ACTIVE AND PASSIVE SYNCHRONIZERS

Lior Shabtay and Adrian Segall
Dept. of Computer Science
Technion, Israel Institute of Technology
Haifa, Israel 32000
email: liors@cs.technion.ac.il, segall@cs.technion.ac.il

December 17, 1991

Abstract

Correctness of synchronizers is examined: a synchronizer is said to be correct, if each execution of
a protocol created by combining it with a synchronous protocol, is equivalent to some execution of the
original (synchronous) protocol. Two families of synchronizers are introduced, passive synchronizers and
active synchronizers. Passive synchronizers do not change the original synchronous protocol and only add
a synchronization protocol for synchronizing it, while active synchronizers may change the original protocol
code, messages, etc.

The timely property, which is sufficient and necessary for correctness of passive synchronizers, and the
matching property, which is sufficient and necessary for correctness of active synchronizers, are introduced.
Lower bounds are set for communication and time complexity of correct passive synchronizers, thus showing
that active synchronizers are necessary for low complexities, while retaining the correctness of the synchronizers.

Two types of active synchronizers are introduced: message-monitoring synchronizers, in which synchronizer code is performed as a result of original-protocol message reception and transmission, and variable-duplicating synchronizers, which duplicate all original-protocol variables at each node. A message-monitoring correct synchronizer with low communication and time complexities is presented. A method is introduced for creating variable-duplicating correct synchronizers from passive incorrect synchronizers, with identical communication and time complexities.

One more family of active synchronizers is examined: message-delaying synchronizers. In message-delaying synchronizers, early messages are saved and treated later. We show that implementation of such synchronizers, using certain passive synchronizers, is not straightforward, since in these cases, nodes may not always know which messages are early and which are timely. Solutions for this problem are suggested.
1 Introduction

In this paper we are dealing with distributed protocols in two network models: the synchronous model and the asynchronous model. In the asynchronous model, messages sent by a node to any of its neighbors are received by that neighbor in a FIFO order within a finite undetermined time and code can be performed by nodes only upon receiving a message. In the synchronous model, all link delays are bounded by some quantity $T$. The network contains a global clock that ticks synchronously at all network nodes at time intervals $T$. Code can be performed upon receipt of a message and/or at clock ticks. Nodes are allowed to send messages only at clock ticks.

Synchronizers are tools for transforming protocols written for a synchronous model to protocols that run on an asynchronous model. The synchronous protocol will be referred to as the original protocol, its messages as the original-protocol messages and its code as the original-protocol code. The original-protocol code is composed of the original-protocol pulse code, which is the part performed at pulses of the synchronous model and the original-protocol message code which is the part performed in the synchronous protocol upon receipt of a message.

The asynchronous protocol created by the synchronizer generates a sequence of 'clock-pulses' at each node of the network, that occur asynchronously at different nodes. The node performs at each pulse the original-protocol pulse code (or a code close to it, see Sec. 3) and sends messages that are identical or almost identical to the original-protocol messages.

Thus, in the created asynchronous protocol we have in general two types of messages and it is important to distinguish between the two: original-protocol messages and synchronizer messages. The latter type are messages introduced by the synchronizer to ensure its proper operation.

The methodology of synchronizers was introduced in [Awer 1], where three synchronizers were presented: the $\alpha$ synchronizer, with overhead of $O(|E|)$ in communication complexity and $O(1)$ in time complexity per pulse, the $\beta$ synchronizer with overhead of $O(|V|)$ in communication and $O(D)$ in time complexity per pulse (when $D$ is the diameter of the network), and the $\gamma$ synchronizer, that enables trade-off between the above complexities. Other types of synchronizers, designed for specific asynchronous models, specific topologies or specific tasks, can be found in [PU 1], [LT 1], [CCGZ 1], [ER 1], [ER 2] and [AS 1].

Applications of synchronizers can be found in [Awer 2], [AP 1], [SM 1] and [LTC 1]. All these applications are new asynchronous protocols created by combining synchronizers with known synchronous protocols that perform a specific task, thus creating asynchronous protocols which are better in terms of communication and time complexity than any other known asynchronous protocol that performs the same task.

Automatic conversion of protocols from the synchronous model to the asynchronous model is another important application of synchronizers. Automatic conversion is especially useful because most of the distributed computer systems and networks comply with the asynchronous model, while software and algorithm development for the synchronous model is much simpler. Thus, a significant amount of effort can be saved when the software is developed in the synchronous model.
and converted automatically for running on the real asynchronous hardware.

All synchronizers in [Awer 1], [PU 1] and [AP 1] have a common feature: a node may perform a new pulse when it knows that it had received all original-protocol messages sent to it by its neighbors at the former pulse. However, it is shown in [LT 1] that this feature is in fact not sufficient and thus the mentioned synchronizers do not ensure proper operation of the created asynchronous protocol. The problem is that the synchronizers allow original-protocol messages sent by a node at a given pulse to arrive at a neighbor node before the time when the latter has performed that same pulse. We refer to such messages as early messages. Obviously, such a situation cannot occur in a synchronous model.

Sometimes, this problem can be fixed for a specific synchronous protocol by changing the asynchronous protocol created by the synchronizer in such a way that it will still work correctly. This cannot be done when using automatic synchronizers, since their outcome should not be further processed by hand.

One way to solve this problem is suggested in [LT 1], [FLS 1], [ER 1], [Awer 1] and will be referred to as message delaying. The idea is that when a message is received too early at a node, it is not processed immediately. Instead, such messages are saved and processed only after the pulse is performed at the node. The message delaying method has two main advantages: It is very intuitive and very simple to implement. However, as explained below, message delaying is not implementable in some cases, and in many others, it requires a very high price in terms of memory, time, etc. For example:

1. In some algorithms, messages are long. In some extreme cases, like some connectivity-test and minimum-hop protocols [Seg 1], each message might be as large as the local memory of the nodes. Therefore, saving messages in buffers may require the size of memory multiplied by a factor proportional to the maximum number of messages that need to be saved in buffers at the same time (for more accurate analysis of memory demands, see Sec. 2.2).

2. In some systems, the number of buffers used by the data-link layer is limited, so if messages are to be saved for long periods of time, they should be copied to other buffers. This happens frequently when the data-link layer and the application are implemented on distinct processors that share a small amount of memory. The operation of copying messages costs an extra time that can be avoided if messages are processed immediately.

3. Some supercomputers consist of a very large number of processors, where each processor is limited in terms of memory and processing power, and thus, no memory is available for saving messages.

4. In some cases, the networked system is implemented in hardware, where each node is a VLSI finite-state-machine. Implementing message-saving capabilities, means more memory cells and many more machine states.

Another problem with this method is that in order to implement it, the nodes have to know at which pulse every received original-protocol message was sent. As shown in Sec. 8, this is not
possible in many synchronizers. We also present in Sec. 8 some solutions to this problem.

Message-delaying is one way to solve the problem of early messages. Another way is suggested in [LT 1]. This solution is a new synchronizer that ensures that an original-protocol message sent by a node at a given pulse arrives at the respective neighbor only after the latter has performed that pulse. The problem with this synchronizer is its bad performance in terms of time and communication overhead.

In the present paper we introduce several synchronizers that ensure proper operation without message-delaying, and with communication and time complexities that are similar to the complexities of $\alpha$, $\beta$ and $\gamma$ proposed in [Awer 1]. The two main methods we are using for this purpose are called message-monitoring and variable-duplicating. Message-monitoring synchronizers are discussed in Sec. 6 and variable-duplicating are discussed in Sec. 7.

Message-monitoring and variable-duplicating synchronizers do not satisfy the above mentioned properties, i.e. that a message sent at a given pulse is received between the time of the same pulse and of the next one at the neighbor. Instead, we suggest a less demanding set of properties that still ensures proper operation. The main idea behind this new set of properties is to require the synchronizer to send at each pulse, original-protocol messages identical to the messages sent by the node at that pulse in the original protocol. Moreover, we require that the local state at the node at the end of the execution will be identical to the local state of the node at the end of the execution of the original protocol. The synchronizers suggested in this paper ensure these properties not only by controlling the timing of the pulses at each node, as done in most previous synchronizers, but also by changing the code of the original-protocol, by allowing the synchronizer to use the original-protocol variables and, in some cases, by changing the format of the original-protocol messages. We refer to such synchronizers as 'active', while synchronizers that leave original-protocol code and messages unchanged and do not use the original-protocol variables, are referred to as 'passive'.

In the rest of this paper we proceed as follows: In Sec. 2 we describe in detail the synchronous and the asynchronous models. In Sec. 3 we define synchronizers precisely. Sec. 4.1 defines passive synchronizers and active synchronizers. Sec. 4.2 contains a discussion of the problems of known synchronizers. We also introduce in this section the set of properties that ensure proper operation of passive synchronizers. Sec. 4.3 contains a proof of the sufficiency and necessity of these properties. In Sec. 4.4 we revisit the passive synchronizer of [LT 1], referred to as $\delta$, that ensures proper operation. In Sec. 4.5 we present $\delta_1$, a slight improvement for the $\delta$ synchronizer. In Sec. 4.6 we present lower bounds for the communication and time complexity of passive synchronizers that ensure proper operation and show that $\delta_1$ is optimal in both senses.

Since the optimal passive synchronizer that ensures proper operation still has high communication and time costs, it is reasonable to seek better performance in the set of active synchronizers. The set of properties that ensure proper operation of active synchronizers is presented and proved in Sec. 5.

In Sec. 6 we introduce one family of active synchronizers that allow better communication complexity — the message-monitoring synchronizers. In this family of synchronizers, synchronizer-
code is triggered by incoming original-protocol message. The message-monitoring synchronizers we present in this section are $\epsilon_1$, $\epsilon_2$ and $\epsilon_3$. In Sec. 6.1 we present $\epsilon_1$, a message-monitoring synchronizer that ensures proper operation without message delaying. This synchronizer has $O(|V|)$ communication complexity and $O(D)$ time complexity, the same as $\beta$. Sec. 6.2 presents the $\epsilon_2$ and $\epsilon_3$ synchronizers which are variations of $\epsilon_1$.

In Sec. 7 we introduce another family of active synchronizers — the variable-duplicating synchronizers. These synchronizers allow better communication and time complexities by using a technique of using a backup copy of all the original-protocol variables and a copy of the original-protocol pulse code changed to reference these backup variables. The variable-duplicating synchronizers we present in this section are $\alpha_1'$ and $\gamma_1'$. In Sec. 7.1 we present the $\alpha_1'$ synchronizer that ensures proper operation without delaying messages, in $O(1)$ time and $O(|E|)$ communication complexity (the same as $\alpha$). The method used to derive $\alpha_1'$ from the $\alpha$ synchronizer is generalized and proved in Sec. 7.1, and this method is used to create synchronizer $\gamma_1'$ from synchronizer $\gamma$ in Sec. 7.2.

Message delaying is discussed in Sec. 8. In Sec. 8.1 we present and prove the $\alpha^*$ synchronizer, which is the message-delaying version of $\alpha$. The problems that arise when trying to implement a message delaying version of $\beta$ are discussed in Sec. 8.2 and a way to solve these problems is presented in synchronizer $\beta_1^*$, described also in that section. This method is generalized for other synchronizers and proved in Sec. 8.3. Synchronizers $\beta_2^*$ and $\beta_3^*$, which are versions of $\beta_1^*$, are described in Sec. 8.4. In Sec. 8.5 we describe synchronizers $\gamma_1^*$, $\gamma_2^*$ and $\gamma_3^*$, which are message delaying versions of $\gamma$. A proof that all message delaying synchronizers ensure proper operation is given in Sec. 8.6.
2 The models

For our purposes, a network can be described by an undirected graph \((V,E)\), where \(V\) is the set of graph nodes and \(E\) is the set of graph edges. Every node represents a processor in the network, and every edge represents a bidirectional communication channel between its two end nodes.

**Definition 2.1 \((G_i)\)**

For a node \(i \in V\), we define \(G_i\) as the set of the neighbors of \(i\) in the network graph.

**Definition 2.2 \((t-,t+)\)**

For a given time \(t\), we denote by \(t-\) and \(t+\) the time just before and just after \(t\) respectively.

2.1 The Synchronous Model

The synchronous model assumes that all link delays are bounded by some quantity \(T\). Pulses are generated synchronously at all nodes in the network at intervals \(T\). Messages are sent only at pulse ticks, and thus arrive at the destination node before the next pulse.

Each node is awakened at each pulse and when receiving a message from some neighbor. At that time, the node processes the message and performs local computations. At each pulse each node may send, if needed, messages to some or all of its neighbors. All local actions are performed atomically.

At each node, the algorithm is built of separate code-blocks, each associated with the receipt of a given type of message. When a message is received by the node, the code-block associated with that type of message is executed. The code-blocks associated with received messages contain only local computations; no messages are sent upon receipt of a message. One additional code-block is introduced, to be executed upon occurrence of a pulse. This block may contain local computations, as well as commands for sending messages.

Messages are processed at each node in the received order, even if they were received from different lines. No other processing order is permitted; in particular, a node is not allowed to save a received message, to be processed after messages received later.

Notice that this synchronous model is event driven, meaning that code is performed when receiving a message, as well as when a pulse occurs. Another common definition of a synchronous model is the pulse driven model. In the pulse driven synchronous model, messages received are held in buffers and do not cause execution of code until the next pulse. When the next pulse occurs, the code performed at the pulse may access these buffers and process them.

In this paper, we prefer the event driven model although the pulse driven is a bit more convenient for the programmer. The reason for preferring the event driven model is that it is, in many cases, much more efficient in terms of memory usage: no buffers need to be allocated in order to save messages. Another reason for preferring the event driven model, is that this model is used also in previous works on synchronizers, like [Awer 1], [Awer 2], [LT 1], [PU 1], [AP 1], [LTC 1].
Definition 2.3 \((\text{pulse}(n))\)

In a synchronous network, we define \(\text{pulse}(n)\), \(n = 0, 1, \ldots\) as the time of the \(n\)-th pulse. Messages may be sent by the nodes only at times \(\text{pulse}(n)\), \(n = 0, 1, \ldots\)

Definition 2.4 \((\text{phase}(n))\)

In a synchronous network, \(\text{phase}(n)\) is defined as the interval \([\text{pulse}(n), \text{pulse}(n+1)]\).

Assumption 2.1 In a synchronous network, at time \(\text{pulse}(n)\), node \(i\) sends at most one message to every neighbor.

Assumption 2.1 has been used in previous works on synchronizers ([Awer 1], [Awer 2], [FLS 1], [LT 1], [PU 1], [ER 1], [ER 2], [CCGZ 1], [AP 1], [AS 1]), and is reasonable because multiple sent messages can be simulated by packing them into one message. This is because the properties of synchronous protocols must hold for all possible legal executions and one such execution is for nodes to consecutively receive all messages from each neighbor sent at a given pulse, with no in between arrival of messages from other neighbors. Thus, the assumption does not affect generality.

Definition 2.5 \((\text{local state}, \Pi_i(t))\)

For a synchronous-protocol execution, the local state \(\Pi_i(t)\) of a node \(i\) at a given time \(t\), is defined as the collection of all local variables at node \(i\) at time \(t\). \(\Pi_i[\mathcal{X}](t)\) denotes the local state of node \(i\) at time \(t\) in the execution \(X\).

Definition 2.6 \((\text{partial execution})\)

A partial execution of a synchronous-protocol \(P\) on a network \(N\) is defined as the part of some execution of \(P\) on \(N\) that takes place during the time interval \([\text{pulse}(0), \text{pulse}(n)]\) for some \(n \geq 0\).

The message sent by a node \(i\) to a neighbor \(k\) at time \(\text{pulse}(n)\) for some \(n \geq 0\) and the state of \(i\) at time \(\text{pulse}(n)+\), are functions of the local state of the node at time \(\text{pulse}(n)-\). These functions, denoted by \(\text{msg}^k_i\) and \(\text{stt}_i\) respectively, are derived from the code-block associated with pulses.

Definition 2.7 \((\text{msg}^k_i)\)

Given a synchronous-protocol, \(\text{msg}^k_i\) is defined as the function that returns the contents of the message sent by \(i\) to its neighbor \(k\) at \(\text{pulse}(n)\), given the local state of node \(i\) at time \(\text{pulse}(n)-\).

Definition 2.8 \((\text{stt}_i)\)

Given a synchronous-protocol, \(\text{stt}_i\) is defined as the function that returns the local state of \(i\) at time \(\text{pulse}(n)+\), given the local state of node \(i\) at time \(\text{pulse}(n)-\).

A synchronous-protocol execution terminates when no more messages are sent and the local states of the nodes stop changing. In other words, \(\text{pulse}(n)\) is the last pulse of a protocol execution if \(\text{msg}^k_i(\Pi_i(\text{pulse}(n)-)) = \emptyset\) and \(\text{stt}_i(\Pi_i(\text{pulse}(n)-)) = \Pi_i(\text{pulse}(n)-)\) hold for all \(i \in V, k \in G_i\).
2.2 The Asynchronous Model

In the asynchronous model, each node is awakened by receiving a message from some neighbor or from the outside world. At that time, the node processes the message, performs local computations, and if needed, sends messages to some or all of its neighbors. All local actions are performed atomically. The messages on a given link are transferred by a reliable DLC protocol, which means that they arrive in finite undetermined time, in FIFO order, and the sender DLC receives an acknowledgment within a finite undetermined time for every message received at the other end.

In the asynchronous model, the algorithm at each node is event driven. This means that the algorithm is built of separate code-blocks, each associated with the receipt of a given type of message. When a message is received by the node, the code-block associated with that type of message is executed. Each code-block may contain local computations, as well as commands for sending messages to neighbors. The contents of the received message is handed on to the code-block, together with the name of the link on which the message had been received.

In this report, we shall consider two versions of the asynchronous model. In the non-delaying version, similarly to the synchronous model, messages are processed at each node in the received order, even if they were received from different lines. No other processing order is permitted; in particular, a node is not allowed to save a received message, to be processed after messages received later. In the delaying version, saving and late processing of messages are allowed.

In the sequel, when analyzing the memory demands of a protocol at each node, we do not count the memory used by the DLC for buffering received and transmitted messages. The reason is that we assume that the DLC task is performed by a smart card that has its own memory (as done in most of the systems, including, for example, IBM-PC in all its versions). It is also reasonable to assume that the memory used in this card is much cheaper, since it does not have to be as fast as the memory accessed by the CPU. Another reason for not counting the memory used by the DLC is that all the protocols we present need the same DLC resources as the synchronizers presented in previous works.

3 Synchronizers

Roughly speaking, a synchronizer is a transformation that transforms a synchronous protocol to an asynchronous one by generating a sequence of 'clock-pulses' at each node of the network, where the pulses occur asynchronously at different nodes. A new phase of the protocol starts at each node at each clock pulse.

In the created asynchronous protocol, we denote the series of pulses at node $i$ by $t_i(0), t_i(1), \ldots$. Original-protocol messages are allowed to be sent by $i$ only at times $\{t_i(n), n \geq 0\}$.

The synchronous protocol will be referred to as the original protocol, its messages as the original-protocol messages and its code as the original-protocol code. The original-protocol code is composed of the original-protocol pulse code, which is the part performed at pulses of the synchronous model.
and the original-protocol message code which is the part performed in the synchronous protocol upon receipt of a message.

In the created asynchronous protocol we have in general two types of messages and it is important to distinguish between the two: original-protocol messages and synchronizer messages. The latter type are messages introduced by the synchronizer to ensure its proper operation.

Here we propose a rigorous definition for synchronizers: The task of combining the synchronous protocol with the synchronizer is done by a transformation that transforms the source code of the original (synchronous) protocol into a code of a new asynchronous protocol.

Definition 3.1 (synchronizer)
A synchronizer is defined as a transformation that transforms the source code of a synchronous protocol into a code of an asynchronous protocol, with the following properties:

(i) It leaves the original variables at the node unchanged.

(ii) It leaves unchanged the original-protocol messages, except possibly, an addition of a prefix or a suffix.

(iii) It may add variables and messages of its own, called synchronizer variables and synchronizer messages respectively. It must add a code-block for each new message type, associated with receipt of that type of message.

(iv) The resulting asynchronous protocol generates a sequence of clock-pulses $t_i(0), t_i(1), \ldots$ at each node $i$, with the property that original-protocol messages may be sent by node $i$ only at times $\{t_i(n), n \geq 0\}$.

(v) It may use additional code-blocks, that are triggered by data-link events (for example, some synchronizers include a code-block that is performed when all original-protocol messages sent at a given time have been acknowledged).

Definition 3.2 (combined protocol)
A synchronous protocol combined with a synchronizer, or in short a combined protocol, is the asynchronous protocol created by the synchronizer when given the original (synchronous) protocol as input.

Definition 3.3 (ensure ... for a protocol)
A synchronizer ensures a property for a synchronous protocol $P$, if every execution of $P$ combined with this synchronizer satisfies this property.

Definition 3.4 (ensure)
A synchronizer ensures a property, if it ensures this property for every synchronous protocol.
4 Passive Synchronizers

4.1 Passive and Active Synchronizers

We shall classify synchronizers into two main types: roughly speaking, passive synchronizers leave the original-protocol messages and the original-protocol code unchanged, while active synchronizers may alter either or both. The synchronizers described in [Awer 1], [LT 1], [PU 1] are passive synchronizers. The definition of passive synchronizers is given below.

Recall that the code of the original (synchronous) protocol consists of code-blocks associated with received original messages, and one additional code-block associated with pulses.

Definition 4.1 (passive synchronizer)

A passive synchronizer is a synchronizer transformation with the following additional properties:

(i) It leaves the original-protocol messages unchanged.

(ii) It leaves the code-blocks associated with received original-protocol messages unchanged.

(iii) It places the code-block associated with pulses in a procedure, called PulseCode; at node \( i \), this procedure is called at times \( t_i(n) \), \( n = 0, 1, 2 \ldots \) Except for that, the code associated with pulses is left unchanged.

(iv) The code-blocks associated with synchronizer messages may contain only computations on synchronizer variables and may send only synchronizer messages.

(v) The code inserted by the synchronizer transformation is independent of the original protocol.

An active synchronizer may create synchronizer code to be performed when original-protocol messages are received by the node, may save received original-protocol messages and process them only after having received more messages, may add a suffix or a prefix to the original-protocol messages, may add code that references and changes the variables of the original-protocol, may change the original-protocol code, etc.

4.2 The problem

According to [Awer 1] and [Awer 2], every synchronizer must ensure the following property:

a) The original-protocol message sent in the combined protocol by \( i \) to neighbor \( k \) at time \( t_i(n) \) arrives at node \( k \) before \( t_k(n + 1) \).

It was shown in [LT 1] that for a simple passive synchronizer and a special version of the BFS protocol, this property is not sufficient to ensure proper operation of the combined (asynchronous) protocol. In this section we show that in fact for any passive synchronizer, there is an original
protocol such that property (a) above is not sufficient for the proper operation of the combined (asynchronous) protocol.

The synchronous model has at least one additional property that is not reflected in property (a). This property is the fact that the message sent to node $k$ by a neighbor $i$ at time $\text{pulse}(n)$ never arrives at $k$ before $\text{pulse}(n)$. In the synchronous model this is obvious, but when translating it to the synchronizer language, it turns out to be nontrivial:

b) The original-protocol message sent in the combined protocol by $i$ to neighbor $k$ at time $t_i(n)$ arrives at $k$ after $t_k(n)$.

Definition 4.2 \textbf{(timely)}

A synchronizer that, combined with any synchronous protocol, ensures properties (a) and (b), is said to be timely.

In [LT 1], it is shown that with the simple synchronizer given there, original-protocol messages in the combined protocol may arrive before their time. It is claimed (without proof) in [LT 1] that the $\alpha$, $\beta$ and $\gamma$ synchronizers presented in [Awer 1] suffer from the same problem.

In this section it is convenient to use a slightly different BFS version. We prove that when it is executed on an asynchronous network combined with any passive synchronizer that does not ensure the timely property for this BFS version, it does not work properly. In addition, we provide here a proof that $\alpha$, $\beta$ and $\gamma$ are not timely, since they do not ensure (b).

The BFS version here is based on the BFS protocol presented in [Awer 2]. This synchronous protocol constructs the minimum-hop distance from each node to some node $s$ for a directed graph. The protocol is executed on a network where each node of the input graph is represented by a processor, and a bidirectional communication link between two processors exists if there is an edge in any direction between them in the directed graph. The protocol creates layers in the network, where each layer is a subset of the nodes in the network; $\text{layer}(n)$ is defined as the set of all nodes at hop-distance $n$ from $s$. The protocol uses one type of messages, $\text{LAYER}(n)$. If a node belongs to $\text{layer}(n)$, it sends a $\text{LAYER}(n)$ message at time $\text{pulse}(n)$ to all its neighbors. Therefore, the protocol works as follows: at time $\text{pulse}(0)$, $s$ sends a $\text{LAYER}(0)$ message to all its neighbors. Each node that receives a $\text{LAYER}(0)$ message knows that it belongs to $\text{layer}(1)$ and sends a $\text{LAYER}(1)$ message to all neighbors at time $\text{pulse}(1)$. Each node that is not in $\text{layer}(0)$ or $\text{layer}(1)$ and receives a $\text{LAYER}(1)$ message knows that it belongs to $\text{layer}(2)$ and sends a $\text{LAYER}(2)$ message at time $\text{pulse}(2)$, and so on.

We shall introduce a slight change that improves the efficiency of the synchronous protocol communication. Instead of including the layer of the sending node in the message, the receiving node will extract this information from the phase during which it receives the first message. The code of the new protocol, referred to as new-BFS, can be found in Table 1.
Protocol new BFS

Messages

MSG

Variables

\( pulse_i \): The number of the present pulse. (\( pulse_i = 0, 1, \ldots, |V| - 1 \).)
\( G_i \): The set of neighbors of node \( i \) in the network.
\( InEdges_i \): List, containing those neighbors of node \( i \) such that \( (j \rightarrow i) \in \bar{E} \).
\( OutEdges_i \): List, containing those neighbors of node \( i \) such that \( (i \rightarrow j) \in \bar{E} \).
\( layer_i \): The estimate of \( i \) about its distance from \( s \) along the shortest path.
\( father_i \): The father of \( i \) in the Breadth-First-Search Tree.
\( layer_i(j) \): The estimate of \( i \) about \( layer_j \).

Initialization

\( pulse_i = 0 \).
\( layer_i = \infty \) for all \( i \neq s \).
\( layer_s = 0 \).
\( father_i = nil \) for all \( i \neq s \).
\( father_s = s \).
\( layer_i(j) = \infty \) for all \( j \in G_i \).

Algorithm for node \( i \)

When receiving MSG from neighbor \( j \) do

\( layer_i(j) \leftarrow pulse_i \)

if \( father_i = nil \) 

then

\( layer_i \leftarrow pulse_i + 1 \)

\( father_i \leftarrow j \)

At new pulse do

\( pulse_i \leftarrow pulse_i + 1 \)

if \( pulse_i = layer_i \)

then

send MSG to all \( j \in OutEdges_i \)

Table 1: The Altered BFS
new-BFS protocol. Assume that the first message that violates (b) is sent by node $j$ to node $i$ at $t_j(n)$. Since this is the first message that violates (b), the local state of $j$ at $t_j(n)$ is the same as the state of $j$ at pulse$(n)$ in some synchronous new-BFS execution (this is proved in the next section). Thus, \texttt{layer}_j$ holds the correct minimum-hop distance from $j$ to $s$, which is $n$ because $j$ sends a message at $t_j(n)$. The violation of (b) means that this message arrives at node $i$ before $t_i(n)$, when the phase at node $i$ is $(n - 1)$ or less. In this case, $i$ sets \texttt{layer}_i(j)$ to be $(n - 1)$ or less, which is wrong. Therefore, if $i$ is in layer $(n + 1)$, it will also set \texttt{layer}_i$ to be $n$ or less, which is even worse (but the wrong setting of \texttt{layer}_i(j)$ is sufficient for our argument). Therefore, any passive synchronizer that does not ensure part (b) of the \texttt{timely} property, does not ensure proper operation.

Synchronizers $\alpha, \beta$ and $\gamma$ in [Awer 1] are passive synchronizers and thus they do not ensure proper operation unless they ensure (b). Here we show that they do not.

In synchronizer $\alpha$ [Awer 1], a node $i$ is said to be 'safe' with respect to $t_i(n)$, if all messages of the original protocol sent by $i$ at $t_i(n)$ have already been received by the respective neighbor. When a node learns that it is safe, it sends a synchronizer message to all its neighbors. Node $i$ generates $t_i(n + 1)$ when it learns that all its neighbors are safe. In order to show that $\alpha$ does not ensure part (b) of the \texttt{timely} property, suppose that $i, j, k$ and $l$ in the network of Fig. 1 have already sent the messages of $t_i(n), t_j(n), t_k(n), t_l(n)$ respectively. Nodes $i, k$ and $l$ learn that they are safe for this pulse, and send synchronizer SAFE messages to their neighbors. Suppose that the message from $j$ to $i$ is slow. Node $k$ receives the safe messages from $l$ and $i$, and generates $t_k(n + 1)$, sending an original-protocol message to $i$. This message can arrive at node $i$ before the one sent by $j$ at $t_j(n)$, contradicting (b).

In synchronizer $\beta$ [Awer 1], an initialization phase creates a directed tree, with some root $s$. Each node, when recognizing that it and all its descendents in the tree are safe, informs its father.
When \( s \) finds out that all nodes in the network are safe, it broadcasts an AWAKE message on the tree. This message tells the nodes to perform the next pulse. To show that synchronizer \( \beta \) does not ensure part (b) of the timely property, consider the network in Fig. 2(i). The spanning tree selected by the initialization phase is shown in Fig. 2(ii). After performing phase(\( n \)), nodes \( s, i \) and \( j \) have discovered that they are safe. Nodes \( i \) and \( j \) have informed \( s \), so \( s \) generates \( t_s(n + 1) \) and sends an AWAKE message to \( i \) and \( j \). The message is received by \( j \), causing it to generate \( t_j(n + 1) \) and to send an original-protocol message to \( i \). This message can arrive at node \( i \) before \( t_i(n + 1) \), contradicting (b).

The \( \gamma \) synchronizer [Awer 1] is composed from \( \alpha \) and \( \beta \). In \( \gamma \), an initialization phase divides the network nodes into connected clusters and assigns a directed spanning tree to every such cluster. Within each cluster, a version of the \( \beta \) synchronizer is applied. To synchronize between the clusters, a version of \( \alpha \) is applied. In the special case when each cluster is one node, the synchronizer created is \( \alpha \). In the special case when there is only one cluster, a version of \( \beta \) is created. Thus, \( \gamma \) does not ensure anything that \( \alpha \) or \( \beta \) do not ensure, including part (b) of the timely property.

4.3 Necessity and sufficiency of the timely property for passive synchronizers

In this section we show that the timely property is necessary and sufficient for proper operation of passive synchronizers. In order to state the "proper operation" property, we need several definitions that are pertinent to all synchronizers, whether they are passive or active.

**Definition 4.3 (local state for a combined protocol)**

For a combined protocol execution, the local state of a node \( i \) is defined as the collection of all local variables belonging to the (original) synchronous protocol at node \( i \).
Definition 4.4 (end of a combined protocol execution)
The end of execution of a combined protocol is defined as the time after which no original-protocol messages are sent and there are no changes in the local states of the nodes in the network.

Definition 4.5 (final outcome)
The final outcome of a protocol execution is defined as the set of all local states at the end of the execution.

Definition 4.6 (equivalent)
Let the asynchronous-protocol $P'$ be the synchronous-protocol $P$ combined with a synchronizer. An execution of $P$ and an execution of $P'$ on the same topology are said to be equivalent, if the message sent on each edge in the execution of $P$ at time $\text{pulse}(n)$ is identical to the original-protocol message sent on the same edge in the execution of $P'$ at time $t_i(n)$ (when $i$ is the node sending the message), and the final outcome of both executions is the same.

Definition 4.7 (implementation)
Protocol $P'$ is an implementation of protocol $P$ if for each execution of $P'$, there exists an equivalent execution of $P$.

Definition 4.8 (proper operation)
A synchronizer ensures proper operation if combined with any synchronous protocol $P$ produces an asynchronous protocol $P'$ that is an implementation of $P$.

Definition 4.9 ($MSG_{i,[X]}^{k}(n)$)
For a synchronous-protocol execution $X$, $MSG_{i,[X]}^{k}(n)$ is defined as the message sent by node $i$ to its neighbor $k$ at time $\text{pulse}(n)$.

For a combined protocol execution $X$, $MSG_{i,[X]}^{k}(n)$ is defined as the original-protocol message sent by node $i$ to its neighbor $k$ at time $t_i(n)$.

In the sequel, when the execution under consideration is obvious, we suppress the subscript $[X]$.

Definition 4.10 ($M_{[X]}^{k}(n)$)
For a synchronous-protocol execution $X$, $M_{[X]}^{k}(n)$ is defined as the ordered set of all messages sent to $k$ by all its neighbors at time $\text{pulse}(n)$, ordered according to the timing of their arrival at the node $k$.

For a combined protocol execution $X$, $M_{[X]}^{k}(n)$ is defined as the ordered set of all original-protocol messages sent to $k$ by all its neighbors $i$, at their own time $t_i(n)$, ordered according to the timing of their arrival at the node $k$.

Members of $M_{[X]}^{k}(n)$ contain the information about the neighbor that has sent the message, namely:

$$M_{[X]}^{k}(n) = ((MSG_{i,[X]}^{k}(n), i) | i \in G_k) .$$

In the sequel, when the execution under consideration is obvious, we suppress of the subscript $[X]$. 
**Definition 4.11 (state\_i(n))**

Let \( P \) be a synchronous protocol, let \( N = (V, E) \) be a network and let \( X \) be an execution of \( P \) on the network \( N \). The local-state of a node \( i \in V \) at time pulse\( (n) \) is a function of the messages that have arrived at this node during all former phases, of the timing of their arrival, and of the initial local-state \( \Pi_i(0) \) at the node. This function will be denoted by \( \text{state}_i(n) \), namely:

\[
\Pi_i(\text{pulse}(n)--) = \text{state}_i(n)(\Pi_i(0), M^i_{[X]}(0), M^i_{[X]}(1), \ldots, M^i_{[X]}(n-1))
\]

**Definition 4.12 (legal execution)**

An execution \( X \) on a network \( N = (V, E) \) is said to be a legal execution of a synchronous protocol \( P \) on the network \( N \) for the initial state \( \Pi_i(0) \), \( \forall i \in V \), if \( \forall n \geq 0 \), \( \forall i \in V \), node \( i \) sends at pulse\( (n) \) the message that is derived from the protocol \( P \), the initial state of \( i \) and the messages received during the former phases. Namely, if:

\[
MSG^k_{i[X]}(n) = msg^k_i(\text{state}_i(n)(\Pi_i(0), M^i_{[X]}(0), M^i_{[X]}(1), \ldots, M^i_{[X]}(n-1)))
\]

In this section, we prove that a passive synchronizer ensures proper operation if and only if it is timely.

**Theorem 4.1 — Sufficiency:**

Let \( P \) be a synchronous protocol and let \( P' \) be the (combined) protocol created by combining \( P \) with a passive timely synchronizer. Then \( P' \) is an implementation of \( P \).

**Proof:** Let \( X' \) be an execution of \( P' \) on a network \( N \). The theorem will be proved if we construct an execution \( X \) of \( P \) on \( N \) that is equivalent to \( X' \).

A synchronous execution is defined by the messages sent by each node at each pulse and by the order of their arrival at each node.

To build the synchronous execution \( X \), we must define the messages that are sent by each node at each pulse and the timing of their arrival. In \( X \), a node \( i \) sends to a neighbor \( k \) at time pulse\( (n) \) the same message that node \( i \) sends to \( k \) at \( t_i(n) \) in \( X' \). The message arrival order in \( X \) is defined as follows: if in \( X' \), \( MSG^k_{i[X]}(n) \) arrives at node \( k \) before \( MSG^k_{j[X]}(n) \), then \( MSG^k_{i[X]}(n) \) arrives at node \( k \) before \( MSG^k_{j[X]}(n) \) in \( X \). In other words, the receiving order of messages sent at a given pulse in \( X \) is the same as the receiving order in \( X' \), and all these messages are received before the next pulse.

The theorem will be proved if we prove the following two claims:

i) \( X \) is a legal execution of the protocol \( P \) on network \( N \).

ii) The local state of any node \( i \) at the end of execution \( X \) is identical to the local state of node \( i \) at the end of \( X' \).
Let $X_n$ (for $n \geq 0$) be the partial execution consisting of the first $n$ phases of $X$. The claims are proved by common induction on $X_n$.

**base:**

i) $X_0$ is a legal partial execution of the protocol $P$ on network $N$.

ii) The local state of any node $i$ at time pulse($0$)– in any execution of $P$ on $N$ is identical to the local state of $i$ at $t_i(0)$– in $X'$.

**proof:**

At pulse($0$)–, the local state of any node $i$ in any execution of $P$ on the network $N$ is the initial state. Part (b) of the *timely* property ensures that in $X'$, no original-protocol messages are received by node $i$ before $t_i(0)$. Therefore, the local state of $i$ at $t_i(0)$– in $X'$ is also the initial state, thus (ii) follows.

To prove (i), recall that $msg_i^k$ is the function that returns the message sent by node $i$ to node $k$ at time pulse($n$) when given $\Pi_i(pulse(n)–)$, and that this function is derived from the code-block associated with pulses.

The message sent by a node $i$ in $P'$ to a neighbor $k$ at time $t_i(n)$ for some $n \geq 0$, is a function of the local state of the node at $t_i(n)$–. This function is $msg_i^k$ too, since it is derived from the procedure PulseCode that contains, since that synchronizer is passive and according to definition 4.1(iii), exactly the same code as the code-block of $P$ associated with pulses.

Thus:

$$msg_i^k(\Pi_i[X_i](pulse(0)–)) = msg_i^k(\Pi_i[X_i](t_i(0)–)) = MSG_i^k[X_i](0) = MSG_i^k[X_i](0).$$

where $t_{init}$ is the time just before the execution $X'$ begins.

Thus, the message sent by node $i$ to node $k$ in $X_0$ according to the definition of $X_0$ is the same as the message that should be sent by node $i$ to node $k$ in a legal execution of $P$ (i.e. $msg_i^k(\Pi_i(pulse(0)–)))$. Therefore, $X_0$ is a legal partial execution of $P$ in the network $N$.

**Induction step:** assume that:

i) $X_n$ is a legal partial execution of the protocol $P$ on network $N$.

ii) The local state of any node $i$ at time pulse($n$)– in $X_n$ is identical to the local state of $i$ at $t_i(n)$– in $X'$.

then:

i) $X_{n+1}$ is a legal partial execution of the protocol $P$ on network $N$.

ii) The local state of any node $i$ at time pulse($n + 1$)– in $X_{n+1}$ is identical to the local state of $i$ at $t_i(n + 1)$– in $X'$.

**Proof of the induction step:**

The messages received by node $i$ during phase($n$) of $X_{n+1}$ must be, by the definition of the synchronous model in Sec. 2, the set of all messages sent to $i$ by its neighbors at time pulse($n$), in other words, the messages in $M_i[X](n)$. 
In $X'$, phase($n$) of node $i$ begins at time $t_i(n)$ and ends just before $t_i(n+1)$. Consider an original-protocol message sent to $i$ by one of its neighbors $k$ that arrived at node $i$ during $[t_i(n), t_i(n+1))$. This message must have been sent at time $t_k(m)$ for some $m$. Part (a) of the timely property ensures that this message arrives before $t_i(m+1)$, so, in order for this message to be received during phase($n$) of $i$, $m$ must satisfy $m \geq n$. Part (b) of the timely property implies that the message arrives after $t_i(m)$, so $m$ must also satisfy $m < n+1$, thus $m = n$. In other words, the set of original-protocol messages that are received by $i$ during $[t_i(n), t_i(n+1))$ is $M^i_{[X']}(n)$.

We know from the way $X$ was built at the beginning of the proof, that $M^i_{[X']}(n) = M^i_{[X']}(n)$. The local state at node $i$ at time pulse($n$) in $X'$ is identical to the local state of $i$ at $t_i(n)$ in $X'$, the messages it receives during phase($n$) of $X$ are identical to the messages received by $i$ during $[t_i(n), t_i(n+1))$ in $X'$ and in the same order. Therefore, since the synchronizer is passive, the local state of node $i$ at time pulse($n+1$) in $X_{n+1}$ is identical to the local state of node $i$ at $t_i(n+1)$ in $X'$ and (ii) is proved.

Since $X_n$ is a legal partial execution of $P$, the following is true for every partial execution $X_{n+1}$ that $X_n$ is a prefix of it:

$$
\Pi_{i,[X_{n+1}]}(\text{pulse}(n+1)-) = \text{state}_i(n+1)(\Pi_{i,[X']}(0), M^i_{[X']}(0), M^i_{[X']}(1), \ldots, M^i_{[X']}(n))
$$

From the definition of $X_{n+1}$:

$$
MSG^k_{i,[X']}(n+1) = MSG^k_{i,[X']}(n+1) = msg^k_i(\Pi_{i,[X']}(t_i(n+1)-))
$$

and from (i) this equals $msg^k_i(\Pi_{i,[X']}(\text{pulse}(n+1)-))$.

Thus $X_{n+1}$ is a legal partial execution of $P$ on network $N$ and the induction step is proved.

After having proved the base and the induction step for the partial executions, it remains to prove claims (i) and (ii) for execution $X$.

Let $t_{final}$ be the end of the execution $X'$. Let $m$ be the minimum number that satisfies $t_i(m) \geq t_{final}$, $\forall i \in V$. Since, for all $i \in V$, no original-protocol messages are sent at $t_i(m)$, by the definition of $X_m$, no messages are sent at time pulse($m$) in $X_m$. We also know that for all $i \in V$, the local state of $i$ does not change at time $t_i(m)$, or, in other words, $\text{st}_{t_i}(\Pi_{i,[X']}(t_i(m)-)) = \Pi_{i,[X']}(t_i(m)-)$. Thus, $\text{st}_{t_i}(\Pi_{i,[X']}(\text{pulse}(m)-)) = \Pi_{i,[X']}(\text{pulse}(m)-)$ and the local states of the nodes do not change on pulse($m$) of $X_m$. Therefore, pulse($m$) is the end of the execution, and $X_m$ is a complete execution. From the definition of $X$ and $X_m$, holds $X \equiv X_m$ and since $X_m$ is a legal partial execution and is also a complete execution, claim (i) is proved. Claim (ii) is proved since the induction claims are true for pulse($m$).

In Sec 4.2 we have shown that the new-BFS protocol does not work properly when combined with a passive synchronizer that does not ensure the timely property for this new-BFS protocol. This is not a necessity proof for the timely property, since a passive synchronizer may ensure the timely property when combined with the new-BFS protocol, but not when combined with some other protocol. The new-BFS may work properly with such a synchronizer even though the latter is not timely.
Theorem 4.2 — Necessity:

Let Syn be a passive synchronizer that is not timely. Then Syn does not ensure proper operation, i.e. there exists a synchronous-protocol $P_{syn}$ such that the combination of $P_{syn}$ with Syn is not an implementation of $P_{syn}$.

Proof: Since Syn is not timely, there exists a synchronous-protocol $P$ for which Syn does not ensure part (a) or (b) of the timely property. We shall build $P_{syn}$ by altering $P$ as follows. Two new variables are added to each node, $pulse\_num$ and $flag$. At each node, $pulse\_num$ is initialized to $-1$, and is incremented at each pulse prior to sending any messages. A suffix is added to each message of $P$. This suffix contains the value of $pulse\_num$ at the sending node. The other new variable at each node, $flag$, is a boolean variable that is initialized to $false$. Each time a message is received by a node, the value of its suffix is compared to the value of the local $pulse\_num$ variable; if the values are different and the value of $flag$ is $false$, $flag$ is set to $true$; otherwise the flag remains unchanged. This implies that once set to $true$, $flag$ remains $true$ for the entire execution.

Note that, since every message sent at time $pulse(n)$ in $P_{syn}$ is received during phase$(n)$, the above changes imply that, in fact, $flag_i$ remains $false$ at all nodes $i$ for each possible execution of $P_{syn}$. Consequently, every execution of $P_{syn}$ results in $flag_i = false$ for all $i$.

Let $P'$ be the combination of protocol $P$ with $Syn$, and let $P'_{syn}$ be the combination of protocol $P_{syn}$ with $Syn$.

The changes done to $P$ in order to create $P_{syn}$ do not effect the contents of the messages sent by the nodes except for the suffixes, or the timing of their transmission. The same is true for $P'$ and $P'_{syn}$. Thus, if there exists an execution $X'$ of $P'$ for which (a) or (b) of the timely property do not hold, then there exists an execution $X'_{syn}$ of $P'_{syn}$ in which messages are sent and received with the same timing as in $X'$, and thus, (a) or (b) are not satisfied in $X'_{syn}$ either.

In $X'_{syn}$, at each node $i$, $pulse\_num_i$ equals $n$ during $(t_i(n), t_i(n + 1))$ for all $n \geq 0$. If a message, sent by node $i$ to a node $j$ at time $t_i(n)$ say, violates part (a) or (b) of the timely property, then this message arrives at node $j$ during $(t_j(m), t_j(m + 1))$ for some $m \neq n$. Since the synchronizer is passive, procedure PulseCode was called exactly $m + 1$ times since the beginning of the execution. From definition 4.1(iii), PulseCode contains exactly the same code as in the code-block of $P$ associated with pulses and thus increases $pulse\_num_j$ each time it is called. Therefore, when this message arrives at node $j$, holds $pulse\_num_j \neq n$ while the suffix of $msg$ is $n$, and therefore, $flag_j$ is set to $true$. We have already proved that no execution of $P_{syn}$ can end with such a final outcome. Thus, $P'_{syn}$ is not an implementation of $P_{syn}$. □
4.4 Synchronizer $\delta$

In [LT 1], the authors introduce a passive timely synchronizer. From Theorem 4.1, this synchronizer, referred to here as $\delta$, ensures proper operation. However, the synchronizer suffers from high communication and time complexity.

The first phase of $\delta$ is exactly the same as the first phase of $\beta$. Synchronizer $\delta$ also uses the same initialization demands as the $\beta$ synchronizer, namely a spanning directed tree, rooted at a node $s$. The first phase, similarly to $\beta$, is done after the execution of each pulse. Each leaf node learns that it is safe with respect to that pulse, and sends a SAFE message to its father on the directed tree. Each node that learns that itself and all nodes in its sub-tree are safe, sends a SAFE message to its father. However, in $\delta$, the definition of safe is different than in $\beta$ as explained presently.

At the end of this process, node $s$ knows that all nodes are safe and all messages of the former pulse had been sent and received. At this time, the network links are empty of any messages.

Now comes the second phase, this phase differs from the one in synchronizer $\beta$. In this phase, $s$ starts a process in which all nodes perform the next pulse, but in such order that no node receives a message from a neighbor that has performed the next pulse before it performs the next pulse itself.

In other words, for all $n \geq 0$, a set of times $t_i(n)$, $\forall i \in V$ should be found, such that for every two neighbors $i, j$, no message sent by $i$ to $j$ at time $t_i(n)$ or later arrives at $j$ before $t_j(n)$. This will ensure that $\delta$ is timely.

This is one of the properties ensured by the PI protocol described in [Seg 1] and by the distributed snapshot protocol described in [CL 1]. The same technique is used in $\delta$ to create $\{t_i(n)\}$. When $s$ learns that all nodes are safe, it sends AWAKE messages to all its neighbors and then performs $t_s(n)$. Each node $i$ that receives an AWAKE message, sends AWAKE messages to all its neighbors and then performs $t_i(n)$. After performing $t_i(n)$ and until receiving AWAKE messages from all neighbors, node $i$ ignores all other AWAKE messages, except for remembering that they have been received.

To make sure that no node is awakened by an AWAKE message belonging to a former pulse, the definition of safe is altered in $\delta$ as follows: a node is said to be safe with respect to a given pulse if each original-protocol message sent by that node at that pulse had already arrived at the respective neighbor, and if all AWAKE messages sent by its neighbors to it when entering that pulse had been received. The first pulse is performed by $s$ in exactly the same way as described above after having received a START message.

Synchronizer $\delta$ is clearly a passive synchronizer, since it does not change the code and messages of the original protocol. The time complexity of this protocol is $D$ (when $D$ is the diameter of the network), and the communication complexity is $|V| + 2|E|$. 
4.5 Synchronizer $\delta_1$

We shall now introduce the $\delta_1$ synchronizer, which is a variation of $\delta$ that has a slightly better communication complexity, while it retains the timely property. We shall prove in Sec. 4.6 that this synchronizer has the best time and communication complexity that can be achieved by any passive timely synchronizer, and thus by any passive synchronizer that ensures proper operation.

In $\delta_1$, when $s$ first receives a START message, it sends AWAKE messages to all its neighbors and then performs $t_s(0)$. Each node $i$, when receiving the first AWAKE message, marks the link from which it was received with a special mark $p_i$, sends AWAKE to all neighbors except $p_i$, and then performs $t_i(0)$.

When node $i$ observes that it is safe and that it has received an AWAKE or a SAFE message from each of its neighbors except $p_i$, it sends a SAFE message to $p_i$. When $s$ observes that it is safe, and that it has received an AWAKE or a SAFE message from each of its neighbors, it starts the protocol all over again, but this time each node $i$ performs $t_i(1)$. In general, after initiating $t_s(n)$, when $s$ first observes that it is safe and has received AWAKE or SAFE messages from all neighbors, it performs the protocol again, performing $t_s(n + 1)$.

This synchronizer is better than $\delta$ in two respects: it does not need an initialization phase, and has a communication complexity of $2|E|$, which is slightly better than $\delta$. The code for $\delta_1$ is given in Table 2. It is clear from the code that this synchronizer is passive.

**Theorem 4.3** $\delta_1$ is a timely synchronizer.

*Proof:* We shall make use of two lemmas.

**Lemma 4.4** Synchronizer $\delta_1$ ensures part (a) of the timely property.

*Proof:* Let $M$ be an original-protocol message sent by node $i \neq s$ to a neighbor $k$ at time $t_i(n)$. Node $i$ sends SAFE to $p_i$ only after $M$ is received and confirmed by node $k$. Node $s$ sends AWAKE messages to all its neighbors and performs $t_s(n + 1)$ only after receiving SAFE messages from all its sons in the created tree. This may happen only after all other nodes, including node $i$, send the SAFE message. In case $k \neq s$, it performs $t_k(n + 1)$ only upon receiving an AWAKE message. Therefore, $t_k(n + 1) \geq t_s(n + 1)$ (equal when $k = s$), and thus, $t_k(n + 1)$ may not happen before $M$ is received and acknowledged by $k$. Hence, (a) is confirmed for messages sent by node $i \neq s$.

Let $M$ be an original-protocol message sent by the root node $s$ to a neighbor $k$ at time $t_s(n)$. Node $s$ sends AWAKE messages and performs $t_s(n + 1)$ only after being safe with respect to $t_s(n)$, and thus, only after $M$ is received and confirmed by $k$. Node $k$ performs $t_k(n + 1)$ when receiving an AWAKE message, therefore $t_k(n + 1)$ may not happen before $M$ is received by node $k$. Thus, (a) is confirmed for messages sent by $s$.

**Lemma 4.5** Synchronizer $\delta_1$ ensures part (b) of the timely property.
Protocol $\delta_1$

Messages

CONFIRM - Delivered by the data link when all original-protocol messages sent are confirmed to have been received by the other end of the link.

SAFE - Sent by each node along the tree to signal that it is safe with respect to the current pulse.

AWAKE - Sent by each node to all its neighbors before performing the next pulse.

Variables

$G_i$: The set of neighbors of $i$.

$p_i$: preferred neighbor.

$e_i(l), l \in G_i$: (number of AWAKE or SAFE messages sent to $l$) -- (number of AWAKE or SAFE messages received from $l$).

$st_i$: 1 when all original-protocol messages sent at the last pulse are confirmed, 0 otherwise.

$m_i$: 1 after each $t_i(n)$, 0 after sending SAFE to $p_i$.

Procedures

PulseCode: Contains the original-protocol pulse code.

Initialization

$s_{st_i} = 1$.

$m_i = 0$.

Algorithm for node $s$

When receiving START, CONFIRM, SAFE from neighbor $j$ or AWAKE from neighbor $j$ do

if CONFIRM then $st_s \leftarrow 1$

if SAFE or AWAKE then $e_s(j) \leftarrow e_s(j) - 1$

if START or ($st_s = 1$) and ($\forall l \in G_s$, $e_s(l) = 0$) then

$m_s \leftarrow 1; st_s \leftarrow 0$;

$\forall k \in G_s$ do

send AWAKE to $k$

$e_s(k) \leftarrow 1$

call PulseCode

Algorithm for node $i \neq s$

When receiving CONFIRM, SAFE from neighbor $j$ or AWAKE from neighbor $j$ do

if CONFIRM then $st_i \leftarrow 1$

if AWAKE and ($m_i = 0$) then

$m_i \leftarrow 1; st_i \leftarrow 0; p_i \leftarrow j$

$\forall k \in G_i - \{p_i\}$ do

send AWAKE to $k$

$e_i(k) \leftarrow 1$

call PulseCode

else

if AWAKE or SAFE then $e_i(j) \leftarrow e_i(j) - 1$

if ($st_i = 1$) and ($\forall k \in G_i - \{p_i\}$, $e_i(k) = 0$) then

$m_i \leftarrow 0$

send SAFE to $p_i$

Table 2: The $\delta_1$ synchronizer
There are two cases to be considered. In the first case, \( k = p_i \), in which case \( t_i(n) < t_k(n) \), since \( k \) performs \( t_i(n) \) only upon receiving the AWAKE message sent by node \( k \) at \( t_k(n) \). Therefore, the message \( M \) may not be received before \( t_k(n) \). In the second case, \( p_i \neq k \), so, before node \( i \) performs \( t_i(n) \), it sends an AWAKE message to \( k \). The original-protocol message \( M \) is sent by \( i \) at \( t_i(n) \), or in other words, after the AWAKE message. If \( k \) has not yet performed \( t_k(n) \) when the AWAKE message from \( i \) is received, it would perform it then. Therefore, since \( M \) is sent after the AWAKE message, it arrives at node \( k \) after \( t_k(n) \).

From lemmas 4.4 and 4.5, synchronizer \( \delta_1 \) is timely.

### 4.6 Lower bounds on the complexity, forced by the timely property

The communication complexity of \( \delta_1 \) is \( 2|E| \). The time complexity of \( \delta_1 \) is \( D \). We shall prove in this section that these are the best time and communication complexities that can be achieved by any timely passive synchronizer. Note that these bounds are true for the exact complexities, and not just for their order.

The complexity of a protocol is defined as the worst-case complexity of the protocol when executed on all networks. Therefore, it is sufficient to find a family of networks for which the lower bounds are as mentioned above. The family of networks for which we prove these bounds is the 2-redundant networks family defined below.

**Definition 4.13 (2-redundant network)**

A network is said to be 2-redundant if there are at least two disjoint paths (that do not contain common nodes or edges) between each two nodes in the network.

**Theorem 4.6** Assume that the nodes in a 2-redundant network \( N \) perform a combined protocol that is created with a timely passive synchronizer. Then no node \( i \) performs \( t_i(n+1) \), before all nodes in \( N \) are safe with respect to pulse(\( n \)).

**Proof:** Assume, in contradiction, that some node \( i \) performs \( t_i(n+1) \) while some other node, say \( k \), is not safe with respect to \( t_k(n) \). This means that there is an original-protocol message \( MSG_{i}^{k}(n) \) sent by \( k \) at \( t_k(n) \) to one of its neighbors, say \( l \), that has not arrived yet at time \( t_i(n+1) \).

Since the network is 2-redundant, there exists at least one path \( i - j_1 - j_2 - \cdots - j_m - l \) that does not contain node \( k \).

A passive synchronizer knows nothing about the original protocol, and thus, any execution of pulse(\( n+1 \)) is possible. Assume that node \( i \) sends an original-protocol message to \( j_1 \) at time \( t_i(n+1) \), node \( j_1 \) sends an original-protocol message to \( j_2 \) at time \( t_{j_1}(n+1) \) and so on for each two neighboring nodes on the path from \( i \) to \( l \).

When the message sent by node \( i \) to \( j_1 \) at time \( t_i(n+1) \) arrives at \( j_1 \), the latter has already performed \( t_{j_1}(n+1) \). When the message sent by node \( j_1 \) to \( j_2 \) arrives, \( j_2 \) has already performed...
$t_j(n + 1)$. This process continues, ending with the message sent by node $j_m$ arriving at $l$ when the latter has already performed $t_i(n + 1)$. If $MSG_k^l(n)$ is very slow, this string of messages may reach node $l$ before $MSG_k^l(n)$. Thus, $MSG_k^l(n)$ arrives at node $l$ only after $t_i(n + 1)$ — contradicting part (a) of the timely property.

The synchronizer must take the possibility of such scenario into consideration. Thus, a timely passive synchronizer may not trigger $t_i(n + 1)$ while a message belonging to pulse($n$) is still on the way.

**Theorem 4.7** The communication complexity of any passive timely synchronizer is no less than $2|E|$.

**Proof:** From Theorem 4.6, in a 2-redundant network $N$, the first node $i$ to perform pulse($n + 1$) must know that all nodes in $N$ are safe with respect to pulse($n$). Since this information must be sent by each node in a message, collecting this knowledge requires at least $|V| - 1$ messages. Just before performing pulse($n + 1$), node $i$ and any other node that performs pulse($n + 1$) based on the information that all nodes in $N$ are safe, must send a warning message to all their neighbors, and every neighbor $k$ must perform $t_k(n + 1)$ immediately upon receiving the first warning message, provided it had not performed $t_k(n + 1)$ beforehand. This is because upon performing $t_i(n + 1)$, node $i$ may send an original-protocol message and part (b) of the timely property requires that such a message arrives at $k$ after $t_k(n + 1)$.

Thus, we can divide the network nodes into two groups depending on the trigger for $t_i(n + 1)$ at the node: in the first group the trigger is knowledge that all nodes in $N$ are safe, in the second the trigger is a warning message. Nodes in the first group must send warning messages to all their neighbors, those in the second must send warning messages to all neighbors, except possibly the one they received the warning from. Thus, the number of warning messages is at least $2|E| - (|V| - 1)$, since there is at least one node in the first group. Therefore, together with the messages propagating safety knowledge, there are at least $2|E|$ messages. □

**Theorem 4.8** The time complexity of any passive timely synchronizer is no less than $D$.

**Proof:** Let $N = (V, E)$ be a 2-redundant network. The time complexity of a synchronizer for the network $N$ is the maximum time between $t_i(n)$ and $t_i(n + 1)$ for all $i \in V$ and for all $n \geq 0$ assuming the delay of all links is 1. Let $u, v \in V$ be two nodes in distance $D$. Assume $t_u(n) \geq t_v(n)$. From Theorem 4.6 node $v$ may not perform $t_v(n + 1)$ before node $u$ is safe with respect to $t_u(n)$. Since the distance between $u$ and $v$ is $D$, it takes at least $D$ units of time until $v$ is informed of the safety of $u$. Thus, $t_v(n + 1) \geq t_u(n) + D$. From the assumption $t_u(n) \geq t_v(n)$ and thus $t_v(n + 1) \geq t_v(n) + D$. □

From Theorems 4.6, 4.7, 4.8 we learn that there is no timely passive synchronizer with time and communication complexities similar to the complexities of $\alpha$, $\beta$ or $\gamma$. This means that there is no point trying to change $\alpha$, $\beta$ or $\gamma$ to obtain a timely passive synchronizer without affecting their complexity.
Theorems 4.6, 4.7, 4.8 give us also a simple tool for checking whether a passive synchronizer ensures proper operation. If a passive synchronizer has a time complexity which is better than $D$ or a communication complexity which is better than $2|E|$, this synchronizer is definitely not timely. From Theorem 4.6, if a passive synchronizer allows a node to perform a new pulse without receiving SAFE (or equivalent) information from all nodes in the network, then this synchronizer is not timely.
5 Achieving better performance

The time complexity of synchronizer $\delta_1$ is identical to the time complexity of synchronizer $\beta$ [Awer 1], which is the worst among $\alpha$, $\beta$ and $\gamma$. The communication complexity of $\delta_1$ is identical to the communication complexity of $\alpha$, which is also the worst among $\alpha$, $\beta$ and $\gamma$. According to Theorem 4.8 and Theorem 4.7, this is the best complexity that can be achieved by a timely passive synchronizer. On the other hand, according to Theorem 4.2, any passive synchronizer that ensures proper operation must be timely.

Therefore, if there is any hope in finding more efficient synchronizers that ensure proper operation, one needs to look outside the class of passive synchronizers. Synchronizers that are not passive need not necessarily be timely and in this section we present the matching property, which is built of two parts: $(a^*)$ and $(b^*)$. The matching property is necessary and sufficient for any synchronizer in order to ensure proper operation. A synchronizer that ensures this property is said to be matching.

Recall that in a synchronous protocol, the message $MSG^k_i(n)$ sent by node $i$ to node $j$ at pulse$(n)$ is a function of the local state at pulse$(n)$$. This function has been defined before, and is called $msg^k_i$:

$$MSG^k_i(n) = msg^k_i(\Pi_i(\text{pulse}(n)-)) = msg^k_i(\text{state}_i(n)(\Pi_i(\text{pulse}(0)-, M^i(0), M^i(1), \ldots, M^i(n-1)))$$

Recall also, that the local state $\Pi_i(\text{pulse}(n)+)$ immediately after performing pulse$(n)$ is a function of the local state at pulse$(n)-$. This function has been defined before, and is called $stt_i$.

$$\Pi_i(\text{pulse}(n)+) = stt_i(\Pi_i(\text{pulse}(n)-)) = stt_i(\text{state}_i(n)(\Pi_i(\text{pulse}(0)-, M^i(0), M^i(1), \ldots, M^i(n-1)))$$

The two parts of the matching property use these function definitions. Part $(a^*)$ says that in the combined protocol, the message sent by a node $i$ at time $t_i(n)$ is the message derived from these functions. This means that the message is identical to the message sent by the node when performing the original synchronous-protocol. Part $(b^*)$ says that the final outcome at each node is identical to the final outcome derived from these functions. Again, this means that it is also identical to the final outcome at the node when performing the original protocol. These arguments show intuitively that, each combined-protocol execution in which the matching property is satisfied, has an equivalent original synchronous-protocol execution. Thus, the matching property ensures proper operation. This will be proved in Theorem 5.1.

We shall now formally present parts $(a^*)$ and $(b^*)$ of the matching property.

**Definition 5.1 (matching)**

Let $Syn$ be a synchronizer, let $P'$ be the combination of an arbitrary synchronous protocol $P$ with the synchronizer $Syn$. $Syn$ is said to be matching if $(a^*)$ and $(b^*)$ below hold for any $P$ and any execution $X'$ of $P'$.
Sec. 5

The contents of the original-protocol message sent in \( X' \) by node \( i \) to node \( k \) at \( t_i(n) \) is a function of \( M^i(0), M^i(1), \ldots, M^i(n-1) \) and the local state \( \Pi_i(0) \) of \( i \) at the beginning of the execution and it satisfies:

\[
MSG^k_i(n) = \text{msg}^k_i(\text{state}_i(n)(\Pi_i(0), M^i(0), M^i(1), \ldots, M^i(n-1)))
\]

a*) The contents of the original-protocol message sent in \( X' \) by node \( i \) to node \( k \) at \( t_i(n) \) is a function of \( M^i(0), M^i(1), \ldots, M^i(n-1) \) and the local state \( \Pi_i(0) \) of \( i \) at the beginning of the execution and it satisfies:

\[
\text{stt}_i(\text{state}_i(m)(\Pi_i(0), M^i(0), M^i(1), \ldots, M^i(m-1))) = \text{state}_i(m)(\Pi_i(0), M^i(0), M^i(1), \ldots, M^i(m-1))
\]

then:

\[
\exists t', \forall t > t', \Pi_i(t) = \text{state}_i(m)(\Pi_i(0), M^i(0), M^i(1), \ldots, M^i(m-1))
\]

We shall now prove that a synchronizer ensures proper operation if and only if it is matching.

**Theorem 5.1 — Sufficiency:**

Let \( P \) be a synchronous protocol and let \( P' \) be the (combined) protocol created by combining \( P \) with a matching synchronizer. Then \( P' \) is an implementation of \( P \).

**Proof:** Let \( X' \) be an execution of \( P' \) on a network \( N \). The Theorem will be proved if we construct an execution \( X \) of \( P \) on \( N \) that is equivalent to \( X' \).

A synchronous execution is defined by the messages sent by each node at each pulse and by the order of their arrival at each node.

To build the synchronous execution \( X \), we must define the messages that are sent by each node at each pulse and the timing of their arrival. In \( X \), a node \( i \) sends to a neighbor \( k \) at time \( pulse(n) \) the same message that node \( i \) sends to \( k \) at time \( t_i(n) \) in \( X' \). The message arrival order in \( X \) is defined as follows: if in \( X' \), \( MSG^k_{i,[X]}(n) \) arrives at node \( k \) before \( MSG^k_{j,[X]}(n) \), then \( MSG^k_{i,[X]}(n) \) arrives at node \( k \) before \( MSG^k_{j,[X]}(n) \) in \( X \). In other words, the receiving order of messages sent at a given pulse in \( X \) is the same as the receiving order in \( X' \), and all these messages are received before the next pulse.

Recall that, an execution \( X \) on a network \( N = (V, E) \) is said to be a legal execution of a synchronous protocol \( P \) on the network \( N \) for the initial state \( \Pi_i(0), \forall i \in V, \) if \( \forall n \geq 0, \forall i \in V, \) node \( i \) sends at \( pulse(n) \) the message that is derived from the protocol \( P \), the initial state of \( i \) and the messages received during the former phases. Namely, if:

\[
MSG^k_{i,[X]}(n) = \text{msg}^k_i(\text{state}_i(n)(\Pi_i(0), M^i_{[X]}(0), M^i_{[X]}(1), \ldots, M^i_{[X]}(n-1)))
\]

The theorem will be proved if we prove the following two claims:

i) \( X \) is a legal execution of the protocol \( P \) on network \( N \), namely the messages sent in \( X \) are produced according to the code of \( P \).
Sec. 5

ii) The local state of node \( i \) at the end of the execution \( X \) is identical to the local state of node \( i \) at the end to execution \( X' \).

Let \( X_n \) (for \( n \geq 0 \)) be consisted of the first \( n \) phases of \( X \). We prove by induction that \( X_n \) is a legal partial execution of \( P \).

**base:**

\( X_0 \) is a legal partial execution of the protocol \( P \) on the network \( N \).

**proof:**

To prove that \( X_0 \) is a legal partial execution, the following property should be satisfied by it:

\[
\forall i \in V, \forall k \in G_i : MSG^k_{i,[X_0]}(0) = msg^k_i(\text{initial-state}) = msg^k_i(\text{state}_i(0)(\Pi_i(0)))
\]

or in other words, the message that is sent at time \( \text{pulse}(0) \) in \( X \) is indeed the message that should have been sent according to the code of \( P \).

This property is satisfied because:

\[
\forall i \in V, \forall k \in G_i : MSG^k_{i,[X_1]}(0) = MSG^k_{i,[X]}(0)
\]

and according to part \((a^*)\) of the matching property:

\[
MSG^k_{i,[X]}(0) = msg^k_i(\text{state}_i(0)(\Pi_i(0)))
\]

**Induction step:**

Assume that \( X_n \) is a partial execution of the protocol \( P \) on the network \( N \), then we need to show that \( X_{n+1} \) is a partial execution of the protocol \( P \) on the network \( N \).

**proof:**

To prove that \( X_{n+1} \) is a legal partial execution, the following property should be satisfied by it:

\[
\forall m \leq n + 1, \forall i \in V, \forall k \in G_i : MSG^k_{i,[X_{n+1}]}(m) = msg^k_i(\text{state}_i(m)(\Pi_i(0), M^i_{[X]}(0), M^i_{[X]}(1), \ldots, M^i_{[X]}(m-1)))
\]

This property is satisfied for all \( m \leq n \) from the induction assumption.

For \( m = n + 1 \), due to the definition of \( X \), \( MSG^k_{i,[X]}(n+1) = MSG^k_{i,[X]}(n+1) \), and according to part \((a^*)\) of the matching property:

\[
MSG^k_{i,[X]}(n+1) = msg^k_i(\text{state}_i(n+1)(\Pi_i(0), M^i_{[X]}(0), M^i_{[X]}(1), \ldots, M^i_{[X]}(n)))
\]

Thus, the induction step is proved.

After having proved the base and the induction step for partial executions, it remains to prove claims (i) and (ii) for execution \( X \).

Let \( m \) be the last pulse of execution in \( X \), then, \( \forall n > m, M^i_{[X]}(n) = \emptyset \) and:
Sec. 5

From the definition of $X'$, $\forall n > m$, $M^i_{X'}(n) = \emptyset$. Now:

$$\forall i \in V : \Pi_{i,X}(\text{pulse}(m + 1) -) = \text{state}_i(m + 1)(\Pi_{i,X}(0), M^i_{X}(0), M^i_{X}(1), \ldots, M^i_{X}(m))$$

From part (b*) of the matching property, $\exists t'$ such that the above equals $\Pi_{i,X}(t')$ for all $t > t'$.

Thus, claim (ii) is proved.

Claim (i) is true since $X \equiv X_m$ and since $X_m$ is a legal partial and complete execution. $\square$

Theorem 5.2 — Necessity:

Let $Syn$ be a synchronizer that does not ensure one or both parts of the matching property. Then $Syn$ does not ensure proper execution, i.e. there exists a synchronous-protocol $PSyn$ such that the combination of $PSyn$ with $Syn$ is not an implementation of $PSyn$.

Proof: Assume that $Syn$ does not ensure part (a*) of the matching property. There exists at least one synchronous protocol, $P_{Syn}$, for which $Syn$ does not ensure (a*). Let $P'$ be the combination of $P_{Syn}$ and $Syn$, and let $X'$ be an execution of $P'$ that does not satisfy (a*). Then $\exists n \geq 0, \exists i \in V, \exists k \in G_i$ such that, in $X'$:

$$MSG^k_i(n) \neq msg^k_i(\text{state}_i(n)(\Pi_i(0), M^i(0), M^i(1), \ldots, M^i(n - 1)))$$

This means that any equivalent execution $X$, is not an execution of the synchronous-protocol $P_{Syn}$. 

If part (b*) of the matching property is not ensured by $Syn$, then there exists at least one synchronous protocol $P_{Syn}$, for which $Syn$ does not ensure (b*). Let $P'$ be the combination of $P_{Syn}$ and $Syn$, and let $X'$ be an execution of $P'$ that does not satisfy (b*), then:

$$\exists m \geq 0, \exists i \in V : \Pi_i(t_{final}) \neq \text{state}_i(m)(\Pi_i(0), M^i(0), M^i(1), \ldots, M^i(m - 1))$$

$$\forall n \geq m : M^i(n) = \emptyset$$

Where $t_{final}$ is the time when the execution $X'$ ends.

Assume, in contradiction, that there exists an execution of $P_{Syn}$ that is equivalent to $X'$. Let us call this execution $X_{Syn}$.

The final outcome of $X_{Syn}$ is:

$$\forall n \geq m, \forall i \in V, \Pi_i(\text{pulse}(n) -) = \text{state}_i(m)(\Pi_i(0), M^i(0), M^i(1), \ldots, M^i(m - 1))$$

Thus, the final outcomes of $X_{Syn}$ and $X'$ are different and the executions are not equivalent — contradiction. $\square$
6 Message-monitoring synchronizers

As shown in previous sections, synchronizers that ensure proper operation and have better performance than \( \delta_1 \) must be non-passive, i.e. active. In this section we present one class of active synchronizers: the message-monitoring synchronizers. In message-monitoring synchronizers, the synchronizer-code, i.e. the code inserted by the synchronizer transformation, is aware of incoming original-protocol messages. This is done by rewriting the code-blocks associated with the incoming original-protocol messages. Thus, part (ii) of definition 4.1 of a passive synchronizer is not satisfied. The details of the implementation of message-monitoring synchronizers are discussed in Sec. 6.1.

Two other types of message-monitoring synchronizers are discussed in Sec. 6.2. One type allows the synchronizer code at a node to know about original-protocol messages sent by that node. This is done by rewriting the procedure in the original-protocol pulse code that sends messages. Thus, part (iii) of definition 4.1 is not satisfied, since the original-protocol pulse code-block is altered. In the other type, the rewritten procedure for sending messages may alter the original-protocol messages by adding a suffix to them. Thus, part (i) of definition 4.1 of a passive synchronizer is not satisfied.

6.1 Synchronizer \( \epsilon_1 \)

In this section we present the \( \epsilon_1 \) synchronizer, which is a message-monitoring matching synchronizer and thus ensures proper operation. The synchronizer has \( O(|V|) \) communication and \( O(D) \) time complexity. The communication complexity of \( \epsilon_1 \) is much better than that of \( \delta_1 \) \( (O(|E|)) \), and is exactly the communication complexity of synchronizer \( \beta \). The time complexity of \( \epsilon_1 \) is the same as the time complexity of \( \delta_1 \) and \( \beta \).

Synchronizer \( \epsilon_1 \) is a variation of the \( \delta \) synchronizer. It needs the same initialization, i.e. a spanning tree for the network graph. The first phase, similarly to \( \delta \) and \( \beta \), is performed after the execution of each pulse. Each leaf node learns that it is safe with respect to that pulse, and sends a SAFE message to its father. Each node that learns that itself and all nodes in its sub-tree are safe, sends a SAFE message to its father. At the end of this process, node \( s \), the root of the spanning tree, knows that all nodes are safe and all original-protocol messages of the former pulse had already been received.

The second phase is different from the one in synchronizer \( \delta \). In synchronizer \( \delta \), node \( s \) starts the second phase by performing a new pulse, but only after sending warning messages to all its neighbors. In this way, the original-protocol messages sent by \( s \) do not arrive at the neighbors before they perform the new pulse. In synchronizer \( \epsilon_1 \), node \( s \) does not send warning messages; instead, it simply performs the new pulse. The idea is to have the original-protocol messages, if such are sent, serve also as warning messages. When a node receives the first original-protocol message that was sent by one of its neighbors at the new pulse, it does not process it immediately; instead, it performs the new pulse and only afterwards processes the received message. The effect is as if the message is received only after the new pulse has been performed. We shall indicate
presently how does the node know that the message was sent at the new pulse.

The other nodes in the network follow the same algorithm. When they perform the new pulse, they do not send warning messages, and the original-protocol messages serve as warning messages.

One problem with the protocol described above is that some nodes might never get to perform the next pulse. For example, if no node sends an original-protocol message to node \( i \) when executing the new pulse, then node \( i \) never gets to know that it should perform the new pulse.

The solution to this problem is to use the spanning tree to ensure that all nodes perform the new pulse. This is done by adding to the protocol described above a synchronizer message, called AWAKE, that is sent by a node to all its sons in the spanning tree before performing a pulse. A new pulse is triggered now either by receiving an AWAKE message or by receiving an original-protocol message prior to the AWAKE message.

Another problem with this protocol is that, when a node \( i \) receives an original-protocol message during \((t_i(n), t_i(n + 1))\), it does not know whether this message belongs to \( M^i(n) \), which means that it should be simply processed, or to \( M^i(n + 1) \), in which case it should trigger \( t_i(n + 1) \).

We shall illustrate this by presenting two execution examples, of two different protocols combined with the synchronizer described above, on the same network. In both executions, node \( i \) receives identical messages in identical order. In both executions, node \( i \) receives during \((t_i(n), t_i(n + 1))\) one original-protocol message from neighbor \( k \). Yet, in one execution, this message belongs to \( M^i(n) \) and in the other execution, the message belongs to \( M^i(n + 1) \). The two execution examples take place in the network shown in Fig. 3a) with the spanning tree of Fig. 3b).

In both examples, node \( s \) performs \( t_s(0) \) and sends an AWAKE message to nodes \( i \) and \( k \). No original-protocol messages are sent by \( s \) at \( t_s(0) \). Nodes \( i \) and \( k \) receive the message at approximately the same time and perform \( t_i(0) \) and \( t_j(0) \) respectively. In the first example, node \( k \) sends at time \( t_j(0) \) an original-protocol message to node \( i \). Node \( i \) has no original-protocol messages to send at time \( t_i(0) \). Thus, node \( i \) sends immediately a SAFE message, and after a while it receives the original-protocol message sent by node \( k \).

In the second example, both \( i \) and \( k \) have no original-protocol messages to send at times \( t_i(0) \) and \( t_j(0) \) respectively. Thus, they send SAFE messages to \( s \) immediately after performing pulse 0. When the two SAFE messages are received by \( s \), the latter sends an AWAKE message to \( i \) and \( k \), while performing \( t_s(1) \). The AWAKE message to \( i \) is delayed. When the AWAKE message to \( k \) arrives, the latter performs \( t_j(1) \) and sends an original-protocol message to node \( i \). This message arrives at node \( i \) before the arrival of the AWAKE message from node \( s \) and thus before \( t_i(1) \).

In both executions, node \( i \) receives the same messages, in the same order: an AWAKE message from node \( s \), and then an original-protocol message from node \( k \). But, in the first example, the message belongs to \( M^i(0) \) and should be processed immediately, while in the second, it belongs to \( M^i(1) \) and thus \( i \) should first perform \( t_i(1) \), and only then process the message.

In order to solve this problem, we add one more phase and one more message type to the synchronizer protocol. The new message type is named ALLSAFE. The new phase starts at the
Figure 3: Three nodes network and spanning tree example

point in time when node $s$ receives SAFE messages from all its sons and knows that all nodes in
the network are safe. Instead of starting the second phase as described above by sending AWAKE
messages and performing the new pulse, it starts a PIF of ALLSAFE messages along the tree
[Seg 1]: a node sends an ALLSAFE message to all its neighbors in the tree except its father when
it receives an ALLSAFE message from its father, and sends an ALLSAFE message to its father
when it has received ALLSAFE messages from all its neighbors in the tree. When node $s$ receives
an ALLSAFE message from all neighbors in the tree, it knows that all nodes in the network have
received the ALLSAFE message. At this point it performs the next pulse and sends AWAKE
messages, thus initiating the phase during which all nodes perform the next pulse.

A node $i$ that receives an original-protocol message during $(t_i(n), t_i(n + 1))$, knows that this
message belongs to $M^i(n)$ if an ALLSAFE message has not been received yet from its father since
performing the last pulse. Node $i$ knows that the message received belongs to $M^i(n + 1)$ if an
ALLSAFE message has already been received from its father. This argument is true because all
nodes are already safe when node $i$ receives the ALLSAFE message, and no node performs the
next pulse before all nodes in the network have received the ALLSAFE message.

The $\epsilon_1$ synchronizer is not a passive synchronizer, since in combined-protocols created by the
$\epsilon_1$ transformation, received original-protocol messages might initiate synchronizer-code execution.
To implement this new feature, the transformation $\epsilon_1$ makes some more alterations to the input
synchronous-protocol:

(i) It places the code associated with each original-protocol message in a procedure. The code
associated with a message named $msg$ is placed (unchanged) in a procedure named $msgCode$. Each
such procedure receives two parameters, which are the message it should process, and the name of
the link from which this message was received.
(ii) It inserts new code-blocks for the original-protocol messages.

(iii) The new code-block written for each original-protocol message must call the procedure that contains the original code for the message exactly once. For example, the new code-block written for an original-protocol message named \( \text{msg} \), must call the procedure \( \text{msgCode} \) exactly once.

The code for the \( \varepsilon_1 \) synchronizer is presented in Table 3.

In \( \varepsilon_1 \), for each pulse, \(|V|\) SAFE messages, \(2|V|\) ALLSAFE messages and \(|V|\) AWAKE messages are sent. Hence, the communication complexity of \( \varepsilon_1 \) is \( 4|V| \). The time complexity of \( \varepsilon_1 \) is \( 2D \).

**Theorem 6.1** \( \varepsilon_1 \) is a matching synchronizer.

**Proof:**

**Lemma 6.2** For all \( n \geq 0 \), and for all \( i \in V \):

\[
\Pi_i(t_i(n)-) = \text{state}_i(n)(\Pi_i(0), M^i(0), M^i(1), \ldots, M^i(n - 1))
\]

**Proof:** The lemma is proved by induction on \( n \).

**Base:**
Node \( s \) performs \( t_s(\theta) \) immediately after receiving the START message, and thus, before any of the other nodes gets a chance to send any message. Therefore, \( \Pi_s(t_s(0)-) = \Pi_s(0) = \text{state}_s(0)(\Pi_s(0)) \).

Assume that an original-protocol message is received by node \( i \) at time \( t \leq t_i(0) \), then, since \( i \) has not performed \( t_i(0) \) yet, it performs \( t_i(0) \) now, before treating the message. Thus, in such cases \( t = t_i(\theta) \) and \( \Pi_i(t_i(\theta)-) = \Pi_i(0) = \text{state}_i(0)(\Pi_i(0)) \). This is true in a trivial way for nodes that perform \( t_i(0) \) before receiving any original-protocol messages, i.e. upon receiving an AWAKE message.

**Induction Step:**
Assume that:

\[
\Pi_i(t_i(n)-) = \text{state}_i(n)(\Pi_i(0), M^i(0), M^i(1), \ldots, M^i(n - 1))
\]

We shall prove that:

\[
\Pi_i(t_i(n + 1)-) = \text{state}_i(n + 1)(\Pi_i(0), M^i(0), M^i(1), \ldots, M^i(n))
\]

According to the induction assumption:

\[
\Pi_i(t_i(n)-) = \text{state}_i(n)(\Pi_i(0), M^i(0), M^i(1), \ldots, M^i(n - 1))
\]

Let \( t_{SAFE}(n) \) denote the first time after \( t_s(n) \) and after all node \( i \neq s \) had already performed \( t_i(n) \), when node \( s \) receives SAFE messages from all its sons and sends ALLSAFE messages. At \( t_{SAFE}(n) \), all the nodes in the network are safe with respect to pulse(n), and thus, \( \forall i \in V \), all the messages...
Protocol ε₁

Messages
CONFIRM – Delivered by the data link when all original-protocol messages sent are confirmed to have been received by the other end of the link.
SAFE – Sent by each node along the tree to signal that it is safe with respect to the current pulse.
AWAKE – Sent by each node to some of its sons in the spanning tree after performing a pulse.
ALL_SAFE – Used to propagate the fact that all network nodes are safe.

Variables
Gᵢ: The set of neighbors of i.
Fatherᵢ: The father of i in the tree. nil for s. Set during the initialization phase.
Sonsᵢ: The set of sons of i. Set during the initialization phase.
Safeᵢ(l), l ∈ Sonsᵢ: 1 when a SAFE message has been received from l, 0 otherwise.
All Safeᵢ(l), l ∈ Sonsᵢ: 1 when an ALL_SAFE message has been received from l, 0 otherwise.
stᵢ: 0 — after tᵢ(n), not safe yet. 1 — node is safe, but not all its sub-tree. 2 — sub-tree is safe, ALL_SAFE not received yet. 3 — ALL_SAFE received from fatherᵢ.

Procedures
msgCode: Contains the original-protocol message code for treating messages of type msg.
PulseCode: Contains the original-protocol pulse code.

Initialization
stᵢ = 3. All Safeᵢ(k) = 1, for all k ∈ Sonsᵢ. Safeᵢ(k) = 1, for all k ∈ Sonsᵢ.

Algorithm for node s
When receiving an original-protocol message named msg from neighbor l do
  call msgCode(msg, l)
  When receiving START, CONFIRM, or SAFE or ALL_SAFE from neighbor l do
    if CONFIRM then stᵢ ← 1
    if SAFE then Safeᵢ(l) ← 1
    if (∀k ∈ Sonsᵢ, Safeᵢ(k) = 1) and (stᵢ = 1) then
      stᵢ ← 3; ∀k ∈ Sonsᵢ, Safeᵢ(k) ← 0; ∀k ∈ Sonsᵢ, All Safeᵢ(k) ← 0
      send ALL_SAFE to all k ∈ Sonsᵢ
    if ALL_SAFE then All Safeᵢ(l) ← 1
    if (∀k ∈ Sonsᵢ, All Safeᵢ(k) = 1) and (stᵢ = 3) then
      stᵢ ← 0
      send AWAKE to all k ∈ Sonsᵢ
      call PulseCode

Algorithm for node i ≠ s
When receiving AWAKE or an original-protocol message named msg from neighbor l do
  if stᵢ = 3 then
    stᵢ ← 0
    send AWAKE to all k ∈ Sonsᵢ
    call PulseCode
  if msg then msgCode (msg, l)
  When receiving CONFIRM, or SAFE, ALL_SAFE from neighbor l do
    if CONFIRM then stᵰ ← 1
    if SAFE then Safeᵰ(l) ← 1
    if (∀k ∈ Sonsᵰ, Safeᵰ(k) = 1) and (stᵰ = 1) then
      stᵰ ← 2; send SAFE to Fatherᵰ;
    if ALL_SAFE then
      if l = Fatherᵰ then
        stᵰ ← 3; ∀k ∈ Sonsᵰ, Safeᵰ(k) ← 0; ∀k ∈ Sonsᵰ, All Safeᵰ(k) ← 0
        send ALL_SAFE to all k ∈ Sonsᵰ
      else All Safeᵰ(l) ← 1
      if (∀k ∈ Sonsᵰ, All Safeᵰ(k) = 1) and stᵰ = 3 then
        send ALL_SAFE to Fatherᵰ

Table 3: The ε₁ synchronizer
in \( M^i(n) \) are received by \( i \) before \( t_{SAFE}(n) \). There are two cases in which \( t_i(n) \) is performed for \( i \neq s \). The first is upon reception of an AWAKE message and the second is upon the reception of the first original-protocol message in \( M^i(n) \). In the first case, node \( i \) performs PulseCode at \( t_i(n) \). All the messages in \( M^i(n) \) are received by node \( i \) after \( t_i(n) \) and before \( t_{SAFE}(n) \), causing node \( i \) to perform the relevant original-protocol message code each time. Therefore, the code performed by node \( i \) from time \( t_i(n) \) and until time \( t_{SAFE}(n) \) is the same code performed by \( i \) during phase\((n)\) of the original-protocol execution during which the messages in \( M^i(n) \) are received. Thus:

\[
\Pi_i(t_{SAFE}(n)) = state_i(n + 1)(\Pi_i(0), M^i(0), M^i(1), \ldots, M^i(n))
\]

In the second case, \( t_i(n) \) is triggered by the first original-protocol message in \( M^i(n) \). Node \( i \) performs PulseCode at \( t_i(n) \), immediately afterwards it performs the original-protocol message code relevant to the original-protocol message received. All the following messages in \( M^i(n) \) are received by node \( i \) before \( t_{SAFE}(n) \), causing node \( i \) to perform the relevant original synchronous-protocol message code each time. Clearly, the original-protocol code performed is identical to the original-protocol code performed in the first case, and thus:

\[
\Pi_i(t_{SAFE}(n)) = state_i(n + 1)(\Pi_i(0), M^i(0), M^i(1), \ldots, M^i(n))
\]

The case for node \( s \) is identical to the first case since \( t_s(n) \) is performed before any other node \( i \neq s \) performs \( t_i(n) \), and thus, before any message in \( M^s(n) \) is sent.

The synchronizer protocol proceeds as follows: A PIF \([Seg 1]\) of ALLSAFE message is performed on the spanning tree, ALLSAFE messages are sent along the tree from the root node \( s \) to the leaf nodes, and then back from the leaf nodes to \( s \). When node \( s \) receives ALLSAFE messages from all its sons, it performs \( t_s(n + 1) \). Since no original-protocol code is performed during \([t_{SAFE}(n), t_s(n + 1)]\), the local state of all nodes does not change. Therefore:

\[
\Pi_s(t_s(n + 1)--) = state_s(n + 1)(\Pi_s(0), M^s(0), M^s(1), \ldots, M^s(n))
\]

At \( t_s(n + 1) \), node \( s \) sends AWAKE messages to all its sons, and may send original-protocol messages to some of its neighbors. Thus \( s \) initiates a process during which all the nodes perform \( t_i(n + 1) \). A node \( i \neq s \) performs \( t_i(n + 1) \) when the first of the following happen: a reception of an AWAKE message or a reception of an original-protocol message. In both cases, the original-protocol pulse code is performed before any original-protocol message code. Therefore, the local-state of \( i \) does not change until \( t_i(n + 1) \), and:

\[
\Pi_i(t_i(n + 1)--) = state_i(n + 1)(\Pi_i(0), M^i(0), M^i(1), \ldots, M^i(n))
\]

\( \square \)

Part \((a^*)\) of the matching property follows from lemma 6.2 and from the fact that the code performed by node \( i \) at each \( t_i(n) \) is identical to the code performed in the original synchronous-protocol at each pulse. Part \((b^*)\) of the matching property follows immediately from lemma 6.2, and thus the theorem is proved.

\( \square \)
6.2 Synchronizers $\varepsilon_2, \varepsilon_3$

Synchronizer $\varepsilon_2$ is another version of $\varepsilon_1$. In synchronizer $\varepsilon_2$, the problem of finding out whether the original-protocol message received belongs to $M^i(n)$ or to $M^i(n+1)$ is not solved by using the ALL_SAFE messages and an extra phase. Instead, a suffix is added to each original-protocol message, containing the information about the pulse in which the message is sent. This synchronizer has better time and communication complexities than $\varepsilon_1$, but may be implemented only if original-protocol message alteration is allowed. Since at each node $i$, the original-protocol messages received during $(t_i(n), t_i(n+1))$ belong only to $M^i(n)$ or $M^i(n+1)$, a one-bit suffix is sufficient.

In order to implement the new feature, the transformation $\varepsilon_2$ should make even more changes to the synchronous-protocol than $\varepsilon_1$:

(i) It inserts a new procedure, named NewSendMessage. The procedure receives two parameters, an original-protocol message, and the name of a link. It sends the message on the link, after adding the proper suffix.

(ii) The transformation changes the original code of the synchronous protocol, each call to send_message in order to send an original-protocol message is replaced by a call to NewSendMessage, with the same parameters.

(iii) A new operation is used by the protocol created by the transformation. This operation is named $\text{concat}$, it receives two operands, one is an original-protocol message and the other is a synchronizer variable name. It creates an expanded original-protocol message in which the value of the synchronizer variable is added as suffix to the original-protocol message.

The code for the $\varepsilon_2$ synchronizer is presented in Table 4.

In some cases, suffix addition cannot be implemented, because no alteration of the original-protocol messages format is allowed. Synchronizer $\varepsilon_3$ is a version of $\varepsilon_2$, that does not require this alteration. The idea of $\varepsilon_3$ is that the procedure NewSendMessage sends two messages each time it is called. The first is a synchronizer message named PULSE_NUM, that contains the number of the pulse at which the procedure is called. Immediately afterwards, it sends the original-protocol message given to it as a parameter. Each node maintains an array. The array at node $i$ is $\text{pulse.num}_i(l)$, $\forall l \in G_i$. When a PULSE_NUM($n$) message is received from neighbor $l$, node $i$ sets $\text{pulse.num}_i(l) \leftarrow n$. When the original-protocol message is received from neighbor $l$, the number of the pulse at which it was sent is the same as the value of $\text{pulse.num}_i(l)$. 
Protocol $\epsilon_2$

Messages
CONFIRM – Delivered by the data link when all original-protocol messages sent are confirmed to have been received by the other end of the link.
SAFE – Sent by each node along the tree to signal that it is safe with respect to the current pulse.
AWAKE – Sent by each node to all its sons in the spanning tree before performing the next pulse.

Variables
Father$_i$ : The father of $i$ in the tree. nil for $s$. Set during the initialization phase.
Sons$_i$ : The set of sons of $i$. Set during the initialization phase.
Safe$_i(l)$, $l \in$ Sons$_i$ : 1 when a SAFE message has been received from $l$, 0 otherwise.
st$_i$ : 1 when all original-protocol message sent by $i$ at the last pulse are confirmed, 0 otherwise.
Z$_i$ : A counter, contains the number of the last pulse performed by node $i$ (a one bit counter is sufficient).
Z$_i^l$ : contains the suffix of the last received original-protocol message (one bit).
Z$_i^A$ : A counter, contains the number of AWAKE messages received (one bit).

Procedures
msgCode : Contains the original-protocol message code for treating messages of type msg.
PulseCode : Contains the original-protocol pulse code.

Initialization
Z$_i$ = -1. Z$_i^A$ = 0. st$_i$ = 1.
Safe$_s(k)$ = 1, for all $k \in$ Sons$_s$.

Algorithm for node $s$

procedure NewSendMessage (msg, l)
msg $\leftarrow$ concat(msg, Z$_s$)
send msg to neighbor $l$

When receiving an original-protocol message named msg from neighbor $l$ do

call msgCode(msg, l)

When receiving START, CONFIRM, or SAFE from neighbor $l$ do

if CONFIRM then st$_i$ $\leftarrow$ 1
if SAFE then Safe$_s(l)$ $\leftarrow$ 1
if ($\forall k \in$ Sons$_s$, Safe$_s(k)$ = 1) and (st$_s$ = 1) then
st$_i$ $\leftarrow$ 0; $\forall k \in$ Sons$_s$, Safe$_s(k)$ $\leftarrow$ 0; Z$_s$ $\leftarrow$ Z$_s$ + 1
send AWAKE to all $k \in$ Sons$_s$
call PulseCode

Algorithm for node $i \neq s$

procedure NewSendMessage (msg, l)
msg $\leftarrow$ concat(msg, Z$_i$)
send msg to neighbor $l$

When receiving AWAKE or an original-protocol message named msg from neighbor $l$ do

if msg then Z$_i^l$ $\leftarrow$ suffix(msg)
else Z$_i^l$ $\leftarrow$ Z$_i^A$, Z$_i^A$ $\leftarrow$ Z$_i^A$ + 1
if Z$_i^l$ $\neq$ Z$_i$ then
st$_i$ $\leftarrow$ 0; $\forall k \in$ Sons$_i$, Safe$_i(k)$ $\leftarrow$ 0; Z$_i$ $\leftarrow$ Z$_i$ + 1
send AWAKE to all $k \in$ Sons$_i$
call PulseCode

if msg then call msgCode(msg, l)

When receiving CONFIRM, or SAFE from neighbor $l$ do

if CONFIRM then st$_i$ $\leftarrow$ 1
if SAFE then Safe$_i(l)$ $\leftarrow$ 1
if ($\forall k \in$ Sons$_i$, Safe$_i(k)$ $\approx$ 1) and (st$_i$ $\approx$ 1) then send SAFE to father$_i$

Table 4: The $\epsilon_2$ synchronizer
7 Variable-duplicating synchronizers

In this section we present another class of active synchronizers, the *variable-duplicating synchronizers*. Variable-duplicating synchronizers are synchronizers that create a set of variables, called the *backup set*, at each node, which contains a copy of all original-protocol variables at that node. Thus part (v) of Definition 4.1 of a passive synchronizer is not satisfied. As shall be seen in the sequel, variable-duplicating synchronizers violate parts (iii) and (iv) of Definition 4.1 as well.

We shall now present a method for transforming passive synchronizers that do not ensure proper operation into variable-duplicating synchronizers that ensure proper operation. The created variable-duplicating synchronizer has the same time and message order of complexities as the passive synchronizer it was created from. The method works for all passive synchronizers in which node $i$ performs $t_i(n+1)$ when it is informed that all its neighbors are safe with respect to pulse$(n)$.

The best way to explain how passive synchronizers are transformed into variable-duplicating synchronizers is by using two stages. In the first stage, the passive synchronizer, named $Syn_i$ say, is transformed into another passive synchronizer $Syn_1$. Synchronizer $Syn_1$ is a special kind of a passive synchronizer that creates at each node $i$ a clock, whose $(2n)$-th pulse is denoted by $t_i(n)$, and whose $(2n+1)$-th pulse is denoted by $T_i(n)$. Messages are sent only at times $t_i(n)$, and the synchronizer ensures that a message sent by node $i$ to node $k$ at time $t_i(n)$ is received during $(T_k(n-1), T_k(n))$. Such a synchronizer will be referred to as a *two-phased synchronizer*.

In $Syn_1$, after each pulse, the protocol of $Syn$ is used to propagate the safety information. When a node $i$ knows that all its neighbors are safe, it performs $T_i(n)$. A node that performs $T_i(n)$ is said to be *ready*. Now, the protocol of $Syn$ is used to propagate the *ready* information, and node $i$ performs $t_i(n+1)$ as soon as it knows that all its neighbors are *ready*, i.e. have performed $T_i(n)$.

*Two-phased* passive synchronizers do not ensure proper operation, since they are not *timely*. The second stage of our method is to transform $Syn_1$ into a variable-duplicating synchronizer $Syn_1'$ that is active and ensures proper operation.

In $Syn_1'$, at each $T_i(n)$ the original-protocol variables are copied into the backup set (thus violating part (iv) of Definition 4.1) and the original-protocol pulse code is performed except of the commands that send messages (thus part (iii) of Definition 4.1 is violated). At each $t_i(n)$, the original-protocol code, altered to reference the backup set variables, is performed, and original-protocol messages are sent (thus part (iii) of Definition 4.1 is violated again). Note that between $T_i(n)$ and $t_i(n+1)$, the original set of variables keeps changing according to the original code when a message is received, but the backup set of variables is frozen.

The method is demonstrated for the $\alpha$ and $\gamma$ synchronizers in Sec. 7.1 and 7.2 respectively. The synchronizers presented in these sections are: $\alpha_1$ and $\gamma_1$ are the *two-phased* versions of $\alpha$ and $\gamma$ respectively; $\alpha_1'$ and $\gamma_1'$ are the variable-duplicating versions of $\alpha_1$ and $\gamma_1$ respectively.
7.1 Synchronizer $\alpha'_1$

We shall now describe synchronizer $\alpha_1$ which is a variation of synchronizer $\alpha$, and a variable-duplicating matching synchronizer named $\alpha'_1$ which is a variation of $\alpha_1$ that ensures proper operation without delaying messages.

In synchronizer $\alpha_1$, when node $i$ learns that it is safe with respect to $t_i(n)$, it sends a SAFE message to all its neighbors. When $i$ learns that all its neighbors are safe, it sends a READY message to all its neighbors. When $i$ receives READY messages from all its neighbors, it generates $t_i(n+1)$. Synchronizer $\alpha_1$ generates an additional sequence of clock-pulses $T_i(0), T_i(1), \ldots$ at each node $i$, with the property that for all $n \geq 0$, $t_i(n) < T_i(n) < t_i(n+1)$. $T_i(n)$ is defined as the $(n+1)$-th time when node $i$ sends READY messages to all its neighbors.

Synchronizer $\alpha'_1$ is a version of $\alpha_1$. The synchronizer protocol of $\alpha'_1$ is identical to the synchronizer protocol of $\alpha_1$, except that the synchronizer transformation of $\alpha'_1$ makes some more changes to the original-protocol. A new set of synchronizer variables is added at each node, this set of variables is a copy of the local state of the node, i.e. it contains a copy of each original-protocol variable. We shall call this variables set the backup set. The initialization conditions of the backup set are identical to the initialization conditions of the original synchronous-protocol variables. Two new procedures are introduced instead of the PulseCode procedure that is used in passive synchronizers (Definition 4.1): PulseVarCode and PulseSendCode. Procedure PulseVarCode is a copy of PulseCode, except that the commands for sending messages are deleted. Procedure PulseSendCode is another copy of PulseCode, except that every reference to an original-protocol variable is replaced by a reference to its twin in the backup set. Another procedure, named BackupCode, copies the values of the original-protocol variables to the backup set. At each $t_i(n)$, the synchronizer code makes a call to PulseSendCode, and at each $T_i(n)$ the synchronizer calls BackupCode and then to PulseVarCode.

The code of $\alpha'_1$ can be found in Table 5.

Synchronizer $\alpha'_1$ ensures that, for each node $i$ and for every pulse $n$, all messages in $M^i(n)$ are received during $(T_i(n-1), T_i(n))$ and these are the only original-protocol messages that are received during this interval. Each node performs at each $T_i(n)$ the original-protocol pulse code, without sending messages. Thus, the local state at $T_i(n)$ is identical to the local state in the original protocol of node $i$ at time pulse$(n+1)$. Furthermore, the messages sent by node $i$ at $t_i(n)$ are identical to the messages sent in the original protocol by $i$ at time pulse$(n)$, since the messages are built using a replica taken at time $T_i(n-1)$ of the local state at pulse$(n)$. A formal proof which shows that $\alpha'_1$ is a matching synchronizer is presented in the sequel.

We shall prove that $\alpha'_1$ ensures proper operation by using a generalized method, that would help creating other synchronizers that ensure proper operation.

In the following we present and prove a generalization of the method by which synchronizer $\alpha'_1$ was derived from $\alpha_1$.

Let $\text{Syn}$ be a passive synchronizer. Suppose that each (combined) protocol $P'$ created when
Protocol $\alpha'_1$

Messages
CONFIRM – Delivered by the data link when all original-protocol messages sent are confirmed to have been received by the other end of the link.
SAFE – Sent to all neighbors to signal that the node is safe with respect to the current pulse.
READY – Sent by each node to all its neighbors to signal that it knows that it is ready for the next pulse.

Variables
$G_i$: The set of neighbors of $i$.
$Z_i$: A counter, contains the number of the latest pulse performed by node $i$ (a one bit counter is sufficient).
$N_i(l)$: A counter that counts the number of SAFE messages received from neighbor $l$ (a one bit counter is sufficient).
$N'_i(l)$: A counter that counts the number of READY messages received from neighbor $l$ (a one bit counter is sufficient).
$st_i$: 0 — after $t(n)$, not safe yet. 1 — node is safe, but not all its neighbors. 2 — node is ready, but not all its neighbors.

Procedures
BackupCode: copies the local state into the backup set.
PulseVarCode: Contains the original-protocol pulse code with all commands for sending messages erased.
PulseSendCode: Contains the original-protocol pulse code altered to reference only the backup set variables.

Initialization
$Z_i = -1$.
$N_i(l) = 0$, $\forall l \in G_i$.
$N'_i(l) = 0$, $\forall l \in G_i$.
$st_i = 1$.

Algorithm for node $i$
When receiving START, CONFIRM or SAFE or READY from neighbor $l$ do
if SAFE then $N_i(l) \leftarrow N_i(l) + 1$
if READY then $N'_i(l) \leftarrow N'_i(l) + 1$
if CONFIRM then
  $st_i \leftarrow 1$
  send SAFE to all $k \in G_i$
if ($st_i = 1$) and ($\forall j \in G_i$, $Z_i < N_i(j)$) then
  $st_i \leftarrow 2$
call BackupCode
call PulseVarCode
send READY to all $k \in G_i$
if ($st_i = 2$) and ($\forall j \in G_i$, $Z_i < N'_i(j)$) then
  $Z_i \leftarrow Z_i + 1$
  $st_i \leftarrow 0$
call PulseSendCode

Table 5: The $\alpha'_1$ synchronizer
combining some synchronous-protocol $P$ with $Syn$ satisfies the following properties:

1. It generates an additional sequence of clock-pulses $T_i(0), T_i(1), \ldots$ at each node $i$, with the property that for all $n \geq 0$, $t_i(n) \leq T_i(n) \leq t_i(n + 1)$.

2. The original-protocol message sent in the combined protocol by $i$ to neighbor $k$ at time $t_i(n)$ arrives at node $k$ before $T_k(n)$.

3. The original-protocol message sent in the combined protocol by $i$ to neighbor $k$ at time $t_i(n)$ arrives at node $k$ after $T_k(n - 1)$ (assume that $T_k(-1) = t_{init}$ when $t_{init}$ is the time just before the execution of $P'$ begins).

Then one can create a variable-duplicating matching synchronizer $Syn'$ that possesses identical time and communication complexities as $Syn$, by using the same technique that was used to create $\alpha'_1$ from $\alpha_1$.

We shall build the transformation of $Syn'$, given the transformation of $Syn$. The new transformation transforms the input synchronous-protocol $P$ into a (combined) protocol, in the way described below.

First, the transformation of $Syn$ is used to transform the input synchronous protocol into a combined protocol $P'_{temp}$. Then $P'_{temp}$ is transformed into the final combined protocol $P'$ as follows:

A new set of synchronizer variables is added at each node, this set of variables is a copy of the set of the original-protocol variables. We shall call this variables set the backup set. The initialization conditions of the backup set are identical to the initialization conditions of the original synchronous-protocol variables.

Two new procedures are introduced instead of the PulseCode procedure: PulseVarCode and PulseSendCode. Procedure PulseVarCode is a copy of PulseCode, except that the commands for sending messages are erased. Procedure PulseSendCode is a copy of PulseCode, except that every reference to an original-protocol variable is replaced by a reference to its twin in the backup set. Another procedure, named BackupCode, copies at times $T_i(n)$ the original-protocol variables values to the backup set, and then the backup variables are frozen until $t_i(n + 1)$.

The synchronizer code created by the $Syn$ transformation is also changed, each call to PulseCode is replaced by a call to PulseSendCode, and at each $T_i(n)$, a call to BackupCode and then to PulseVarCode are inserted.

**Theorem 7.1** The (combined) protocol $P'$, created when combining a synchronous protocol $P$ with the synchronizer $Syn'$ created as described above, satisfies the matching property.

**Proof:** We first prove the following property by induction on $n$:

$$\forall n \geq 0, \forall i \in V, \Pi_i(T_i(n - 1) - ) = state_i(n)(\Pi_i(0), M^i(0), M^i(1), \ldots, M^i(n - 1))$$
Sec. 7

**base:**

\[ \Pi_i(T_i(-1)-) = \Pi_i(0) = state_i(0)(\Pi_i(0)) \]

**Induction step:**

Assume that the property is true for \( n \). We shall prove it for \( n + 1 \).

From the induction assumption:

\[ \Pi_i(T_i(n-1)-) = state_i(n)(\Pi_i(0), M^i(0), M^i(1), \ldots, M^i(n-1)) \]

At \( T_i(n-1) \), the synchronizer calls procedures BackupCode and PulseVarCode at node \( i \). Procedure PulseVarCode changes the values of the original-protocol variables in the same way the code-block associated with pulses changes them in \( P \).

During \( (T_i(n-1), T_i(n)) \), according to parts \( (a^T) \) and \( (b^T) \) of the two-phase property, all and only the original-protocol messages in \( M^i(n) \) are received. At \( t_i(n) \), procedure PulseSendCode is performed, but this procedure does not reference the original-protocol variables. Therefore, the code performed during \( (T_i(n-1), T_i(n)) \) changes the local-state in exactly the same way as the code performed at node \( i \) when executing the original synchronous-protocol \( P \) during phase\((n)\) in which \( M^i(n) \) is received.

Thus:

\[ \Pi_i(T_i(n)-) = state_i(n+1)(\Pi_i(0), M^i(0), M^i(1), \ldots, M^i(n)) \]

This concludes the proof of the induction step.

In the rest of the proof, \( \Pi_i^T \) denotes the collection of the variables in the backup set.

At each \( T_i(n) \), procedure BackupCode is performed at node \( i \). This procedure copies the values of the original-protocol variables to the backup set.

Hence:

\[ \Pi_i^T(T_i(n)+) = state_i(n+1)(\Pi_i(0), M^i(0), M^i(1), \ldots, M^i(n)) \]

The only two procedures that change \( \Pi_i^T \) are BackupCode and PulseSendCode. These procedures are not called during \( (T_i(n), t_i(n+1)) \), and therefore:

\[ \Pi_i^T(t_i(n+1)-) = state_i(n+1)(\Pi_i(0), M^i(0), M^i(1), \ldots, M^i(n)) \]

The code in PulseSendCode is identical to the original-protocol pulse code except for referencing the backup set variables instead of the original-protocol variables. Therefore, the messages sent by the procedure are a function of \( \Pi_i^T \):

\[ MSG^i_k(n+1) = msg^i_k(\Pi_i^T(t_i(n+1)-)) = msg^i_k(state_i(n+1)(\Pi_i(0), M^i(0), M^i(1), \ldots, M^i(n))) \]
Sec. 7

This concludes the proof of part \((a^*)\) of the matching property.

To prove part \((b^*)\) of the matching property assume that \(\forall n \geq m, M^i(n) = \emptyset\) and:

\[
(*) \quad \text{stt}_i(state_i(m)(\Pi_i(0), M^i(0), M^i(1), \ldots, M^i(m-1))) = \\
\text{state}_i(m)(\Pi_i(0), M^i(0), M^i(1), \ldots, M^i(m-1))
\]

We know that:

\[
\Pi_i(T_i(m-1)) = \text{state}_i(m)(\Pi_i(0), M^i(0), M^i(1), \ldots, M^i(m-1))
\]

Thus, since no messages are received from time \(T_i(m-1)\) on, the only times when \(\Pi_i\) can be changed is when PulseVarCode is called at \(T_i(m), T_i(m+1), \ldots\) but from equation \((*)\) we know that this procedure does not change \(\Pi_i\). Therefore \(t' = T_i(m-1)\) and \((b^*)\) is proved. \(\square\)

Notice, that in synchronizer \(\beta\), parts \((a^T)\) and \((b^T)\) of the two-phase property are satisfied when \(T_i(n) = t_s(n)\), but still part \((T)\) is not satisfied, since \(t_s(n)\) is not generated at each node, and is known only to node \(s\). Therefore the synchronizer \(\beta\) cannot be migrated into a new synchronizer using the technique proved in Theorem 7.1 since the nodes \(i \neq s\) do not know when to perform BackupCode and PulseVarCode.

A good example for a synchronizer that ensures the two-phase property is the SAD synchronizer described in [CCGZ 1].

Since \(\alpha'_1\) is derived from \(\alpha_1\) by using the same technique proved in Theorem 7.1, in order to prove that \(\alpha'_1\) ensures proper operation, we need only to prove that \(\alpha_1\) ensures the two-phase property.

In order to prove this, we shall present a generalization of the method by which synchronizer \(\alpha'_1\) was derived from synchronizer \(\alpha\).

Before discussing this method, we present a new definition. This definition has been used first in [AP 1].

**Definition 7.1** (ready)

Node \(i\) is said to be ready for pulse\((n+1)\), if all messages sent to \(i\) from its neighbors at pulse\((n)\) have already arrived.

If all neighbors of node \(i\) are safe with respect to pulse\((n)\), then node \(i\) is ready for pulse\((n+1)\). The idea of synchronizer \(\alpha_1\) is that a node \(i\) may receive messages belonging to \(M^i(n+1)\) only after having received SAFE messages from all neighbors, thus realizing that it is ready for pulse\((n+1)\).

With a synchronizer in which node \(i\) performs \(t_i(n+1)\) when it is informed that all its neighbors are safe with respect to pulse\((n)\), the two-phase property is not ensured. However, similarly to \(\alpha_1\), a new synchronizer can be constructed, where a node changes its state to ready when it realizes that all its neighbors are safe with respect to some pulse\((n)\). The ready information is propagated in the network in exactly the same way the safe information was formerly propagated. Node \(i\) performs \(t_i(n+1)\) when it realizes that it and all its neighbors are ready for pulse\((n+1)\).
A synchronizer created using this technique ensures the two-phase property. $(T)$ is ensured since at each node $i$, $T_i(n)$ is defined as the $(n+1)$-st time when node $i$ changes its state to ready. $(a^T)$ is ensured since at $T_i(n)$, each neighbor $j$ of node $i$ is already safe with respect to $t_j(n)$. $(b^T)$ is ensured since each neighbor $j$ of $i$ performs $t_j(n + 1)$ only after node $i$ performs $T_i(n)$.

For example, in synchronizer $\alpha$, when receiving all confirmations for the original-protocol messages sent at a given pulse, each node sends a SAFE message to all its neighbors. After receiving SAFE messages from all its neighbors, each node performs the next pulse. In synchronizer $\alpha_1$, when a node receives SAFE messages from all its neighbors, it merely changes its state to ready and sends READY messages to all its neighbors. Only after having received READY messages from all its neighbors, which means that all its neighbors know that they are ready, a node performs the next pulse.

The general method proposed in this section will be used to design synchronizer $\gamma_1'$ in Sec. 7.2. This method can be also used to derive from $\beta$ a synchronizer that ensures proper operation. We shall not discuss this synchronizer, since synchronizers $\epsilon_1$, $\epsilon_2$, $\epsilon_3$ presented in sections 6.1 and 6.2 have the same communication and time complexities, but do not need extra memory at each node and are also simpler.

### 7.2 Synchronizer $\gamma_1'$

In synchronizer $\gamma$ [Awer 1], an initialization phase creates a partition of the network into clusters. The partition is defined by a spanning forest of the network. Each tree in the forest defines a cluster of nodes. Between each two neighboring clusters, one preferred link is selected. Inside each cluster there is a leader node. We say that a cluster is safe if all nodes in the cluster are safe.

SAFE messages are forwarded down along each tree in the forest. When a cluster leader knows that all nodes in the cluster are safe, it broadcasts this knowledge along the tree by using CLUSTER_SAFE message. Each node forwards this message to all its sons and along all adjacent preferred links.

The next phase determines the time at which all neighboring clusters are known to be safe. In this phase, each leaf node in the cluster sends a NEIGHBORS_SAFE message to its father after having received a CLUSTER_SAFE message on all adjacent preferred links. Each intermediate node sends a NEIGHBORS_SAFE message to its father after having received a NEIGHBORS_SAFE message from all its sons and CLUSTER_SAFE from all adjacent preferred links. At the end of this process, the leader node knows that all neighboring clusters are also safe.

In synchronizer $\gamma$, the leader node of each cluster knows at this stage that all nodes in the cluster are ready (all nodes in the cluster and in the neighboring clusters are safe), so it broadcasts an AWAKE message over the cluster tree, causing all nodes in the cluster to perform the next phase. As with $\beta$, message delaying cannot be performed on $\gamma$, since a node cannot distinguish early messages from timely ones.

Synchronizer $\gamma_1$ is a version of $\gamma$ created by using the method described in Sec. 7.1. In syn-
chronizer $\gamma_1$, when the leader node knows that all nodes in the cluster are ready, it propagates this knowledge to all nodes in the cluster and to the neighboring clusters. This is done by broadcasting a CLUSTER_READY message along the tree. Each node forwards the message to all its sons and to all adjacent preferred links.

The next phase determines the time at which all neighboring clusters are known to be ready. In this phase, each leaf node in the cluster sends a NEIGHBORS_READY message to its father after having received a CLUSTER_READY message on all adjacent preferred links. Each intermediate node sends a NEIGHBORS_READY message to its father after having received a NEIGHBORS_READY message from all its sons and a CLUSTER_READY message from all adjacent preferred links. At the end of this process, the leader node knows that all the nodes in the neighboring clusters are also ready.

Now, the leader node can perform the next pulse and also can broadcast an AWAKE message over the cluster tree, causing all nodes in the cluster to perform the next pulse.

Synchronizer $\gamma_1$ is transformed into synchronizer $\gamma_1'$ that ensures the matching property by using the method proved in Theorem 7.1.

Obviously, the method proved in Theorem 7.1 doubles the required memory at each node. Recall that we use $\gamma_1'$ and $\alpha_1'$ when message-delaying is not allowed. On the other hand, one of the reasons for not using message-delaying is saving memory. The amount of memory needed for using message delaying is equal to the number of links connected to each node times the length of the largest message. Therefore, when deciding between message-delaying on one hand and variable-duplicating on the other, one should be compare that amount of memory to the one needed to backup the local state.

In many cases, synchronizers like $\gamma_1'$ and $\alpha_1'$ still save memory, because the length of the messages may be as large as the local memory size or because the maximum number of links connected to each node is large or unknown. Several optimizations may be used in order to make the memory overhead of $\alpha_1'$ and $\gamma_1'$ even smaller. One optimization is, for example, saving only the variables needed for creating and sending messages. For protocols in which the same message is sent to many neighbors, another optimization may be to save the message to be sent instead of the local state.

The synchronizer for hypercube topology presented in [PU 1] is another passive synchronizer that does not ensure part (b) of the timely property. This synchronizer can be changed to ensure proper operation by using the same technique.
8 Message delaying synchronizers

Synchronizers \( \alpha, \beta \) and \( \gamma \) [Awer 1] are passive synchronizers that ensure part (a) of the \textit{timely} property, but not part (b). According to Theorem 4.2, this means that these synchronizers do not ensure proper operation. As seen in Sec. 4.2, the problem is that messages may arrive too early at the respective neighbor. A way to solve this problem is by not processing such messages immediately when they are received. When a message is received too early, it is saved in a buffer, and at a later time, it is taken out of this buffer and processed. For example, a message sent by node \( j \) to node \( i \) at \( t_j(n) \), should be processed at node \( i \) only after \( t_i(n) \). If the message is indeed received after that time, it is said to be \textit{timely} and should be processed immediately. If the message is, however, received before that time, in which case it is said to be early, it is saved in a buffer and not processed. Immediately after \( t_i(n) \), the early message can be taken out of the buffer and processed. This idea has been suggested in [LT 1], [FLS 1], [ER 1], [Awer 1].

The problem with this solution is that in some cases, message delaying (saving messages in order to process them at a later time) is not implementable, and in many others, it requires a very high price in terms of memory, time, etc. On the other hand, the method has the advantages of being simple and intuitive.

In Sec. 2 we have defined two versions of the asynchronous model, the \textit{non-delaying} version, and the \textit{delaying} version. Message delaying synchronizers can be implemented only in the latter.

In Sec. 8.1 and 8.2, we show that this remedy works with synchronizer \( \alpha \), but cannot be directly applied to synchronizers \( \beta \) and \( \gamma \). The problem is that in the latter, when a node receives an original-protocol message, it may not be able to distinguish whether the message is a \textit{timely} message and should be processed immediately or is an \textit{early} one and should be saved to be processed at a later time.

In sections 8.2, 8.3, 8.4 and Sec. 8.5 we suggest several solutions for this problem, and in Sec. 8.6 we prove that all these solutions ensure proper operation, i.e. ensure the matching property.

### 8.1 Synchronizer \( \alpha^* \)

In synchronizer \( \alpha \) [Awer 1], a node \( i \) is said to be 'safe' with respect to \( t_i(n) \), if all messages of the original protocol sent by \( i \) at \( t_i(n) \) have already been received by the respective neighbors. When a node learns that it is safe, it sends a SAFE message to all its neighbors. Node \( i \) generates \( t_i(n + 1) \) when it learns that all its neighbors are safe.

Synchronizers \( \beta \) and \( \gamma \) [Awer 1] work somewhat differently, but still a node performs a pulse when it learns that all its neighbors are safe with respect to the former pulse.

**Lemma 8.1** The (combined) protocol created by combining any synchronous protocol with one of the synchronizers \( \alpha, \beta \) or \( \gamma \) satisfies the property that the original-protocol message sent by \( i \) to neighbor \( k \) at time \( t_i(n) \) arrives at node \( k \) before \( t_k(n + 1) \) and after \( t_k(n - 1) \).
Sec. 8

Proof: When node \( k \) performs \( t_k(n+1) \), its neighbor \( i \) is safe with respect to \( t_i(n) \), which means that the message sent by \( i \) to \( k \) at \( t_i(n) \) had already been received. On the other hand, when node \( i \) performs \( t_i(n) \) and sends a message to \( k \), node \( k \) is safe with respect to \( t_k(n-1) \), which means that \( t_k(n-1) < t_i(n) \).

If a combined protocol satisfies the property in Lemma 8.1, a message sent by a node \( i \) to a neighbor \( k \) and received during \((t_k(n),t_k(n+1))\), was sent at time \( t_i(n) \) or \( t_i(n+1) \). In other words, the message belongs to \( M^k(n) \) or \( M^k(n+1) \). However, if indeed a message sent at \( t_i(n+1) \) is received by \( k \) during \((t_k(n),t_k(n+1))\) and is processed, the combined protocol does not behave as the original protocol since in the latter, all messages sent at \( t_i(n+1) \) arrive during \((t_k(n+1),t_k(n+2))\). It was shown in [LT1] that such misbehavior can occur with each of the synchronizers \( \alpha, \beta, \gamma \).

In order to implement \( \alpha^* \), which is the message delaying version of \( \alpha \) that ensures proper operation, we need a fifo of messages at each node.

When an original-protocol message is received too early at some node, the message is pushed into the fifo. The fifo is emptied by the node processor immediately after performing the next pulse and each message that is popped out of the fifo is processed as if it had just arrived.

From Lemma 8.1, the original-protocol messages received by a node \( i \) during \((t_i(n),t_i(n+1))\) belong either to \( M^i(n) \) or to \( M^i(n+1) \). Since the messages of \( M^i(n) \) received during that interval are processed immediately, the messages that are in the fifo at time \( t_i(n+1) \) belong to \( M^i(n+1) \). These messages are processed only after \( t_i(n+1) \), as if they were received in the correct interval.

In synchronizer \( \alpha^* \), it is easy to recognize whether a message belongs to \( M^i(n) \) or to \( M^i(n+1) \), since in each interval, nodes send SAFE messages to all neighbors when they are safe. Thus, a node \( i \) can decide whether a message belongs to \( M^i(n) \) or to \( M^i(n+1) \) by simply counting the number of SAFE messages received from that neighbor. Since there are only two possibilities, a one bit counter is sufficient for this purpose.

The code of \( \alpha^* \) is presented in Table 6.

8.2 Synchronizer \( \beta^* \)

In synchronizer \( \beta \), an initialization phase creates a directed tree with a root node \( s \). When recognizing that itself as well as all nodes in its sub-tree are safe, each node sends a SAFE message to its father. When the root node \( s \) finds out that all nodes in the network are safe, it broadcasts an AWAKE message along the tree. This message informs the nodes that they should perform the next pulse.

As shown below, a modification similar to the one of Sec. 8.1 does not work on \( \beta \). The reason is that a node may not be able to distinguish between timely and early messages. Therefore, the node cannot decide whether to process this message immediately or to push it into the fifo.

We shall prove this claim by introducing two execution examples, of two different protocols combined with synchronizer \( \beta \), on the same network. In both executions, node \( i \) receives identical messages in identical order. In both executions, node \( i \) receives during \((t_i(n),t_i(n+1))\) one original-
Protocol $\alpha^*$

**Messages**

CONFIRM - Delivered by the data link when all original-protocol messages sent are confirmed to have been received by the other end of the link.

SAFE - Sent to all neighbors to signal that the node is safe with respect to the current pulse.

**Variables**

- $G_i$: The set of neighbors of $i$.
- $Z_i$: A counter, contains the number of the latest pulse performed by node $i$ (a one bit counter is sufficient).
- $N_i(l)$: A counter that counts the number of SAFE messages received from neighbor $l$ (a one bit counter is sufficient).
- $st_i$: 1 when all original-protocol messages sent at the last pulse are confirmed, 0 otherwise.

**Procedures**

- $msgCode$: Contains the original-protocol message code for processing messages of type $msg$.
- $PulseCode$: Contains the original-protocol pulse code.

**Initialization**

- $Z_i = -1$.
- $N_i(l) = 0$, $\forall l \in G_i$.
- $st_i = 1$.

**Algorithm for node $i$**

When receiving an original-protocol message named $msg$ from neighbor $l$

```
if $Z_i = N_i(l)$ then ; (timely message)
call msgCode ($msg$, $l$)
else push ($msg$, $l$) ; (early message)
```

When receiving START, CONFIRM or SAFE from neighbor $l$ do

```
if SAFE then $N_i(l) \leftarrow N_i(l) + 1$
if CONFIRM then
    $st_i \leftarrow 1$
    send SAFE to all $k \in G_i$
if ($st_i = 1$) and ($\forall j \in G_i$, $Z_i < N_i(j)$) then
    $Z_i \leftarrow Z_i + 1$; $st_i \leftarrow 0$
call PulseCode
while not fifo_empty do
    pop ($message$, $neighbor$)
call msgCode ($message$, $neighbor$) ; (where $msg$ is the type of $message$)
```

Table 6: The $\alpha^*$ synchronizer
protocol message from neighbor $k$. Yet, in one execution, this message belongs to $M^i(n)$ and in the other, it belongs to $M^i(n+1)$. The two execution examples take place in the network shown in Fig. 4a) with the spanning tree of Fig. 4b) (Very similar examples, designed for synchronizer $\epsilon_1$, have been given in Sec. 6.1).

In both examples, node $s$ performs $t_s(0)$ and sends an AWAKE message to nodes $i$ and $k$. No original-protocol messages are sent by $s$ at $t_s(0)$. Nodes $i$ and $k$ receive the message at approximately the same time and perform $t_i(0)$ and $t_k(0)$ respectively. In the first example, node $k$ sends at time $t_k(0)$ an original-protocol message to node $i$. Node $i$ has no original-protocol messages to send at time $t_i(0)$. Thus, node $i$ sends immediately a SAFE message, and after a while it receives the original-protocol message sent by node $k$.

In the second example, both $i$ and $k$ have no original-protocol messages to send at times $t_i(0)$ and $t_k(0)$ respectively. Thus, they send SAFE messages to $s$ immediately after performing pulse 0. When the two SAFE messages are received by $s$, the latter sends an AWAKE message to $i$ and $k$, while performing $t_s(1)$. The AWAKE message to $i$ is delayed. When the AWAKE message to $k$ arrives, the latter performs $t_k(1)$ and sends an original-protocol message to node $i$. This message arrives at node $i$ before the arrival of the AWAKE message from node $s$ and thus before $t_i(1)$.

In both executions, node $i$ receives the same messages, in the same order: an AWAKE message from node $s$, and then an original-protocol message from node $k$. But, in the first example, the message belongs to $M^i(0)$ and should be processed immediately, while in the second, it belongs to $M^i(1)$ and should be delayed.

The same problem occurs in synchronizer $\gamma$ and in the synchronizer for hypercube topology presented in [PU 1]. In fact, it will occur in any synchronizer that does not require nodes to send synchronizer messages at each pulse to all neighbors, unless some other mechanism to identify early messages is employed. Synchronizer $\beta_1$, to be introduced next, constructs such a mechanism by
Sec. 8
requiring two waves in each direction on the tree, as opposed to one wave in $\beta$. In this way, each node will be given a time $t_i^{\text{READY}}(n)$ between $t_i(n)$ and $t_i(n+1)$, such that all messages that arrive before $t_i^{\text{READY}}(n)$ are timely messages, i.e. belong to $M^i(n)$, and those arriving after $t_i^{\text{READY}}(n)$ are early messages, i.e. belong to $M^i(n+1)$.

Synchronizer $\beta_1$ needs the same initialization as $\beta$, i.e. a spanning tree, with root $s$ say, for the network graph. The messages used by synchronizer $\beta_1$ are SAFE, READY, and AWAKE. A SAFE message is sent by each node to its father when all nodes in its sub-tree, including itself, are safe. When node $s$ finds out that all nodes in the network are safe, it starts a PIF of READY messages along the tree [Seg 1]: a node sends a READY message to all its neighbors in the tree except its father when it receives a READY message from its father, and sends a READY message to its father when it has received READY messages from all its neighbors in the tree. When node $s$ receives a READY message from all neighbors in the tree, it knows that all nodes in the network have received the READY message, and it broadcasts an AWAKE message on the tree. This message tells the nodes to perform the next pulse.

**Lemma 8.2** When performing $\beta_1$, a node $i$ that receives an original-protocol message during $(t_i(n), t_i(n+1))$ can decide whether this message belongs to $M^i(n)$ or to $M^i(n+1)$.

**Proof:** Let $t_i^{\text{READY}}(n)$ be the first time after $t_i(n)$ when node $i \neq s$ receives a READY message. This message was sent by the father of node $i$ in the tree. We shall show that all original-protocol messages received by node $i$ during $(t_i(n), t_i^{\text{READY}}(n))$ belong to $M^i(n)$ and those received during $(t_i^{\text{READY}}(n), t_i(n+1))$ belong to $M^i(n+1)$.

To prove this, assume that the original-protocol message is received by node $i$ before $t_i^{\text{READY}}(n)$. This means that node $s$ has not performed $t_s(n+1)$ yet. Since $s$ is the first to perform every pulse, no message belonging to $M^i(n+1)$ had been sent yet, and thus the message belongs to $M^i(n)$. If the original-protocol message is received after $t_i^{\text{READY}}(n)$, then all nodes in the network are safe with respect to the $n$-th pulse. Thus, the message does not belong to $M^i(n)$.

When node $s$ receives an original-protocol message during $(t_s(n), t_s(n+1))$, the message belongs to $M^s(n)$, because no node performs pulse $(n+1)$ before node $s$ does. Therefore, the statement of the Lemma is ensured at node $s$ in a trivial way, and no message-delaying mechanism is needed at node $s$.

The message-delaying version of $\beta_1$ is referred to as $\beta_1^*$. In the latter, an original-protocol message received after $t_i^{\text{READY}}(n)$ and before $t_i(n+1)$ is placed in a fifo and the messages in the fifo are processed immediately after $t_i(n+1)$. The communication complexity of $\beta_1^*$ is $4|V|$ and the time complexity is $2D$ — both double the complexity of $\beta$.

### 8.3 A generalization

The method described in Sec. 8.2 can be generalized to other synchronizers that have the problem of distinguishing between timely and early messages. The method works for synchronizers that
Sec. 8

use SAFE messages to inform nodes that all their neighbors are safe, but do not necessarily send SAFE messages to all neighbors. The same generalized method has been used for other purposes in Sec. 7.1.

Recall that a node $i$ is said to be ready for pulse($n + 1$), if all messages sent to $i$ from its neighbors at pulse($n$) have already arrived.

If all neighbors of node $i$ are safe with respect to pulse($n$), then node $i$ is ready for pulse($n + 1$). The idea of synchronizer $\beta^*_1$ is that a node $i$ may receive messages belonging to $M^t(n + 1)$ only after having received a READY message that informs the node that it is ready for pulse($n + 1$).

As indicated in Sec. 8.2, with a synchronizer in which node $i$ performs $t_i(n + 1)$ when it is informed that all its neighbors are safe with respect to pulse($n$), a node cannot distinguish between timely and early messages unless SAFE messages are sent to all neighbors. However, similarly to Sec. 8.2, a new synchronizer can be constructed, where a node changes its state to ready when it realizes that all its neighbors are safe with respect to some pulse($n$). The ready information is propagated in the network in exactly the same way the safe information was formerly propagated. Node $i$ performs $t_i(n + 1)$ when it realizes that it and all its neighbors are ready for pulse($n + 1$).

This technique solves the problem, since a node that receives an original-protocol message before changing its state to ready knows that this message was sent at the current pulse (no node performs the next pulse before all its neighbors know that they are ready). If the message is received when the node is in ready state, it knows that the message was sent at the later pulse.

For example, in synchronizer $\beta$, SAFE messages are forwarded downtree to node $s$, and at a finite time, node $s$ knows that all nodes in the network are safe. At that time, node $s$ broadcasts this knowledge along the tree by sending messages that cause the nodes to perform the new pulse. In synchronizer $\beta_1$, these messages only cause the nodes to realize that they are ready. The fact that the nodes know they are ready is forwarded downtree to node $s$, and at the end of this stage, node $s$ knows that all nodes in the network know that they are ready. At that time, node $s$ broadcasts this knowledge along the tree by sending messages that cause the nodes to perform the new pulse.

The general method proposed in this section will be used to design synchronizer $\gamma^*_1$ in Sec. 8.5.

### 8.4 Synchronizers $\beta^*_2, \beta^*_3$

A different method for solving the problem of distinguishing between timely and early messages has been suggested for the SOS synchronizer presented in [CGZ 1] and for dealing with synchronizers in dynamic networks in [AS 1]. The idea is that a suffix is added to each original-protocol message, containing the number of the pulse at which the message is sent. In [FLS 1], the assumption that each original-protocol message contains the number of the pulse is included in the synchronous model. This idea is mentioned also in [Awer 1] for use in synchronizer $\gamma$.

When implementing this scheme with a synchronizer that has the property in Lemma 8.1, a one-bit suffix is sufficient, since the messages received by node $i$ during $(t_i(n), t_i(n + 1))$ belong either to $M^t(n)$ or to $M^t(n + 1)$. The implementation of this scheme for $\beta$ will be referred to as $\beta^*_2$. 

Note that synchronizer $\beta_2^*$ can be implemented only if alteration of the original-protocol messages is allowed.

Synchronizer $\beta_3^*$ is another implementation, that does not require changes in the format of the original-protocol messages. Yet, the communication complexity of combined-protocols created with $\beta_3^*$ may be up to twice the complexity achieved with $\beta_1^*$ or $\beta_2^*$, depending on the complexity of the original protocol. The idea of $\beta_3^*$ is that each node sends two messages for each original-protocol message. The first is a synchronizer message named PULSE_NUM, that contains the number of the pulse at which the message is sent. Immediately afterwards, it sends the original-protocol message. Each node maintains an array. The array at node $i$ is $\text{pulse.num}_i(l), \forall l \in G_i$. When a PULSE_NUM message is received from neighbor $k$, node $i$ sets $\text{pulse.num}_i(k)$ as the pulse number contained in this PULSE_NUM message. When the original-protocol message is received from neighbor $k$ during $(t_i(n), t_i(n + 1))$, the message is processed immediately if $n = \text{pulse.num}_i(k)$ and is delayed otherwise.

### 8.5 Synchronizers $\gamma_1^*, \gamma_2^*, \gamma_3^*$

In synchronizer $\gamma$ [Awer 1], an initialization phase creates a partition of the network into clusters. The partition is defined by a spanning forest of the network. Each tree in the forest defines a cluster of nodes. Between each two neighboring clusters, one preferred link is selected. Inside each cluster there is a leader node. We say that a cluster is safe if all nodes in the cluster are safe.

SAFE messages are forwarded down along each tree in the forest. When a cluster leader knows that all nodes in the cluster are safe, it broadcasts this knowledge along the tree by using CLUSTER_SAFE message. Each node forwards this message to all its sons and along all adjacent preferred links.

The next phase determines the time at which all neighboring clusters are known to be safe. In this phase, each leaf node in the cluster sends a NEIGHBORS_SAFE message to its father after having received a CLUSTER_SAFE message on all adjacent preferred links. Each intermediate node sends a NEIGHBORS_SAFE message to its father after having received a NEIGHBORS_SAFE message from all its sons and CLUSTER_SAFE from all adjacent preferred links. At the end of this process, the leader node knows that all neighboring clusters are also safe.

In synchronizer $\gamma$, the leader node knows at this stage that all the nodes in the cluster are ready (all nodes in the cluster and in the neighboring clusters are safe), so it broadcasts an AWAKE message over the cluster tree, causing all nodes in the cluster to perform the next phase. As with $\beta$, message delaying cannot be performed on $\gamma$, since a node cannot distinguish early messages from timely ones.

Synchronizer $\gamma_1^*$ is a version of $\gamma$ created by using the method described in Sec. 8.3 (synchronizer $\gamma_1^*$ has also been described in Sec. 7.2). In synchronizer $\gamma_1^*$, when the leader node knows that all nodes in the cluster are ready, it propagates this knowledge to all nodes in the cluster and to the neighboring clusters. This is done by broadcasting a CLUSTER_READY message along the tree.
Each node forwards the message to all its sons and to all adjacent preferred links.

The next phase determines the time at which all neighboring clusters are known to be ready. In this phase, each leaf node in the cluster sends a NEIGHBORS_READY message to its father after having received a CLUSTER_READY message on all adjacent preferred links. Each intermediate node sends a NEIGHBORS_READY message to its father after having received a NEIGHBORS_READY message from all its sons and a CLUSTER_READY message from all adjacent preferred links. At the end of this process, the leader node knows that all the nodes in the neighboring clusters are also ready.

Now, the leader node can perform the next pulse and also can broadcast an AWAKE message over the cluster tree, causing all nodes in the cluster to perform the next pulse.

Synchronizer \( \gamma^* \) is derived from \( \gamma_1 \) in the same way synchronizer \( \alpha^* \) was derived from \( \alpha \) in Sec. 8.1 and \( \beta^*_1 \) was derived from \( \beta_1 \) in Sec. 8.2.

Synchronizers \( \gamma^2 \) and \( \gamma^3 \) are derived from synchronizer \( \gamma \) in exactly the same way that \( \beta^2 \) and \( \beta^3 \) are derived from synchronizer \( \beta \).

It should be pointed out that synchronizer \( \gamma \) presented in [Awer 1] uses sequence numbers, while the message delaying version of \( \gamma \) discussed in [LT 1] does not.

The same three techniques can be used for solving the problem for the hypercube-topology synchronizer presented in [PU 1].

### 8.6 A correctness proof for message delaying synchronizers

We shall now prove that all message delaying synchronizers presented in the last sections ensure proper operation.

Let Syn be a passive synchronizer that ensures:

a) The original-protocol message sent in the combined protocol by \( i \) to neighbor \( k \) at time \( t_i(n) \) arrives at node \( k \) before \( t_k(n + 1) \).

b') The original-protocol message sent in the combined protocol by \( i \) to neighbor \( k \) at time \( t_i(n) \) arrives at node \( k \) after \( t_k(n - 1) \).

c') When an original-protocol message that was sent by node \( i \) arrives at node \( k \) during \((t_k(n), t_k(n + 1))\), the latter can distinguish, based on the local synchronizer-variables alone, if this message belongs to \( M^k(n) \) or to \( M^k(n + 1) \).

Note that since Syn is a passive synchronizer, it cannot use the knowledge indicated in (c') in order to delay early messages. However, Syn can be transformed into a message-delaying matching synchronizer Syn* which works in the following way:

The synchronizer protocol used in Syn* is identical to the one used in Syn, except that received original-protocol messages are treated in a special way.
When an original-protocol message received at some node is *early*, the message is pushed into the fifo. The fifo is emptied by the node processor immediately after performing the next pulse and each message that is popped out of the fifo is processed as if it had just arrived.

The processor at each node is able to decide whether a message is received too early, since a message from $M^i(n)$ received during $(t_i(n), t_i(n+1))$ is a timely message, while a message from $M^i(n+1)$ received during the same interval is an early message. From (a) and (b'), no other messages can be received during $(t_i(n), t_i(n+1))$.

**Theorem 8.3** The synchronizer $Syn^*$, created as described above, is a *matching* synchronizer and thus, ensures proper operation.

**Proof:** In order to prove the theorem, we shall prove that a combined protocol $P'$, created by combining a synchronous protocol $P$ with $Syn^*$, satisfies the *matching* property. This is done by proving the following properties by induction on $n$:

i) $\forall n \geq 0, \forall i \in V, \Pi_i(t_i(n)-) = state_i(n)(\Pi_i(t_i(0))- , M^i(0), M^i(1), \ldots, M^i(n-1))$

ii) $\forall n \geq 0, \forall i \in V$, at time $t_i(n)-$, the fifo contains all messages in $M^i(n)$ that arrived at node $i$ before $t_i(n)$, and in the order they were received.

**base :**
All the original-protocol messages received before $t_i(0)$ are pushed into the fifo since they are early messages. Therefore, none of these messages changes the local state of $i$:

$$\Pi_i(t_i(0)-) = \text{initial state of } i = state_i(0)(\Pi_i(t_i(0))-)$$

This proves (i) for $n = 0$.

As has been stated before, messages received before $t_i(0)$ belong to $M^i(0)$. This proves (ii) for $n = 0$.

**induction step :**
Assume that (i) and (ii) are true for $n$. We shall prove (i) and (ii) for $n+1$.

Since only the original code of $P$ changes the local-state, we shall track the flow of this code only.

From the induction assumption :

$$\Pi_i(t_i(n)-) = state_i(n)(\Pi_i(t_i(0))- , M^i(0), M^i(1), \ldots, M^i(n-1))$$

On time $t_i(n)$, PulseCode is performed, and then all messages of $M^i(n)$ that were received before $t_i(n)$ are popped out of the fifo, and the corresponding code-blocks are performed. This is done in the same order in which these messages were received. During $(t_i(n), t_i(n+1))$ the rest of the messages of $M^i(n)$ are received and the corresponding code-blocks are performed.
All the messages not in $M^i(n)$, received during $(t_i(n), t_i(n+1))$ are pushed into the fifo and do not affect the local-state. Therefore, the code performed during $(t_i(n), t_i(n+1))$ is identical to the code performed at node $i$ when executing the original synchronous-protocol $P$ during phase($n$) in which $M^i(n)$ is received.

Thus: $\Pi_i(t_i(n+1)-) = state_i(n+1)(\Pi_i(t_i(0))-), M^i(0), M^i(1), \ldots, M^i(n)$

and claim (i) is proved for $n+1$.

As described before, the fifo is emptied at $t_i(n)$. During $(t_i(n), t_i(n+1))$ only messages of $M^i(n)$ and of $M^i(n+1)$ can be received by node $i$. Messages of $M^i(n)$ are processed immediately, and messages of $M^i(n+1)$ are pushed into the fifo. This proves (ii) for $n+1$.

parts (a*) and (b*) of the matching property follow directly from (i).
References


