received, thereby avoiding the creation of new message trees.

The variable firstChain, on the other hand, denotes the first message chain of the YAWA part of the algorithm received by a process. This variable is completely ignored by the Transitory Quiescence part of the algorithm. It is used by the YAWA algorithm both to check whether the process has already been reached by the wave, and to create new message chains when the process forwards the token belonging to the wave.

Method *Transitory Quiescence Send*

```
send(message m)
% The execution of this method is triggered when an active process p in the basic computation attempts to perform a send
event of a message m.
    if TQRoot then
        % As the process is a TQRoot, there are no message that can be considered the cause of the message m.
        % Therefore, this message will become the root of a new message tree.
        % id is the messageSetId that will be associated with this message.
        create messageInfo mesinfo = <id, p, 1>;
        create messageChain meschain = [mesinfo];
        create messageTree mestree with root mesinfo;
        mesforest += mestree;
        <m, meschain> departs over the correct outgoing channel;
    else
        % The process is not a TQRoot, so there will be a message chain of which this message will be a branch.
        % Therefore, we buffer this message until we know how many more messages will branch off from that message
chain.
        add m to buffer;
```

The method send is used when a process wants to send a message. If the process is a TQRoot, then this sending of a message will result in a new message tree. So the appropriate datastructures are created, after which the message is sent. However, when the process isn't a TQRoot, the actual sending of the message needs to be delayed, because the process may want to send further messages later, yet before it receives another message. Should this happen, then those messages need to become part of the same message tree as the message that is currently being sent, so they'll need to be sent as a single batch. In order to achieve this, the message to be sent is stored in a buffer.

Method *Transitory Quiescence Receive*

```
receive(message m)
% The execution of this method is triggered when a process p in the basic computation attempts to perform a receive event
of a message m.
% This message will be accompanied by a messageChain meschain.
    if TQRoot then
        % As a message will arrive at p, further messages sent by p will be considered caused by this message.
        % Therefore the process no longer needs to be a TQRoot.
        TQRoot = false;
        wait for a message <m, meschain> to arrive.
        lastMessageChain = meschain;
        p completes its receive statement, consuming m;
    else if state == active and not TQRoot then
        % The last message to arrive will be considered the cause of all messages sent from now on.
        % Therefore, messages belonging to the previously arrived message no longer need to be buffered.
        clearBuffer;
        wait for a message <m, meschain> to arrive.
        lastMessageChain = meschain;
        p completes its receive statement, consuming m;
    else % p was passive
        wait for a message <m, meschain> to arrive.
        lastMessageChain = meschain;
        state = active;
        p completes its receive statement, consuming m;
```

First of all, it should be noted that receive can be called before a message has actually arrived at the process. When receive is called before a message has arrived, the call will block until a message becomes available, as denoted by the line "wait for a message <m, meschain> to arrive".

As any messages that will be sent after this point will be considered to be caused by the receipt of this message, the process receiving the message no longer needs to be a TQRoot, because the defining characteristic of a TQRoot is that it sends messages that were not caused by any other messages. So if the process was a TQRoot, it sets the variable to false, correctly sets its other variables, and passes the message on.

Furthermore, there is no further need to buffer any messages for which the process executed a send event before this point, because any new messages for which the process executes a send will be added to the message tree of the newly received message. An active process that was not a TQRoot therefore starts by clearing its buffer, after which it correctly sets its variables and passes the message on to the process.

A process that was passive had already cleared its buffer when it became passive, so it just sets its variables, including setting the state of the process to active, and passes the message on.

Method *Transitory Quiescence Turn Passive*

```
turnPassive
% The execution of this method is triggered when a previously active process p in the basic computation turns passive.
    if TQRoot then
        % Any further messages this process will send will be considered caused by the message this process receives when it turns active again.
        % Therefore, this process will no longer be capable of creating messages that were not caused by any other message.
        TQRoot = false;
    else
        % This was an active process, but not a TQRoot, so it previously received a message.
        % As no more messages will be sent by this process before a message is received, buffering is no longer necessary.
        clearBuffer;
        lastMessageChain = [ ];
    state = passive;
```

A process that turns passive won't send any further messages until it receives a message. Since the defining characteristic of a TQRoot is that it sends messages that were not caused by any other messages, there is no need for such a process to be an TQRoot anymore. So a process that was a TQRoot simply sets TQRoot to false, and then sets its state to passive.

A process that was not a TQRoot, on the other hand, was active, so it must have previously received a message. Since no further messages will be sent until the receipt of the next message, there is no more need to buffer messages. Therefore, this process will clear its buffer, reset its lastMessageChain to denote that there is no message chain to add to anymore, and finally will set its state to passive.

Method *Transitory Quiescence Clear Buffer*

```
clearBuffer
% This is an auxiliary method. It's execution is not triggered by particular events; rather, this method is called by the methods receive and turnPassive.
% This method either guarantees the departure of all messages buffered by process p, or sends a confirmation message to the root of the message tree containing lastMessageChain
    if buffer == $\emptyset$ then
    % No more messages will be added to this chain
        <CONFIRMATION, lastMessageChain> departs over the outgoing channel to the sender of the original messageInfo in lastMessageChain;
    else
        % A number of messages that had been buffered have to depart.
        % Note that each of these messages will have the same messageSetId id.
        int nummes = number of messages in buffer;
        create messageInfo mesinfo = <id, p, nummes>;
        messageChain meschain = lastMessageChain + mesinfo;
        **for** each message m in buffer **do**
            <m, meschain> departs over the correct outgoing channel;
```

The method clearBuffer either sends all messages that were stored in the buffer, or, if there were no messages stored, it confirms the receipt of the last message the process received (and implicitly all messages received in the message chain before that), by sending lastMessageChain to the process that stored the message tree belonging to that chain.

CONFIRMATION denotes a unique value for a message to tell the receiving process that this is a control message rather than a message belonging to the computation.

A process sending further messages will add a new node to each message chain accompanying these messages, to represent the batch of messages that is currently being sent. Each of these nodes will contain the same id, as a messageSetId represents a batch of messages rather than a single message.

Method *Transitory Quiescence Receive Signal*

```
**receiveSignal**(∅ , messageChain meschain)
% The execution of this method is triggered when a pair <CONFIRMATION, meschain> arrives at the root of a message tree.
  identify the messageTree mestree that has as root the first element of meschain;
  follow the branches of mestree that match each of the messageSetIds in meschain;
  **if** a certain messageSetId id in meschain can't be found in mestree **then**
      % Let us call the last messageSetId that was found in mestree lastid.
      % id represents the set of siblings caused by one of the messages of lastid.
      % Furthermore, this is the first time the arrival of one of the messages from this set of siblings has been confirmed.
      % This means that we are now aware of an entire new branch of this message tree.
      add a new branch to the tree, starting at lastid, with each child being the next messageSetId in meschain;
      % Since the root of the tree received meschain, we know that at least one of the children of the last messageSetId in the chain arrived.
      add an external node "complete" as a child of the last messageSetId in meschain;
  **else** % We were fully aware of the existence of each messageSetId in meschain.
      % The last message in the chain was received and did not generate new messages
      add an external node "complete" as a child of the last messageSetId in meschain;
  **if** each node of mestree has a number of children equal to the number of targets in the messageInfo of the node **then**
      % The receipt of all messages in this tree has been confirmed.
      remove mestree from mesforest;
```

The method receiveSignal handles a message chain that has been sent back to the TQRoot that is responsible for the corresponding message tree. This message chain needs to be added to the appropriate message tree, further filling out the branches of that tree and potentially causing that tree to become removed.

The first task of the process is to identify to what extent the message chain was already represented in the message tree. This involves following the tree from the root downwards, until a node is reached where none of the children matches the next id in the message chain, or until the entire message tree is matched.

If a certain id can't be found in the tree, then the entire message chain from that point onwards was thus far unknown to the process, and needs to be added to the tree. This is done by creating as a child of the last message info that was found the first message info that wasn't found. Each following node is then given as a single child the next message info in the chain. After this has been done, there will be one message info that does not yet have a child: the last message info in the chain. This message info represents a batch of messages, at least one of which arrived, because the only possible way for a message chain to be sent back is by calling clear buffer after that chain has been received but without messages in the buffer. Therefore, this last node is given a single child, "complete", denoting that at least one of the messages of that batch arrived. This same child "complete" is also created when the entire message chain could be traced in the tree, as the receipt of the message chain confirms the receipt of at least one message.

Then, if each node in the tree has a number of children equal to the number of targets in the corresponding message info, we know that all messages that were spawned as a result of this message tree have arrived, so the entire tree can be removed from its message forest.

Method *YAWA Non Initiator Becomes Empty*

```
nonInitiatorBecomesEmpty
% The execution of this method is triggered in one of two ways.
% It is executed either when the last message tree in the message forest of a process that is not a TQRoot is removed,
% or when a process that has an empty message forest stops being a TQRoot.
    if initiator then
        % The YAWA algorithm can be started by sending a token to all neighbors of p.
        create messageInfo mesinfo = <id, p, numberOfNeighbors>;
        create messageChain meschain = [mesinfo];
        create messageTree mestree, with root mesinfo;
        for each neighbor of p do
            send <token, meschain> over the correct outgoing channel;
    else
        if firstMessageChain ≠ [ ] then
            % We received a token before,
            % so we forward a token to all neighboring processes;
            create messageInfo mesinfo = <id, p, numberOfNeighbors>;
            messageChain meschain = firstMessageChain + mesinfo;
            for each neighbor of p do
                send <token, meschain> over the correct outgoing channel;
```

When the initiator for the YAWA algorithm has called nonInitiatorBecomesEmpty and the message forest that has been created as a result becomes empty, termination can be concluded.

YAWA creates a wave that visits each process, and only moves on when that process is not a TQRoot and its message forest is empty. Since a message tree is created in the process that starts that wave, this message tree doesn't become complete until after all processes are not TQRoots and have an empty message forest.

nonInitiatorBecomesEmpty specifies what should be done when the conditions for forwarding the wave become true. The process that is the initiator for YAWA can start the wave when it is no longer a TQRoot and its message forest is empty. This is done in the first if statement, that creates a message tree and sends a token to all neighbors.

A process that is not an initiator for YAWA can start forwarding the token to all its neighbors when the conditions for forwarding become true. Of course, this assumes that at least one token has reached the process, which is checked by the firstMessageChain ≠ [ ] if statement.

Method *YAWA Receive Token*

```
receiveToken(token, messageChain meschain)
% The execution of this method is triggered when the process receives a token.
    if firstMessageChain $\neq$ [ ] or initiator then
        % The process either already forwarded the token, or it will do so at a later time.
        send meschain to sender of first message in chain;
    else
        % This is the first time a token arrives at this process.
        firstMessageChain = meschain;
        if mesforest == $\emptyset$ and not TQRoot then
            % This token will create no further message trees,
            % so we forward a token to all neighbor processes;
            create messageInfo mesinfo = <id, p, numberOfNeighbors>;
            messageChain meschain = firstMessageChain + mesinfo;
            for each neighbor of p do
                send <token, meschain> over the correct outgoing channel;
```

When a process receives a token belonging to the YAWA part of the algorithm, there are a number of possibilities. The first is that the process received a token before. In that case, that previously received token will be used to build the message chains of the tokens that will be forwarded, so the token can be sent back to the initiator for YAWA, which is the first process listed in the message chain.

If the process did not receive a token before, then any forwarded tokens should be added to the message chain of this token. To mark that this is the first token received, it is stored in firstMessageChain. If the requirements for forwarding tokens are met, the token is immediately forwarded. Otherwise, it will be forwarded when this process calls nonInitiatorBecomesEmpty.