A SYNCHRONIZER WITH LOW MEMORY OVERHEAD

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

September 7, 1993

Abstract

A new message-delaying version of synchronizer $\gamma$, named $\zeta$, is presented. Synchronizer $\zeta$ ensures that original-protocol messages received by a node from nodes in the same cluster are never early, and thus, no buffers for their temporary storage are necessary. Only original-protocol messages on edges leading to nodes of other clusters (external edges) may be early. The $z$-partition algorithm is introduced to reduce the number of external edges connected to each node, thus reducing the memory overhead of $\zeta$. For an arbitrary $z$, this algorithm ensures that the external degree of each node is no more than $\left\lfloor \frac{|E|}{|V|} \right\rfloor - 1$. The $z$-partition algorithm increases the time complexity of $\zeta$ to $O(z + \log_k |V|)$ per pulse. We prove that the tradeoff between memory overhead and time complexity achieved by the $z$-partition algorithm is optimal.

*The work of A. Segall was supported by the Fund for the Promotion of Research at the Technion.
1 Introduction

This paper deals with distributed protocols in two network models: the synchronous model and the asynchronous model. In the asynchronous model, nodes perform operations only upon receiving a message from some neighbor or from the outside world. At that time, the node processes the message, performs local computations, and may send messages to some or all of its neighbors. All local actions are performed atomically. Messages sent by a node to any of its neighbors are received in a FIFO order within a finite undetermined time. Messages are processed at each node in the received order, even if they were received from different links. The contents of any received message is not available to the code activated by later received messages, unless saved in the local variables when received.

The communication complexity of an asynchronous protocol for a network $N$ is defined as the maximum number of messages sent during an execution of the protocol. The time complexity of an asynchronous protocol is defined as the largest time needed to complete an execution of the protocol, assuming the message delay over all links in the network is bounded by one time unit.

The synchronous model assumes that all link delays are bounded by some quantity referred to as a time unit. Pulses are generated synchronously at all nodes in the network at time unit intervals. Messages are sent only at pulse ticks, and thus arrive at the destination node before the next pulse. Operations are performed by a node only at the time of a pulse or when receiving a message. When a node receives a message, it processes the message and performs local computations. At the time of a pulse, the node may perform local computations and in addition, it may send messages to some or all of its neighbors. All local actions are performed atomically.

This synchronous model is message-and-pulse driven, meaning that the messages are processed only upon arrival, and are not available to the code at any later time. In particular, the contents of any received message is not available to the code at the time of a pulse, unless saved in the local variables when received. The message-and-pulse driven model is used in previous works concerning synchronizers, like [Awer 1], [Awer 2], [LT 1], [PU 1], [AP 1], [LTC 1], [AIR 1], [ShS 1], [ShS 2], [ShS 3].

We also assume that in a synchronous network, at time pulse($n$), each node sends at most one message to each neighbor. This assumption 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], [ShS 1], [ShS 2], [ShS 3]), and is reasonable because multiple sent messages can be simulated by packing them into one message.

Synchronizers are tools for transforming protocols written for the synchronous model into protocols that run on an asynchronous network. The synchronous protocol will be referred to as the original protocol. The asynchronous protocol created by the synchronizer generates a sequence of pulses at each node. The pulses occur asynchronously at different nodes. At each pulse, the nodes perform the original-protocol pulse code and send messages which are identical to the original-protocol messages. In certain circumstances, slight changes in the pulse code and/or in messages
are allowed (see [ShS 1]).

The methodology of synchronizers was introduced in [Awer 1], where three synchronizers were presented: the $\alpha$ synchronizer, with an overhead of $O(|E|)$ in communication complexity and $O(1)$ in time complexity per pulse, the $\beta$ synchronizer with an 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, which 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], [KGZ 1], [ER 1], [ER 2], [RS 1], [AS 1], [ShS 1] and [ShS 2]. Applications of synchronizers can be found in [Awer 2], [AP 1], [SM 1] and [LTC 1]. Other aspects of synchronizers were studied in [AIR 1], [AIJJR 1], [MR 1] and [ShS 3].

All synchronizers ensure that a node may perform a new pulse when it knows that it has received all original-protocol messages sent to it by its neighbors at the former pulse. However, most 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. On the other hand, in a synchronous model, the nodes perform the pulse simultaneously, and every message sent at a certain pulse arrives at the neighbor node after the pulse has been performed. Thus, if no special care is taken, the simulation of the synchronous algorithm, created by using a synchronizer, may allow erroneous executions. We refer to original-protocol messages which are not early as timely messages.

One way to solve this problem is suggested in [LT 1], [FLS 1], [ER 1], [Awer 1], [KGZ 1], [ShS 1], [ShS 2] and is referred to as message delaying. The idea is that when a message is early, it is not processed immediately. Instead, such messages are stored in memory and processed only after the pulse is performed at the node. This method is intuitive and simple to implement. It requires buffering at most one message per link at a time, and thus the required number of buffers is equal to the degree of the node. The size of each buffer must correspond to the longest possible original-protocol message. The purpose of this paper is to investigate ways to reduce this amount of memory.

In Sec. 3 we present synchronizer $\zeta$, which is a new version of synchronizer $\gamma$. When using this synchronizer, the edges connected to each node are divided into two groups, one on which early messages may be received and one on which early messages are never received.

In synchronizers $\gamma$ and $\zeta$, the network is partitioned into clusters. Edges that connect two nodes in the same cluster are referred to as ‘internal edges’. Edges that connect nodes belonging to different clusters are called ‘external edges’. Synchronizer $\zeta$ ensures that original-protocol messages received from internal edges are timely, so that early original-protocol messages may be received only from external edges. Thus, the number of buffers that need to be reserved at each node for early messages is the external degree of the node, namely the number of external edges connected to the node. If the partition of the network is such that the external degree of the nodes is minimized, the required number of buffers is minimized too.

In [ShS 3], we introduce the notion of memory overhead of synchronizers. The memory overhead
of a synchronizer for a network $N$ and a synchronous protocol $P$ is defined as the maximum over the
nodes in $N$ of the memory needed for the synchronizer variables and buffers when combined
with $P$. The memory overhead of a synchronizer is expressed by using the following parameters:

- $S$: the maximum over the nodes in the network, of the amount of memory used for storing the
  synchronous-protocol variables.

- $m$: the length of the largest original-protocol message.

- Network topology parameters: $|V|$, $|E|$, the maximum node degree over the network $d$, etc.

In [ShS 3], we discuss the known synchronizers and their memory overhead.

The memory overhead of message-delaying synchronizers is composed of two parts: the memory
required for the synchronizer protocol and the memory needed for saving the delayed messages.
The maximum number of messages that may be delayed at the same time for any synchronizer is
$(d-1)$. Thus, the amount of memory needed for saving delayed messages is $(d-1)m$. The amount
of memory needed for the synchronizer protocol in $\gamma$ is $O(d)$, thus the memory overhead of the
message-delaying version of $\gamma$ is $(d-1)m + O(d)$.

In this paper we assume that the maximum degree of a node in a network of $|V|$ nodes is
bounded only by $|V| - 1$. This assumption is not always true, since in many cases each node has a
bounded number of links that can be connected to it. However, in such cases, $|E| = O(|V|)$ and thus
synchronizer $\alpha$ is a better choice than $\gamma$ or $\zeta$, since it allows $O(|V|)$ communication and $O(1)$ time
complexity for such cases. When assuming that $d$ is bounded only by $|V|$, the memory overhead of
synchronizer $\gamma$ is bounded by $(|V| - 1)m + O(|V|)$. Observe that for some original-protocols
(e.g., the synchronous protocol presented in [ShS 3]), $m$ can be as large as $S$. For such cases, the
memory overhead of $\gamma$ is $(|V| - 1)S + O(|V|)$.

As shown in Sec. 4.1, if $\zeta$ uses the original partition algorithm of $\gamma$ [Awer 1], the memory
overhead is the same as in $\gamma$, namely $(|V| - 1)m + O(|V|)$. The reason is that the external degree of
the nodes can still be as large as $|V|$. In order to reduce the external degree, we present in Sec. 4.2
a new partition algorithm, referred to as the $z$-partition algorithm. This algorithm is given two
parameters, $k$ and $z$, and ensures that the external degree of the nodes does not exceed $\lceil \frac{|V|}{z} \rceil - 1$.
Therefore, the memory overhead of $\zeta$ when using the $z$-partition algorithm is $(\lceil \frac{|V|}{z} \rceil - 1)m + O(|V|)$.
When using this partition algorithm, the time complexity of $\zeta$ is $O(z + \log_k |V|)$ per pulse, and the
communication complexity is $O(k|V|)$ per pulse. This gives a tradeoff between the time complexity
and the memory overhead of $\zeta$. However, selecting $z \leq \log_k |V|$, we obtain a reduction of the
memory overhead by $z$, while the order of time complexity is not increased.

In [ShS 1], we have presented synchronizer $\epsilon$, which has a memory overhead of $O(d)$ — much
better than the memory overhead of $\zeta$. However, the time complexity of $\epsilon$ is $O(D)$ per pulse, which
is very poor. In fact, we prove in [ShS 3] that this is the best time complexity that can be achieved
when the memory overhead is less than $S$. 
One more type of synchronizers presented in [ShS 1] is variable-duplicating synchronizers. Variable-duplicating synchronizers use a constant memory overhead of $O(d) + S$ while message-delaying synchronizers use a memory overhead that varies from $O(d)$ to $(d - 1)S + O(d)$, depending on the size $m$ of the original-protocol messages. Thus, variable-duplicating synchronizers are optimal for the worst case, but when the original-protocol messages are short, message-delaying synchronizers use less memory overhead. In cases where $(d - 1)m < S$, we prefer message-delaying synchronizers. Another reason for preferring message-delaying synchronizers in some cases is that they are twice as fast and have half the communication complexity of variable-duplicating synchronizers.

In the rest of this paper we proceed as follows: In Sec. 2 we review synchronizers $\epsilon$ and $\gamma$ and the partition algorithm of $\gamma$. Sec. 3 presents synchronizer $\zeta$. Sec. 4 discusses partition algorithms for $\zeta$. In Sec. 4.1 we prove that the partition algorithm of $\gamma$ does not reduce the memory overhead of $\zeta$ to less than the memory overhead of $\gamma$. The $z$-partition algorithm is discussed in sections 4.2, 4.3 and 4.4. In Sec. 5 we prove that the $z$-partition algorithm is optimal.

2 Preliminaries

2.1 Safety

In a combined protocol, a node $i$ is said to be safe with respect to pulse($n$), if all original-protocol messages sent by node $i$ at time $t_i(n)$ have already been received by the respective neighbors.

There are two ways for a node to know it is safe. The first is by using the confirm property of the data-link layer: the data-link layer informs the application when each transmitted message is confirmed to have been received by the respective neighbor. The second way is to require each node that receives an original-protocol message, to send back an explicit acknowledgement message.

Most synchronizers are based on the observation that a node $i$ can perform pulse($n$) when all its neighbors are safe with respect to pulse($n - 1$), since this means that node $i$ has received all the messages sent to it at pulse($n - 1$). The different synchronizers differ in the way this information is delivered to each node from its neighbors.

2.2 Synchronizer $\epsilon$

Since synchronizer $\epsilon$ [ShS 1] is used as a building block when constructing synchronizer $\zeta$ presented in this paper, we give here a brief description of $\epsilon$. In synchronizer $\epsilon$, an initialization phase creates a directed spanning tree for the network graph. After performing pulse($n - 1$), SAFE messages are converged along the spanning tree from the leaves to the root. Each node sends a SAFE message to its parent as soon as it is safe and has received SAFE messages from all its children. By the end of this process, the root node knows that all the nodes in the network are safe with respect to pulse($n - 1$) and ready to perform pulse($n$).
At this point, the root node sends AWAKE messages to all its children in the spanning tree and performs pulse\((n)\). Other nodes send AWAKE messages to all their children in the spanning tree and perform pulse\((n)\) upon receiving the first of the two following messages: an AWAKE message or an original protocol message which was sent at pulse\((n)\). In case the pulse was activated by the receipt of an original-protocol message, this message is processed immediately after the pulse is performed.

Observe that no early messages need to be saved since all messages are processed by the code activated by their arrival and before any other messages are processed (See Sec. 2.3). However, the code is performed in the right order — when an early message activates the pulse, the pulse is performed before the message is processed.

The memory overhead of \(\epsilon\) is \(O(d)\). The time and communication complexities of \(\epsilon\) are \(O(D)\) and \(O(|V|)\) per pulse respectively.

Three versions of \(\epsilon\) are discussed in [ShS 1]. These versions differ in the way the synchronizer distinguishes between original-protocol messages which were sent at different pulses. Synchronizer \(\epsilon_1\) does it by using an additional synchronization phase between each two pulses. Synchronizer \(\epsilon_2\) adds a suffix to each original-protocol message. The suffix contains the number of phase at which the message was sent (one bit is sufficient). In synchronizer \(\epsilon_3\) the pulse number is sent in a separated message before the original protocol message.

## 2.3 The memory model

Synchronizer \(\epsilon\) is based on the assumption that a message that is processed in the code activated by its arrival should not be saved in a special buffer, whereas a message that is processed on a later time, should. This follows from the model definition (Sec. 1, first paragraph), which says that a node can process an earlier received message, after processing later received messages, only if the earlier message was saved in the local variables when received.

In order to motivate this restriction, consider the following node configuration model: received messages are handed over to the application (which is, in our case, a combination of a synchronizer with a synchronous protocol) from a lower layer. The interface between the application and the lower layer is implemented by using one buffer. Whenever a message is received from one of the links, the lower layer copies it to the buffer and signals that there is a message to be processed. The application should process the message and then signal that the buffer can be used again. Only at this point, the lower layer may copy the next received message to the buffer. This model reflects some existing node configurations (e.g., smart communication boards). In this model, the only way to process an earlier received message after processing later received messages is if the application copies the message into a local buffer when received.

Another common node configuration is one in which there is a separated lower layer entity for each link. The interface between the application and each lower layer entity is performed in the same way as stated above. In this case, an earlier received message can be processed after later
received messages, if these messages were received from other links. However, if the interface to the lower layer is shared by some applications, each application should process the received messages immediately and release the interface buffers so that messages aimed to other applications are not delayed. Therefore, synchronizer \( \gamma \) is advantageous with this node configuration model as well.

We believe that designing more complicated interface schemes, that allow the application to process received messages in a non-fifo manner, leaving messages at the interface buffers or at the lower layer buffers for long periods of time, requires allocation of more buffers at the interface or at the lower layer. Therefore, such schemes do not help in reducing the memory overhead. Non-fifo interface schemes may also be impossible to implement or undesirable in some situations.

### 2.4 Synchronizer \( \gamma \)

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.

After each pulse, SAFE messages are converged along each cluster tree from the leaves to the leader node. At this point, CLUSTER\_SAFE messages are broadcast on the edges and preferred links of each tree, telling all neighboring clusters that this cluster is safe. The information that neighboring clusters are safe is brought to the cluster leader node by means of CLUSTER\_READY messages which converge along each tree, and then the leader node initiates a broadcast of AWAKE messages along the cluster tree, which cause the nodes in the cluster to perform the next pulse.

In synchronizer \( \gamma \), early messages may arrive to each node from each link which is not part of its cluster spanning tree. Thus, each node should reserve \( d - 1 \) buffers for saving early messages, and the memory overhead is \( (d - 1)m + O(d) \). The communication and time complexities of \( \gamma \) are \( O(k|V|) \) and \( O(\log_k(|V|)) \) per pulse respectively (\( k \) is a parameter given at initialization time, see Sec. 2.5).

The message delaying version of \( \gamma \) should be able to distinguish between early and timely original-protocol messages. In [ShS 1] and [ShS 2] we suggest three versions of synchronizer \( \gamma \) : \( \gamma_1 \), \( \gamma_2 \) and \( \gamma_3 \), which differ in the mechanism they use for distinguishing between early and timely messages.

### 2.5 The partition algorithm of \( \gamma \)

The initialization phase of \( \gamma \) is called a partition algorithm. This algorithm creates a partition of the network into clusters, builds a spanning tree for each cluster and selects a preferred link between each two neighboring clusters.

In this partition algorithm, clusters are built one by one. Each time, a new cluster is built from nodes in the remaining graph (nodes which are not in one of the previously built clusters). This is
done by selecting a leader node and then creating the cluster from nodes in the remaining graph that are around this leader node.

The job of creating the cluster around a given leader node is performed by a procedure named \texttt{ClusterCreation}. The operation of this procedure is controlled by a constant \( k \) \((2 \leq k \leq |V|)\) given to the partition algorithm when initiated. \texttt{ClusterCreation} operates in the following way: the selected leader node triggers an execution of a BFS protocol in the remaining graph. The nodes at each new BFS layer join the cluster as long as the number of nodes in the new layer is at least \((k - 1)\) times the total number of nodes in all previous layers.

**Definition 2.1 \((H_p)\)**

*Given a partition \( P \) of a network into clusters with spanning trees, \( H_p \) is defined as the maximum height of a cluster spanning tree.*

The time complexity of \( \gamma \) when using a partition \( P \) is \(4H_p\). The \texttt{ClusterCreation} procedure ensures that \( H_p \leq \log_k |V| \), therefore the time complexity of \( \gamma \) is \(4 \log_k |V|\).

**Definition 2.2 \((E_p)\)**

*Given a partition \( P \) of a network into clusters with spanning trees and with a preferred link between each two neighboring clusters, \( E_p \) is defined as the number of preferred links plus the total number of edges in all cluster spanning trees.*

The communication complexity of \( \gamma \) when using a partition \( P \) is \(4E_p\). The \texttt{ClusterCreation} procedure ensures that \( E_p \leq k|V| \), therefore the communication complexity of \( \gamma \) is \(4k|V|\).

The partition algorithm is initiated by one node. If a leader is not present in the network a-priori, a leader election procedure [GHP 1], [Awer 3] must be used.

Let \( v_{\text{init}} \) denote the node which starts the algorithm. Node \( v_{\text{init}} \) starts the algorithm by calling the \texttt{ClusterCreation} procedure. This procedure ends at node \( v_{\text{init}} \), after creating a cluster led by it. At this point, node \( v_{\text{init}} \) calls the \texttt{SearchForLeader} procedure, which searches for a leader of a new cluster. This procedure ends at a free node, if any. This free node calls \texttt{ClusterCreation} which creates a cluster around it, then it calls \texttt{SearchForLeader}, and so on.

The \texttt{SearchForLeader} procedure operates as follows: the procedure is initiated by the leader of a cluster \( C \) which has just been formed. The rejected layer of \( C \) is examined. In case it is not empty, one of the nodes in this layer is selected to be the next cluster leader. If the rejected layer is empty, the center of activity backtracks to the cluster from which the leader of \( C \) was elected, and the above procedure is repeated there. In the sequel we denote the cluster from which the leader of \( C \) was elected as the parent of \( C \). Notice that the parent-child relation between the clusters creates a DFS tree.

At the time when the activity of the \texttt{SearchForLeader} procedure backtracks from a cluster \( C \) to its parent cluster, all nodes in the neighborhood of \( C \) have already been joined to clusters. At this point, the cluster leader initiates the \texttt{PreferredLinkElection} procedure.
The Preferred Link Election procedure operates as follows: first, the edges are given distinct weights. The weight of a link \((i, j)\) is defined as the pair \((\min\{i, j\}, \max\{i, j\})\), and these pairs are ordered lexicographically. The preferred link between two neighboring clusters is selected to be the minimum-weight link which connects nodes from the two clusters. The election of the links is performed locally at each cluster, by means of a broadcast and convergecast process.


3 Synchronizer $\zeta$

In this section we discuss a new synchronizer, named $\zeta$, which is a message-delaying version of synchronizer $\gamma$ that provides a tradeoff between the time and communication complexities and the memory overhead. Synchronizer $\zeta$ provides a significant reduction in the memory overhead for a small penalty in time and communication complexity.

Recall that the initialization phase of $\gamma$ creates a partition of the network into separate clusters.

**Definition 3.1 (internal edges)**

We define an internal edge as an edge which connects two nodes belonging to the same cluster.

**Definition 3.2 (external edges)**

We define an external edge as an edge which connects two nodes belonging to different clusters.

As shown in [LT 1] and in [ShS 1], when performing the message-delaying version of synchronizer $\gamma$, early original-protocol messages may arrive at each node from each link connected to the node. These early messages are saved when received and used only after the pulse is performed.

In synchronizer $\zeta$, we suggest altering the mechanism by which the leader node causes the nodes in the cluster to perform a new pulse. The mechanism is altered such that original-protocol messages sent over internal edges are always timely. Thus, nodes do not ever need to save messages arriving from internal edges, but still need to save early messages arriving from external edges.

In synchronizer $\gamma$, after each pulse, the nodes perform a protocol to inform each leader node when all the nodes in its cluster are ready to perform the next pulse. The leader node then causes the nodes in the cluster to perform a new pulse by broadcasting an AWAKE message over the cluster tree. In synchronizer $\zeta$, we suggest replacing this broadcast with the mechanism of synchronizer $\epsilon$ [ShS 1]. Since $\epsilon$ ensures that no original-protocol messages arrive early, we ensure that original-protocol messages sent by nodes to other nodes in the same cluster are always timely. This mechanism is described here briefly and a description of synchronizer $\epsilon$ is given in Sec. 2.2.

When the leader node knows that all the nodes in its cluster are ready to perform the new pulse, it sends AWAKE messages to its children in the cluster spanning tree and then performs the pulse. Nodes in the cluster perform the new pulse when the first of the following happens:

1. an AWAKE message is received from the parent of the node in the tree, or
2. an original-protocol message sent at the new pulse is received from an internal edge.

Early original-protocol messages which are received from external edges do not cause the node to perform the new pulse; instead such messages are saved and processed only after the pulse. A node which performs the new pulse sends an AWAKE message to each of its children in the cluster tree before performing the new pulse. If the new pulse was triggered by the receipt of an original-protocol message, the message is processed immediately after performing the pulse.

The protocol requires nodes to be able to distinguish between timely and early original-protocol messages. When receiving an original-protocol message from an internal edge, the node should
perform a new pulse only if the message is early. When receiving an original-protocol message from an external edge, the node processes this message immediately if it is timely, and saves the message in order to be processed later if it is early.

In [ShS 1] and [ShS 2] we have discussed three different techniques for identifying whether a received original-protocol messages is timely or early. The first, used in synchronizers $\epsilon_1$, $\gamma_1$, and $\gamma_2$, is to use two waves of the synchronizer protocol between each two consecutive pulses. The second, used in $\epsilon_2$ and $\gamma_2$, is to add a suffix which contains the pulse number to each original-protocol message (one bit is sufficient). The third, used in $\epsilon_3$ and $\gamma_3$, is to send the pulse number in a distinct message, just before sending the original-protocol message. Details about the exact implementation of each of the three techniques, and a discussion of the pros and cons of each technique, can be found in [ShS 1] or in [ShS 2]. All three techniques may be used when implementing $\zeta$.

In order to show that $\zeta$ works properly, the following two properties should be proved:

1. nodes do not perform a pulse before they are ready to do so (received all the messages of the former pulse).
2. messages sent at pulse($n$) are processed by the receiving node after performing pulse($n$) and before performing pulse($n+1$).

In order to show that the first property holds, recall that in $\zeta$, exactly as in $\gamma$, the leader node of each cluster sends AWAKE messages and performs pulse($n+1$) only after all the nodes in the cluster and in the neighboring clusters are safe (by definition a node is safe with respect to pulse($n$) when all the messages it has sent at pulse($n$) have been received by the respective neighbors). Thus, when the leader performs pulse($n+1$), all the nodes in the cluster have already received the messages sent to them at pulse($n$). The leader node is the first in the cluster to perform pulse($n+1$) and, therefore, the first property holds.

In order to show that the second property holds, recall that original-protocol messages sent to a node $i$ at pulse($n$) are received by $i$ between $t_i(n-1)$ and $t_i(n+1)$. Those received on external edges before $t_i(n)$ are delayed and processed only after $i$ performs $t_i(n)$. If the first message received by $i$ on an internal edge is received before $t_i(n)$, it causes $i$ to perform $t_i(n)$ and only after this is done, the message is processed. Thus, all messages sent to node $i$ at pulse($n$) are processed after $t_i(n)$ and before $t_i(n+1)$, and the second property holds.

To conclude this section, we describe the full protocol of $\zeta$:

After the nodes perform pulse($n$), 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 a CLUSTER_SAFE message. Each node forwards this message to all its children 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 parent after having received a CLUSTER_SAFE message on all adjacent preferred links. Each intermediate node sends a NEIGHBORS_SAFE message to its parent after having received a NEIGHBORS_SAFE
message from all its children 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.

The leader node of each cluster knows at this point that all the nodes in the cluster are ready for the next pulse (all the nodes in the neighboring clusters are safe), so it initiates the process described above, in which AWAKE messages are forwarded along the cluster spanning tree and nodes perform the pulse upon receiving an AWAKE message or the first original-protocol message sent at this pulse and received from an internal edge. After the nodes perform pulse\((n + 1)\), the process starts all over again.
4 Partition Algorithms for $\zeta$

Recall that the memory overhead of the $\zeta$ synchronizer at a certain node is proportional to the number of external edges connected to it:

**Definition 4.1** (external degree of a node)

The external degree of a node in a partition $P$ of a network $N$ is defined as the number of external edges connected to it, when the partition $P$ is used.

We seek a partition algorithm that reduces the external degree of the nodes as much as possible for each given network.

4.1 The original partition algorithm

In this section we show that the partition algorithm [Awer 1] used for synchronizer $\gamma$ does not help to reduce the memory overhead when used to initialize $\zeta$. We show that this claim is true by presenting an example of a network in which a node that has a degree $d = |V| - 2$ also has an external degree of $|V| - 2$.

Consider the network of Fig. 1. Assume that node $s$ initiates the partition algorithm. If $k = |V|$, each node becomes a separate cluster. If $k \leq |V| - 1$, the first created cluster contains the nodes $s$ and $v_1, v_2, \ldots, v_{|V|-2}$. In both cases, node $s'$ forms a single-node cluster, and therefore, the external degree of $s'$ is $|V| - 2$. Thus, the memory overhead of $\zeta$ is, in this case, identical to the memory overhead of $\gamma$ and very high ($m \times (|V| - 2) + O(|V|)$).

4.2 The $z$-partition algorithm

In this section we present a partition algorithm that, given a parameter $z$ ($2 \leq z \leq |V|$), reduces the maximum external degree of the nodes to $\left\lceil \frac{|V|}{z} \right\rceil - 1$. We assume that the nodes know in advance the exact value of $|V|$. If this is not the case, the leader-election procedure [GHP 1],[Awer 3], which...
initiates this algorithm, can be used to provide this information without increase of communication or time complexities. For this partition algorithm, we classify the nodes of the network into the following three groups:

1. **red nodes** — nodes that have degree which is larger than $\lfloor \frac{|V|}{2} \rfloor - 1$.
2. **gray nodes** — nodes which are not red, but have a red neighbor.
3. **white nodes** — are not red and do not have a red neighbor.

We also classify the edges in the network into three categories:

1. **red edges** — a red edge is an edge that connects a red node with some other node.
2. **white edges** — connect white nodes.
3. **blue edges** — none of the above.

Our goal in the $z$-partition algorithm is to ensure that all red nodes have an external degree of 0. Thus, no node has an external degree of more than $\lfloor \frac{|V|}{2} \rfloor - 1$ and our goal is fulfilled. This is done by ensuring that all neighbors of a red node are in the same cluster as the red node itself. The $z$-partition algorithm creates two kind of clusters:

1. clusters consisting of red and gray nodes only. We will call such clusters ‘$z$-clusters’.
2. clusters consisting of white nodes only, created by using the original ClusterCreation procedure of the original partition algorithm. We will call such clusters ‘$k$-clusters’.

The $z$-partition algorithm consists of an initialization stage and a partition algorithm that starts when the initialization ends. The initialization stage uses two types of messages: PASS_TYPE and PASS_MEMBERSHIP.

- A PASS_TYPE($flag$) message contains a flag that tells whether the node that sends the message is red or not.
- A PASS_MEMBERSHIP($flag$) message contains a flag that tells whether the node that sends the message should be a member of a $z$-cluster (red or gray nodes) or of a $k$-cluster (white nodes).

The initialization stage is initiated by one node $s$. Node $s$ starts an execution of a PIF (Propagation of Information with Feedback) [Seg 1] of PASS_TYPE messages. By the end of this PIF protocol, each node knows which of its neighbors is a red node (if any). Now, each node knows also whether it should be a member of a $z$-cluster or of a $k$-cluster: members of a $z$-cluster are red nodes and their neighbors (which are the gray nodes).

At this point, node $s$ initiates another PIF, this time of PASS_MEMBERSHIP messages. When this PIF protocol ends, each node knows which of its neighbors belongs to a $z$-cluster and which belongs to a $k$-cluster. When the second PIF protocol ends, the initialization stage is over and node $s$ initiates the partition algorithm.

The partition algorithm is a version of the original partition algorithm with an altered ClusterCreation procedure. The new ClusterCreation procedure is built of two sub-procedures:
The main procedure calls \( z \text{-Cluster Creation} \) if the cluster leader (which is the node that initiates the procedure) is a red or gray node, and calls \( k \text{-Cluster Creation} \) if the cluster leader is a white node.

The \( k \text{-Cluster Creation} \) procedure operates in a way which is very similar to the operation of the original \( \text{Cluster Creation} \) procedure. The only difference is that this procedure builds the cluster from white nodes only. In other words, the procedure uses only white edges and ignores other types of edges.

The \( z \text{-Cluster Creation} \) procedure builds the cluster spanning tree by executing a BFS protocol that uses only red edges. The BFS algorithm continues to append new layers to the cluster as long as these layers are not empty.

The \( \text{Search For Leader} \) and \( \text{Preferred Link Election} \) procedures used by the \( z \)-partition algorithm are those used by the original partition algorithm presented in [Awer 1].

### 4.2.1 The time and communication complexities of synchronizer \( \zeta \) when using the \( z \)-partition algorithm

In order to understand how large the diameter of a \( z \)-cluster can be, see Fig. 2. This figure demonstrates the case in which \( 2z - 1 \) red nodes form a path in which the first red node shares a neighbor (or more) with the second one, the second red node shares neighbors with the third one, etc. The nodes in this network are joined into one long \( z \)-cluster by the \( z \)-partition algorithm. This is due to the fact that each red node is in distance of 2 from the preceding red node and the next red node in the path, and all the other nodes are in distance of one from a red node. Since all the red nodes are ordered in a long path, this is the longest \( z \)-cluster possible. The diameter of such a cluster is at most \( 4z - 5 \).

Recall that the time complexity of \( \zeta \) per pulse is four times the height of the highest cluster spanning tree (called \( H_p \)). The \( k \text{-Cluster Creation} \) procedure creates clusters of height \( H_p \leq \log_k |V| \) (this is proved in [Awer 1]). The highest \( z \)-cluster is at most of height \( 4z - 5 \). Thus, the \( z \)-partition algorithm ensures that \( H_p \leq \max\{\log_k |V|, 4z - 5\} \). The time complexity of \( \zeta \) per pulse, when using the \( z \)-partition algorithm, is therefore, \( 4 \times \max\{\log_k |V|, 4z - 5\} \).

The communication complexity of synchronizer \( \zeta \) is four times the number of edges which participate in the synchronizer protocol (called \( E_p \)). This contains the edges of the spanning forest, which are at most \(|V| - 1\), and the chosen preferred links.

The \( k \text{-Cluster Creation} \) procedure ensures that the number of preferred links connecting neighboring \( k \)-clusters is at most \((k - 1)|V|\). The maximum number of \( z \)-clusters is \( z \). There are at most \(|V|\) clusters in the whole partition. Thus, the number of preferred links connecting \( z \)-clusters to \( k \)-clusters or \( z \)-clusters, is at most \( z |V| \). We conclude that the upper bound on the number of preferred links is \(((k - 1) + z)|V|\), and therefore, \( E_p \leq (k + z)|V| \). The communication complexity of \( \zeta \) per pulse, when using the \( z \)-partition algorithm, is therefore, \( 4(k + z)|V| \).

These results show that the \( z \)-partition algorithm provides a tradeoff between the memory
overhead of $\zeta$, and its time and communication complexity: the larger the parameter $z$ given to this partition algorithm, the smaller the memory overhead, and the larger the communication and time complexities. For example, selecting $z = \min\{k, \log |V|\}$, reduces the memory overhead of $\zeta$ by factor $z$, namely from $(|V| - 1)m + O(|V|)$ to $(\frac{|V|}{z} - 1)m + O(|V|)$, without increasing the time or communication complexities.

4.3 An improvement of the $z$-partition algorithm

In this section we present an improved $z$-partition algorithm which does not affect the communication complexity of $\zeta$ as long as $z \leq \sqrt{|V|}$. For the case where $z > \sqrt{|V|}$, the improved algorithm still provides better communication complexity than the $z$-partition algorithm.

The improvement is done by changing the $k$-Cluster Creation procedure. Here we describe these changes, in Sec. 4.3.1 we show how this improvement reduces $E_p$ to be $O(k|V| + z^2)$ and in Sec. 4.3.2 we show that this improvement does not change the order of $H_p$.

In the original $k$-Cluster Creation procedure, a new layer of nodes is examined at each pulse. If the new layer contains at least $k - 1$ times more nodes than are in all former layers, it joins the cluster; if not, it is rejected.

This $k$-Cluster Creation procedure is changed as follows: in the altered procedure, as long as the cluster contains less than $z$ nodes, the new layer is not rejected, no matter how few nodes it contains. In the case where the new layer contains no nodes at all, while there are less than $z$ nodes in the cluster, this cluster joins the cluster that elected its leader (its parent cluster in the clusters DFS tree). In the sequel, we call the action of joining the parent cluster ‘hooking’.

Observe that in the case where the problematic cluster is the first created by the partition algorithm, it cannot perform hooking. In this case, the cluster remains as it is.

4.3.1 The communication complexity of $\zeta$ when using the improved $z$-partition algorithm

The following discussion refers to a partition created by an execution of the improved $z$-partition algorithm, on an arbitrary network.

Lemma 4.1 The number of preferred links between $k$-clusters is at most $(k - 1)|V|$.

Proof: The proof of this lemma is similar to the one in [Awer 1], since it is not affected by the changes made to $k$-Cluster Creation. We count only preferred links from a $k$-cluster to $k$-clusters that were formed after it. In this way each link is counted exactly once.

The $k$-Cluster Creation procedure rejects a new layer only if the number of nodes in this layer is less than $k - 1$ times the number of nodes already in the cluster. Thus, when a $k$-cluster which contains $c$ nodes is formed, at most $(k - 1)c$ free white nodes are at distance of one hop from it.
Therefore, at most \((k - 1)e\) of the \(k\)-clusters formed afterwards may be its neighbors. The sum of preferred links between \(k\)-clusters is thus less than \((k - 1)\Sigma e \leq (k - 1)|V|\).

**Hooking** of clusters does not change this outcome, since it only reduces the number of \(k\)-clusters and thus the number of preferred links between them.

\[\text{Lemma 4.2} \quad \text{The number of } k\text{-clusters is at most } \left\lceil \frac{|V|}{z} \rightceil + 1.\]

**Proof:** With the altered \(k\_\text{Cluster\_Creation} \) procedure, every \(k\)-cluster except for the first one, hooks on to another cluster if it contains less than \(z\) nodes. Thus, each of the remaining \(k\)-clusters contains at least \(z\) nodes and therefore the number of \(k\)-clusters cannot exceed \(\left\lceil \frac{|V|}{z} \rightceil + 1\). Number 1 is added to take into consideration the fact that if the first cluster created is a \(k\)-cluster with less than \(z\) nodes, it does not hook to any other cluster.

The number of \(z\)-clusters is at most \(z\) since each one contains at least \(\left\lceil \frac{|V|}{z} \rightceil\) nodes. Thus, the number of preferred links between \(z\)-clusters is at most \(z^2\), and the number of preferred links connecting \(z\)-clusters to \(k\)-clusters is at most \(z \times \left(\left\lceil \frac{|V|}{z} \rightceil + 1\right) \leq |V| + z\).

An upper bound for the number of preferred links can be now easily computed:

1. \((k - 1)|V|\) links between \(k\)-clusters,
2. \(z^2\) links between \(z\)-clusters,
3. \(|V| + z\) links connect \(z\)-clusters to \(k\)-clusters.

This sums up to \(k|V| + z^2 + z\), and when adding at most \(|V| - 1\) edges of the spanning forest, we gain: \(E_p \leq (k + 1)|V| + z^2 + z - 1 < (k + 2)|V| + z^2\). The communication complexity of \(\zeta\) with this partition is \(4 \times E_p < 4(k + 2)|V| + 4z^2\) per pulse, which is obviously better than the complexity achieved by the \(z\)-partition algorithm.

In case we select \(z \leq \sqrt{|V|}\), the communication complexity becomes \(4(k + 2)|V| + 4|V| = 4(k + 3)|V|\), which is of the same order as the communication complexity with the original partition algorithm presented in [Awer 1].

### 4.3.2 The time complexity of \(\zeta\) when using the improved \(z\)-partition algorithm

Let us first analyze the maximum height of a cluster spanning tree in a partition created by the improved \(z\)-partition algorithm, assuming no **hooking** is done.

The \(z\_\text{Cluster\_Creation} \) procedure is not changed, thus the maximum height of a \(z\)-cluster is \(4z - 5\).

The highest possible \(k\)-cluster is created as follows: the first \(z - 2\) layers contain exactly one node each, creating a cluster of \(z - 1\) nodes, \(z - 2\) hops high. The later formed layers must obey the rule that each layer contains \(k - 1\) times more nodes than all former layers. Thus, no more than \(\log_k |V|\) such layers can be added. The height of this cluster is no more than \(z + \log_k |V|\).

Observe that a \(k\)-cluster \(C_1\) hooks on another cluster \(C_2\) only when there are no free white nodes neighboring \(C_1\). Thus, new \(k\)-clusters neighboring \(C_1\) are not created, and no cluster hooks
The height of $C_1$ is at most $z$, and no other cluster can hook it, therefore $C_1$ adds at most $z$ to the height of $C_2$. In fact, no matter how many $k$-cluster hook on $C_2$, they do not add more than $z$ hops to its height.

The maximum height of a cluster is therefore $H_p = \max\{4z-5, z+\log_k |V|\} + z < 5z + \log_k |V|$. The time complexity of $\zeta$ is $4 \times H_p < 20z + 4\log_k |V|$ per pulse. This is the same as the order of time complexity achieved by the $z$-partition algorithm.

### 4.3.3 The time and communication complexities of the improved $z$-partition algorithm

- The algorithm starts with an execution of a leader election procedure [GHP 1],[Awer 3], which also counts the number of nodes in the network. The time complexity of this procedure is $O(|V| \log |V|)$ and the communication complexity is $O(|V| \log |V| + |E|)$.

- Now comes the turn of the initialization stage, which consists of two PIF waves, one of PASS_TYPE messages and one of PASS_MEMBERSHIP messages. The time and communication complexities of the initialization stage are $O(|V|)$ and $O(|E|)$ respectively.

- The BFS algorithm used by the $k$-Cluster_Creation procedure consists of pulses, where in each pulse, the cluster leader broadcasts PULSE messages towards the leaves. Each leaf node sends LAYER messages over all adjacent links. These messages are acknowledged by ACK messages, and then COUNT messages converge back to the leader node. Each $k$-cluster creation consists of at most $O(z + \log_k |V|)$ pulses, thus at most $O((z + \log_k |V|)|V|)$ PULSE and COUNT messages are sent during the whole algorithm. One LAYER and one ACK message at most is sent over each link, yielding at most $O(|E|)$ messages.

The hooking process takes exactly two messages, and thus does not change the order of complexity of the procedure invocation.

Thus, the communication complexity of the $k$-Cluster_Creation procedure is $O((z + \log_k |V|)|V| + |E|)$. Consider a $k$-cluster with $c$ nodes, the height of this cluster spanning tree is at most $h \leq z + \log_k c$. Hence, the total time spent in forming this cluster is $O((z + \log_k c)c) = O(ze + c\log_k c)$. Summing up for all the $k$-clusters gives $O(z|V| + |V| \log_k |V|)$ — the overall time complexity of the invocations of $k$-Cluster_Creation.

- If we sum up the heights of all $z$-cluster trees (without counting the extra height gained by hooking), we gain a maximum sum of $4z - 5$. A $z$-cluster is built by using a BFS algorithm similar to the one used by $k$-Cluster_Creation. Thus, in the worst case, a total number of $4z - 5$ pulses are performed by all the invocations of $z$-Cluster_Creation. Each pulse takes at most $4z - 5$ time units. Thus, the sum of the time spent by all invocations of $z$-Cluster_Creation is $O(z^2)$.

As for the $k$-Cluster_Creation procedure, the $z$-Cluster_Creation procedure sends at most one LAYER and ACK messages over each link. At each pulse, at most $O(|V|)$ PULSE and COUNT mes-
messages are sent. Thus, the total communication complexity of all invocations of \( \text{Cluster\_Creation} \) is \( O(|V| + |E|) \).

- The \text{Search\_For\_Leader} procedure traverses the already created clusters in a DFS order, trying to find a free node which would be the leader node of a new cluster. Each time a cluster is traversed in search for a new candidate is referred to as a \( \text{move} \) (see [Awer 1]).

A \( \text{move} \) involves a broadcast of \text{TEST} messages and a convergecast of \text{CANDIDATE} messages. It also involves a string of \text{LEADER} messages from the cluster leader to the elected new node, or a string of \text{RETREAT} messages to the leader of the cluster that has elected the current leader (its parent in the clusters DFS tree). Thus, for both kind of clusters, \( C_{\text{move}} = O(|V|) \) and \( T_{\text{move}} = O(z + \log_k |V|) \).

Each cluster contributes two \( \text{moves} \) to the execution of the algorithm, one is when the leader of the cluster is elected, and one is when the cluster is traversed and no new leader is found. There are at most \( O(|V|) \) clusters in the network and therefore the total communication complexity of all the invocations of \text{Search\_For\_Leader} is \( O(|V|^2) \), and the total time complexity is \( O((z + \log |V|)|V|) \).

- \text{Preferred\_Link\_Election} involves one broadcast and one convergecast of messages over each cluster, per each preferred link connected to the cluster. Since the number of preferred links is \( O(k|V| + z^2) \), the total communication complexity of this procedure is \( O(k|V|^2 + z^2 \times |V|) \).

The procedure is invoked concurrently on different clusters, therefore only the last invocation affects the time complexity of the whole algorithm. Since at most \( |V| \) preferred links are connected to each cluster, the time complexity of this last invocation is \( O((z + \log |V|)|V|) \).

The total communication complexity of the improved \( \zeta \)-partition algorithm is:

\[
O(|V| \log |V| + |E|) + O(|E|) + O((z + \log |V|)|V| + |E|) + \\
O(z|V| + |E|) + O(|V|^2) + O(k|V|^2 + z^2 \times |V|) = \\
O(k|V|^2 + z^2 \times |V|)
\]

The total time complexity of the improved \( \zeta \)-partition algorithm is:

\[
O(|V| \log |V| + O(|V|) + O((z + \log |V|)|V|) + O(z^2) + \\
O((z + \log |V|)|V|) + O((z + \log |V|)|V|) = \\
O((z + \log |V|)|V|)
\]

### 4.4 A further improvement of the \( \zeta \)-partition algorithm

The improved \( \zeta \)-partition algorithm presented in Sec. 4.3 is satisfactory for most cases. For example, selecting \( z = \min \{ \log_k |V|, \sqrt{|V|} \} \) reduces the memory overhead of \( \zeta \) by factor \( z \), while retaining the time and communication complexities. Furthermore, for virtually all \( k \) and \( |V| \) such that
$k < |V|$ holds, $\log_k |V| \leq \sqrt{|V|}$. Thus, selecting $z$ such that the time complexity of $\zeta$ is not damaged ($z \leq \log_k |V|$) ensures also that the communication complexity is not damaged ($z \leq \sqrt{|V|}$).

In this section we present a further improvement of the $z$-partition algorithm which reduces the communication complexity of $\zeta$ to $O(k|V|)$ — identical to the communication complexity of $\gamma$ [Awer 1] and to the lower bound proved in [Awer 1]. This improvement makes the $z$-partition algorithm more complicated. Hence, we suggest that it would not be used in cases where $z \leq \sqrt{|V|}$, where the partition algorithm of Sec. 4.3 is satisfactory.

The main idea of this improvement is the following: after the improved $z$-partition protocol execution ends, an additional algorithm is executed. In this algorithm, every two neighboring $z$-clusters which contain less than $z$ nodes are combined into one cluster. In the sequel we refer to $z$-clusters with less than $z$ nodes as small $z$-clusters, and other $z$-clusters as large $z$-clusters. The additional algorithm repeats the process of combining neighboring small $z$-clusters as long as possible.

4.4.1 The communication complexity of $\zeta$ when using the further improved $z$-partition algorithm

The number of edges in the spanning forest is less than $|V|$. The number of preferred links between $k$-clusters remains $(k-1)|V|$ and the number of preferred links which connect $z$-clusters to $k$-clusters remains $|V| + z$.

The further improved partition algorithm ensures that there are no preferred links which connect small $z$-clusters. Therefore, when counting the number of preferred links between $z$-clusters, it is sufficient to count only the links adjacent to large $z$-clusters.

There are at most $\left\lceil \frac{|V|}{z} \right\rceil$ large $z$-clusters, each of which may have at most $z$ preferred links which connect it to other $z$-clusters. Hence, the number of preferred links connecting $z$-clusters is at most $\left\lceil \frac{|V|}{z} \right\rceil \times z \leq |V|$.

$$E_p \leq |V| + (k - 1)|V| + |V| + z + |V| = (k + 2)|V| + z < (k + 3)|V|$$

Thus, when using the further improved partition algorithm, the communication complexity of $\zeta$ is $O(k|V|)$ per pulse. This is optimal and identical to the communication complexity achieved when using the original partition algorithm of [Awer 1].

4.4.2 The time complexity of $\zeta$ when using the further improved $z$-partition algorithm

Each new cluster created after the execution of the improved $z$-partition protocol, is composed of two clusters. Each of these clusters contains less than $z$ nodes. Hence, the height of the composed cluster is at most $2z - 3$, and the value of $H_p$ remains $O(z + \log_k |V|)$. The time complexity of $\zeta$ remains, therefore, $O(z + \log_k |V|)$ per pulse.
4.4.3 Implementation details

A new procedure, named Combine_Clusters, is introduced. This procedure is activated just before the Preferred_Link_Election procedure, when the activity of the Search_For_Leader procedure backtracks from the cluster. The Preferred_Link_Election procedure is activated at each cluster upon completion of the Combine_Clusters procedure. Observe that when the Combine_Clusters procedure is activated at a cluster, all nodes in the neighborhood of this cluster have already been joined to clusters.

- The Combine_Clusters procedure in \( k \)-clusters is started by the leader of the cluster. The leader node makes the nodes in the cluster send FORMED messages on all incident links. The FORMED messages contain the identity of the cluster leader. They inform all the neighboring clusters that this cluster is not going to change. After sending the FORMED messages, the nodes of the cluster wait for a FORMED message to be received from each link. A convergecast protocol over the cluster spanning tree is used to inform the leader node when all the nodes have received FORMED messages from all links. This ends the Combine_Clusters procedure and starts the Preferred_Link_Election procedure.

- The Combine_Clusters procedure starts in \( z \)-clusters with a process of counting the nodes in the cluster. This is performed by a broadcast and convergecast of messages along the cluster spanning tree. Observe that the size of the cluster computed by the Cluster_Creation procedure may not be valid, since some \( k \)-clusters may have hooked it.

A cluster which contains \( z \) nodes or more (a large cluster), performs the process of sending FORMED messages and waiting for FORMED messages from all neighboring clusters, just as in \( k \)-clusters.

In the sequel, distinct weights are assigned to all links. The weights are identical to those used in the Preferred_Link_Election procedure: the weight of a link \((i, j)\) is defined as the pair \((\min\{i, j\}, \max\{i, j\})\), and these pairs are ordered lexicographically.

- In a cluster which contains less than \( z \) nodes (a small cluster), the leader node initiates a process in which all the nodes in the cluster send COMBINE_WISH messages over all incident links. A COMBINE_WISH message contains the name of the cluster leader and the number of nodes in the cluster. After performing this, the cluster leader waits until COMBINE_WISH or FORMED messages are received over all the links connected to the nodes of the cluster.

In case a FORMED message has been received from all neighboring clusters, the cluster has no small neighboring \( z \)-clusters. Thus, it cannot be combined with any other cluster and has reached its final size. In this case, the cluster leader initiates a broadcast of FORMED messages: each node in the cluster sends a FORMED message over each of its incident links. Since FORMED messages have already been received from all neighboring clusters, the Combine_Clusters procedure ends and the Preferred_Link_Election procedure is initiated.
In case a COMBINE_WISH message is received from some neighboring clusters, the edges over which the COMBINE_WISH messages from neighboring clusters have been received are edges leading to clusters with which the current one may be combined. In the sequel, we call such edges potential edges.

The leader node starts a process in which the potential edge with the smallest weight is selected. Then, a COMBINE_REQUEST message is sent to the node on the other side of this edge. Only clusters that have selected each other are combined. The combining process is performed over the edge on which COMBINE_REQUEST messages have been sent in both directions. There always exists such an edge; at least the potential edge with the smallest weight is such.

The combining of the two clusters is performed in the same way as done in [GHP 1]: the cluster spanning trees are combined along the edge over which the two COMBINE_REQUEST messages were sent. The new cluster leader is one of the nodes adjacent to this edge.

After the clusters are combined, the new cluster leader calculates the size of the created cluster. In case the created cluster is still a small $z$-cluster and still has small $z$-cluster neighbors, a new edge is selected and the process repeats all over again.

If the created cluster is a large $z$-cluster, it broadcasts FORMED messages, and waits until FORMED messages are received from all neighboring clusters.

When a FORMED message is received from a potential edge by a node of a small $z$-cluster, the edge from which the message has been received stops being a potential edge. Denote the cluster at the other end of this edge by $C$. The cluster leader is informed that $C$ is not interesting anymore. At this point, the cluster leader informs all nodes in the cluster that edges leading to $C$ have stopped being potential edges. If the selected edge is one of these, a new edge is selected.

### 4.4.4 The time and communication complexities of the further improved $z$-partition algorithm

Exactly $2|E|$ FORMED messages are sent by the Combine_Clusters procedure, in a process which takes at most $O(z + \log |V|)$ time. At most $2|E|$ COMBINE_WISH messages are sent, in at most $O(z)$ time. The process of realizing that FORMED and COMBINE_WISH messages have been received over all incident links takes $O(|V|)$ messages and $O(z + \log |V|)$ time.

At most $z$ combinations of clusters take place. Each involving $O(|V|)$ communication and $O(z)$ time, yielding overall $O(z|V|)$ communication and $O(z^2)$ time complexity.

On the other hand, since the number of preferred links is now reduced to $O(k|V|)$, the communication complexity of Preferred_Link_Election is reduced to $O(k|V|^2)$. Hence, the overall communication complexity is reduced to $O(k|V|^2)$, the same as the communication complexity of the original partition algorithm of [Awer 1]. The time complexity remains $O(|V|\log |V| + z|V|)$.
Figure 2: A cluster with diameter $4z - 5$
5 Lower bound for the tradeoff between memory overhead and time complexity of synchronizer $\zeta$

In order to bound the memory overhead of synchronizer $\zeta$ to $(\lfloor \frac{|W|}{z} \rfloor - 1)m + O(|V|)$, the partition algorithm must create a partition in which there are no nodes with external degree of more than $\lfloor \frac{|W|}{z} \rfloor - 1$.

In this section we prove that there exists a family of networks for which any partition that achieves the above goal, creates clusters of height $H_p = \Omega(z)$.

An example for such a family of networks is shown in Fig. 3. A network in this family is built of a path of $\lfloor \frac{s}{2} \rfloor$ red nodes. Each two successive red nodes are not neighbors, but share at least $\lfloor \frac{|W|}{z} \rfloor$ neighbors.

**Lemma 5.1** Let $P$ be a partition of a network from the family presented in Fig. 3. Assume that in $P$ no node has an external degree of more than $\lfloor \frac{|W|}{z} \rfloor - 1$. Then, each two successive red nodes must be in the same cluster.

**Proof:** Assume that $i$ and $j$ are two successive red nodes. Assume also that in the partition $P$ these nodes are not in the same cluster. Let $G_{ij}$ be the set of common neighbors of nodes $i$ and $j$. $G_{ij}$ contains at least $\lfloor \frac{|W|}{z} \rfloor$ nodes.

At least $\lfloor \frac{|W|}{z} \rfloor + 1$ nodes from $G_{ij}$ must be in the cluster of node $i$, or else, the external degree of $i$ would be more than $\lfloor \frac{|W|}{z} \rfloor - 1$. All these $\lfloor \frac{|W|}{z} \rfloor + 1$ nodes are not in the cluster of $j$. Therefore, the external degree of node $j$ is at least $\lfloor \frac{|W|}{z} \rfloor + 1$ — contradiction.

**Theorem 5.2** A partition algorithm that ensures to $\zeta$ a memory overhead of no more than $(\lfloor \frac{|W|}{z} \rfloor - 1)m + O(|V|)$, requires time complexity of $\Omega(z)$ per pulse of $\zeta$.

**Proof:** Let $N$ be a network from the family presented in Fig. 3, containing $\lfloor \frac{s}{2} \rfloor$ red nodes. There are at least $\lfloor \frac{|W|}{z} \rfloor$ gray nodes between every two successive red nodes in $N$. Let $P$ be a partition that ensures to $\zeta$ a memory overhead of $(\lfloor \frac{|W|}{z} \rfloor - 1)m + O(|V|)$. Thus, when using $P$, the nodes of $N$ have an external degree of no more than $\lfloor \frac{|W|}{z} \rfloor - 1$.

According to Lemma 5.1, every two successive red nodes must be in the same cluster. Thus, by using a simple inductive argument, we conclude that all red nodes must be in the same cluster. The diameter of this cluster is $z - 1$, therefore any spanning tree of this cluster has a height of at least $\lfloor \frac{z-1}{2} \rfloor$ hops.

The time complexity of $\zeta$ when used with the partition $P$, is four times the height of the highest cluster spanning tree. Therefore, in our case the time complexity of $\zeta$ is $\Omega(z)$.

\[\square\]
Figure 3: A one cluster network with diameter $z$
References


