


References


Since the network is a hypercube ring, or cubic torus, there exists a node $l$ which is at distance $n$ from at least one node in $W$, $n + 1$ from $k$ and $n + 2$ from $j$ (for example, in hypercubes, $l$ can be found by selecting a node $w$ from $W$ and moving away from it in $n$ dimensions which are not the dimension which separates $w$ from $k$ or the dimension which separates $j$ from $k$). Thus, $V^E$ contains at least one item that is not 0, while the same item is 0 in $V^j$ and $V^k$. There also exists an item which is not 0 in $V^j$ and is 0 in $V^k$. Thus, $V^k$ and $V^j$ are linearly independent, and $V^E$ cannot be represented as a linear combination of $V^k$ and $V^j$.

\begin{lemma}
In the network of Lemma A.3, let $W$ be a subset of $G_k$, $E = \sum_{i \in W} A_i(n)$ and $j \notin W$ a neighbor of $k$. Then, node $k$ cannot store $A_k(n)$, $A_j(n - 1)$ and $E$ using $M_{syn} + 2S$ bits of memory.
\end{lemma}

**Proof:** Let $V^k$, $V^j$ and $V^E$ be the vectors which represent $A_k(n)$, $A_j(n)$ and $E$ respectively. There exists an item which is not 0 in $V^k$ and is 0 in $V^j$. There also exists an item which is not 0 in $V^E$ and is 0 in $V^k$ and $V^j$, thus the vectors are linearly independent. \hfill \Box

\begin{lemma}
In the network of Lemma A.3, let $W$ be a subset of $G_k$ and $E = \sum_{i \in W} A_i(n)$. Then, the value of $E$ cannot be extracted from $A_k(n)$ and \{ $A_i(n) | i \in G_k - W$ \}.
\end{lemma}

**Proof:** Let $j \in W$ be a neighbor of $k$ which is in $W$. There exists a node $l$ which is at distance $n$ from $j$, but at distance of more than $n$ from $k$ and all the nodes in $G_k - j$. The value of each bit in $E$ depends on the initial value of $A_l(0)$. Yet, the values $A_k(n)$ and \{ $A_i(n) | i \in G_k - W$ \} are not functions of $A_l(0)$, and thus, no bit of $E$ can be extracted from $A_k(n)$ and \{ $A_i(n) | i \in G_k - W$ \}. \hfill \Box
Lemma A.2 Let \( A = (A_1, A_2, \ldots, A_N) \) be a \( N \)-dimensional vector, in which each element contains \( S \) randomly chosen bits. Let \( V^1, V^2, \ldots, V^K \) be \( K \) linearly independent vectors, each of \( N \) items. Let \( X^1, X^2, \ldots, X^K \) be the inner multiplications \( V^1 \cdot A, V^2 \cdot A, \ldots, V^K \cdot A \) respectively. Then, the amount of memory needed to store \( X^1, X^2, \ldots, X^K \) and \( V^1, V^2, \ldots, V^K \) is at least \( KS \) bits.

Proof: Assume, in contradiction, that \( X^1, X^2, \ldots, X^K \) and \( V^1, V^2, \ldots, V^K \) can be stored using less than \( KS \) bits. This can be used in order to store \( A_1, A_2, \ldots, A_N \) using less than \( NS \) bits: \( A_1, A_2, \ldots, A_{N-K} \) are stored as they are, in \((N-K)S\) bits. Instead of storing \( A_{N-K+1}, \ldots, A_N \) we store \( X^1, X^2, \ldots, X^K \) and \( V^1, V^2, \ldots, V^K \), using less than \( KS \) bits. \( A_1, A_2, \ldots, A_{N-K} \) can be obtained from the stored data by solving the \( K \) created linear equations. Thus, \( A_1, A_2, \ldots, A_N \) are stored in less than \( NS \) bits — contradiction.

The following three Lemmas discuss a hypercube, ring, or cubic torus network in which the nodes perform the combined protocol created by combining a fixed-overhead synchronizer \( Syn \) with the synchronous protocol \( P \), created for \( Syn \) in the proof of Lemma 1. Recall that \( M_{Syn} \) denotes the maximum over the nodes of the amount of memory needed for the synchronizer variables, \( M_T \) denotes the amount of memory needed to store an image of the entire network topology and the current pulse number, and \( S \) denotes the number of bits used by \( P \) at each node. Recall also that \( S > 2M_{Syn} + M_T \). \( G_k \) denotes the set of neighbors of node \( k \).

Lemma A.3 In the network described above, let \( W \) be a subset of \( G_k \) and \( E = \Sigma_{i \in W} A_i(n) \). Then, node \( k \) cannot store \( A_k(n) \) and \( E \) using \( 2M_{Syn} + S \) bits of local memory.

Proof: Assume, in contradiction, that \( A_k(n) \) and \( E \) can be stored in the local memory of node \( k \) using \( 2M_{Syn} + S \) bits. Now, let us add \( M_T \) more bits to this local memory, containing an image of the entire network topology and the pulse number \( n \).

Recall that at the initialization, the local variables \( A_i \) at the nodes are given a random value \( A_i(0) \). The values \( A_k(n) \) and \( E \) are both linear combinations of these original values. Let \( V^k \) and \( V^E \) be the vectors which represent \( A_k(n) \) and \( E \) respectively as linear combinations of \( (A_i(0)) \). \( V^k \) and \( V^E \) can be extracted from the network topology and the pulse number.

Since the network is a hypercube, ring, or cubic torus, for each node \( j \in W \), there exists a node \( i \) which is at distance \( n \) from \( j \) and at distance \( n+1 \) from \( k \). Therefore, the \( l \)-th item in \( V^E \) is not \( 0 \), while the \( l \)-th item in \( V^k \) is \( 0 \). Thus, \( V^E \) and \( V^k \) are linearly independent.

The \( 2M_{Syn} + M_T + S \) (less than \( 2S \)) bits of memory in node \( k \) contain \( A_k(n), E \), and sufficient information for extracting the two linearly independent vectors which represent \( A_k(n) \) and \( E \). This contradicts Lemma A.2.

Lemma A.4 In the network of Lemma A.3, let \( W \) be a subset of \( G_k \), \( E = \Sigma_{i \notin W} A_i(n) \) and \( j \notin W \) a neighbor of \( k \). Then, node \( k \) cannot store \( A_k(n) \), \( A_j(n) \) and \( E \) using \( M_{Syn} + 2S \) bits of memory.

Proof: Let \( V^k \), \( V^j \) and \( V^E \) be the vectors which represent \( A_k(n), A_j(n) \) and \( E \) respectively. As in Lemma A.3, all we need to prove is that \( V^E, V^k \) and \( V^j \) are linearly independent.
We conclude that at time \( t_i(n) \), the local memory of \( i \) contains the information about \( A_i(n) \) and about the sum of the values of all early messages. Each of these values is \( S \) bits long. The local memory consists of \( S + M_{syn} \) bits, plus the buffer which holds the currently processed message. Thus, three cases should be discussed:

1. \( t_i(n) \) is triggered by a synchronizer message: in this case, the \( S + 2M_{syn} \) bits of local memory contain the \( 2S \) bits of information. This is not sufficient, as proved in Lemma A.3, leading to a contradiction.

2. \( t_i(n) \) is triggered by an original-protocol message sent at pulse\((n)\): in this case, the \( S + M_{syn} \) bits of local memory, plus the \( S \) bits of the buffer, contain the \( 3S \) bits of information. This is not sufficient, as proved in Lemma A.4, leading to a contradiction.

3. \( t_i(n) \) is triggered by an original-protocol message sent at pulse\((n - 1)\): in this case, as in the former one, the \( S + M_{syn} \) bits of local memory, plus the \( S \) bits of the buffer, contain the \( 3S \) bits of information. Again, this is not sufficient, as proved in Lemma A.5, leading to a contradiction.

Lemma 1 is proved by showing that a fixed-overhead synchronizer that does not satisfy the predicate of the Lemma for \( P \) and \( N \), would cause wrong synchronous-protocol messages to be sent. Yet, this does not mean that the stronger property of having a correct final outcome is also violated. We claim that in this case, wrong synchronous-protocol messages lead to a wrong final outcome. This follows from the fact that in \( P \), the local state of each node at pulse\((n + 1)\) and the messages sent by each node at pulse\((n + 1)\), are functions of the messages sent by the nodes at pulse\((n)\). Thus, by using simple induction, one can show that for each \( n \), the final outcome of the protocol can be calculated as a function of the messages sent by the nodes at pulse\((n)\). Hence, wrong messages sent at pulse\((n)\) lead to a wrong final outcome.

Lemmas A.3, A.4 and A.5, presented in the sequel, are used for proving Lemma 1. The following two Lemmas are used when proving these three Lemmas.

**Lemma A.1** The amount of memory needed to store \( X \) randomly chosen bits is \( X \) bits.

**Proof:** Assume that the \( X \) randomly chosen bits can be stored in \( Y < X \) bits of memory. The word created from the \( X \) bits can take \( 2^X \) different values. The \( Y \) bits field can represent only \( 2^Y \) different values. Thus, at least two \( X \)-bits-long words must have identical \( Y \)-bits-long representation, and therefore, the original \( X \) bits cannot always be uniquely extracted from their \( Y \)-bits-long representation. □

In Lemma A.2, \( N \) represents the number of nodes in the network. \( A_1, A_2, \ldots, A_N \) represent the initial value of the synchronous-protocol variable \( A_i \) at each of these nodes. \( X^1, X^2, \ldots, X^K \) represent values that should be stored in the local memory of a node (examples for such values are \( A_i(n) \) and the sum of the values of the early messages). \( V^1, V^2, \ldots, V^K \) are vectors that represent \( X^1, X^2, \ldots, X^K \) as linear combinations of \( (A_1, A_2, \ldots, A_N) \).
Appendix

A The proof of the order predicate lemma

In this section we provide a proof to Lemma 1, which says the following: Let $Syn$ be a fixed-overhead synchronizer and let $N$ be a hypercube, ring, or cubic torus network. There exists a synchronous protocol $P$ such that $Syn$ satisfies the following predicate for $P$ and $N$: for every node in $N$, all original-protocol messages sent to the node at pulse$(n)$ arrive at the node before any original-protocol message sent to it at pulse$(n+1)$.

Proof: (of Lemma 1) Since the memory overhead of $Syn$ does not depend on the specific synchronous protocol, it is fixed for the network $N$. Let $M_{Syn}$ denote the maximum over the nodes of the amount of memory, measured in bits, needed to store the synchronizer variables at the node. Let $M_T$ denote the amount of memory needed to store an image of the entire network topology and the current pulse number (which is between 0 and $D-2$).

Let $S$ be an integer such that $2M_{Syn} + M_T < S$. The protocol $P$ is defined as follows: each node $i$ maintains an integer variable $A_i(n)$ built of $S$ bits. At initialization time, each $A_i$ is given an arbitrary value independently from the others (the initial values of $A_i$ may possibly be given from the outside world). Each node knows the diameter $D$ of the network. The nodes perform $D-2$ pulses. At each pulse, each node $i$ sends the contents of $A_i$ to all its neighbors. When receiving a message from a neighbor, node $i$ adds its contents to the local variable $A_i$. We will show that $P$ is the protocol we are looking for: assume, in contradiction, that $Syn$ does not satisfy the predicate of the Lemma for $P$ and the network $N$.

Let $X$ be an execution of the protocol created by combining $P$ and $Syn$, in which the predicate of the Lemma is violated. Let $i$ be the node at which the predicate is first violated and $MSG$ be the first message to violate the predicate of the Lemma. Assume that node $i$ receives $MSG$ after performing $t_i(n-1)$ and before performing $t_i(n)$. Therefore, node $i$ has received all messages sent to it at pulse$(n-2)$. In order to violate the predicate of the Lemma, $MSG$ must have been sent by some neighbor $j$ at $t_j(n)$ and received by node $i$ before all messages sent to it at pulse$(n-1)$ are received.

When receiving $MSG$, node $i$ cannot perform $t_i(n)$ yet, since it has not received all messages sent to it at pulse$(n-1)$. Denote the value that node $i$ should send to all its neighbors at $t_i(n)$ by $A_i(n)$. Clearly, the value of $A_i(n)$ can be extracted from the local memory of node $i$ at time $t_i(n^-)$ $^1$. It is also clear that $A_i(n+1)$ can be extracted from the local memory at time $t_i(n+1^-)$. $A_i(n+1)$ is the sum of $A_i(n)$ and the values of the messages sent to $i$ at pulse$(n)$. We divide these messages into early messages and timely messages (the group of early messages is not empty). After $t_i(n)$, node $i$ receives only the timely messages. As proved in Lemma A.6, the sum of values of the early messages cannot be extracted from $A_i(n)$ and the timely messages.

$^1$For a given time $t$, we denote by $t-$ and $t+$ the time just before and just after $t$ respectively.
already performed \( t_j(n+1) \). When the message sent at \( t_j(n+1) \) from node \( j_1 \) to \( j_2 \) is processed, node \( j_2 \) has already performed \( t_j(n+1) \). This process continues, ending with the message sent by node \( j_{n-1} \) being processed at \( j_l = i \) when the latter has already performed \( t_i(n+1) \). Since \( Syn \) is a fixed-overhead synchronizer, it cannot save messages in buffers for a later use, thus original-protocol messages are processed in the code-block activated by their arrival. Thus, the string of messages described above is not delayed at any node: a node that receives a message from the string, immediately sends the next one. If \( MSG_i^1(n) \) is very slow, this string of messages may reach node \( l \) before \( MSG_i^1(n) \), contradicting Lemma 1.

\[ \square \]

**Theorem 1** The time complexity of any fixed-overhead synchronizer is no less than \( D \).

**Proof:** The proof of the theorem is similar to the proof of Theorem 4.8 in [ShS 1]. The time complexity of a synchronizer for a hypercube (ring, torus), when combined with a synchronous protocol, is the maximum time between \( t_i(n) \) and \( t_i(n+1) \) over all nodes and all \( n \geq 0 \), assuming the delay of all links is bounded by 1. Assume that this fixed-overhead synchronizer is combined with \( P \) of Lemma 2. This means that the combined protocol must satisfy the predicate of Lemma 2. Let \( u, v \in V \) be two nodes at distance \( D \). Assume \( t_u(n) > t_v(n) \). From Lemma 2, node \( v \) does 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, \( t_v(n+1) \geq t_u(n) + D \), and from the assumption that \( t_u(n) > t_v(n) \) follows \( t_v(n+1) \geq t_u(n) + D \).

\[ \square \]

### 5.2 Lower bounds on the memory overhead of synchronizers

As discussed in Sec. 4, the memory overhead of the known fixed-overhead synchronizers is \( O(d) \), while the memory overhead of all known variable-duplicating synchronizers (for arbitrary networks) is at least \( O(d) + S \) and the memory overhead of message-delaying synchronizers is at least \((d - 1)m + O(d)\). For the worst case original-protocol (such as \( P \)), \( m = S \) holds, and the memory overhead of a message-delaying synchronizer for such an original-protocol is \((d - 1)S + O(d)\).

Since all synchronizers are based on some kind of distributed protocol in which nodes communicate with their neighbors, we assume that all synchronizers (for arbitrary networks) need at least \( O(d) \) memory in order to operate. Thus, the fixed-overhead synchronizers \( \delta \) and \( \epsilon \) have optimal memory-overhead. Yet, we have proved in the previous section that fixed-overhead synchronizers have at least \( O(D) \) time complexity. From Corollary 1.1, a synchronizer that does not satisfy the predicate of Lemma 1 needs memory overhead of \( \Omega(S) \) in order to ensure proper operation. From Theorem 1 and Lemma 2, a synchronizer that satisfies the predicate of Lemma 1 and uses memory overhead of less than \( \Omega(S) \), has at least \( O(D) \) time complexity. Thus, all synchronizers that ensure proper operation with time complexity which is less than \( O(D) \), use memory overhead of \( \Omega(S) \). As already claimed, all synchronizers use some kind of a distributed protocol that needs \( \Omega(d) \) memory in order to operate. Therefore the lower bound for the memory overhead of synchronizers with time complexity which is less than \( O(D) \), is \( \Omega(d + S) \). This lower bound is achieved by variable-duplicating synchronizers.
may possibly be given from the outside world). Each node knows the diameter \( D \) of the network. The nodes perform \( D - 2 \) pulses. At each pulse, each node \( i \) sends the contents of \( A_i \) to all its neighbors. When receiving a message from a neighbor, node \( i \) adds its contents to the contents of \( A_i \) and stores the sum in \( A_i \). Since \( P \) is synchronous, this means that the value sent by \( i \) at pulse\( (n) \) is the sum of the contents of \( A_i \) just after pulse\( (n-1) \) and the contents of all the messages sent to \( i \) from all its neighbors at pulse\( (n-1) \).

Now suppose that \( Syn \) combined with \( P \) is performed on a network and suppose that node \( i \) receives an original-protocol message \( MSG_i^j(n) \) sent to it by a neighbor \( j \) at \( t_j(n) \), before receiving the original-protocol message \( MSG_i^k(n-1) \) sent to it by another neighbor \( k \) at \( t_k(n-1) \). Node \( i \) cannot perform \( t_i(n) \) upon receiving \( MSG_i^j(n) \) or before, since the values sent at \( t_i(n) \) must contain the contents of \( MSG_i^k(n-1) \). Upon receiving \( MSG_i^j(n) \), node \( i \) also cannot add its contents to \( A_i \), since in this case, the value sent at \( t_i(n) \) would be incorrect. However, \( i \) must add the contents of \( MSG_i^j(n) \) to \( A_i \) sometime before performing \( t_i(n+1) \), and therefore, it has no choice but to save it in some way until after \( t_i(n) \) is performed, and for this we need at least \( S \) bits. This means that the memory overhead of the synchronizer is at least \( S \) bits, and thus, the synchronizer is not a fixed-overhead synchronizer.

The proof given in Appendix A shows that in hypercubes, rings, and cubic toruses, it is not possible to encode \( A_i \) and the early messages with less than \( 2S \) bits. \( \square \)

**Corollary 1.1** A synchronizer that does not satisfy the predicate of Lemma 1, needs memory overhead of at least \( \Omega(S) \) in order to ensure proper operation.

**Corollary 1.2** For \( Syn \), \( P \) and a network as defined in Lemma 1, an original-protocol message sent to a node \( i \) at pulse\( (n) \) is processed by \( i \) only after it performs \( t_i(n) \).

**Lemma 2** Let \( Syn \) be a fixed-overhead synchronizer. There exists a synchronous protocol \( P \) such that \( Syn \) satisfies the following predicate for \( P \) and for every hypercube, every ring, and every cubic torus: no node performs pulse\( (n+1) \) before all nodes are safe with respect to pulse\( (n) \).

**Proof:** The proof of the Lemma is similar to the proof of Theorem 4.6 in [ShS 1]. Let \( P \) be the synchronous protocol selected for \( Syn \) in Lemma 1. 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 \( i \), that has not arrived yet at time \( t_i(n+1) \).

Since hypercubes, rings, and cubic toruses are 2-connected, there exists at least one path \( i = j_0 \rightarrow j_1 \rightarrow j_2 \rightarrow \ldots \rightarrow j_{q-1} \rightarrow j_q = i \) that does not contain node \( k \).

From corollary 1.2, when \( P \) combined with \( Syn \) is executed, an original-protocol message sent to a node at pulse\( (n+1) \) is processed by the node only after the latter has performed pulse\( (n+1) \).

Note that in \( P \), every node sends a message to each of its neighbors at each pulse. When the message sent by node \( i = j_0 \) to \( j_1 \) at time \( t_i(n+1) \) is processed at node \( j_1 \), the latter has
5 Impossibility Results

5.1 Lower bound on the time complexity of fixed-overhead synchronizers

The two known fixed-overhead synchronizers, $\delta$ and $\epsilon$, have time complexity of $O(D)$. In this section we prove that $D$ is the lowest time complexity that can be achieved by a fixed-overhead synchronizer.

We first sketch the proof. The proof is given in three steps: in Lemma 1 we prove that for every fixed-overhead synchronizer and every hypercube network, there exists an original synchronous protocol for which the following is satisfied: no original-protocol message sent at pulse$(n + 1)$ arrives at a node before a message sent at pulse$(n)$. In Lemma 2 we prove that in hypercube networks, a fixed-overhead synchronizer must satisfy the following predicate for the original-protocol found in Lemma 1: no node performs pulse$(n + 1)$ before all nodes in the network are safe with respect to pulse$(n)$. The time complexity is proved for hypercube networks and for the original-protocol found for the synchronizer in Lemma 1. This is sufficient since the time complexity of the synchronizer is defined as the maximum over all networks and all original synchronous protocols. All the characteristics of hypercube networks used in this proof also hold for rings and cubic torus networks.

Definition 5.1 (satisfy a predicate for a protocol and a network)

A synchronizer satisfies a predicate for a synchronous protocol $P$ and a network $N$, if every execution of $P$ combined with this synchronizer on the network $N$ satisfies this predicate.

The two known fixed-overhead synchronizers, $\delta$ and $\epsilon$, share one common behavior: each of these synchronizers require the designation of one node, called $s$, that executes a different algorithm than the other nodes in the network. After each pulse, the nodes perform a protocol that gives to node $s$ the information that all nodes are safe with respect to the current pulse. After receiving this information, node $s$ initiates a protocol that triggers the next pulse at all nodes. In other words, the known fixed-overhead synchronizers satisfy the following predicate: no node performs pulse$(n + 1)$ before all nodes are safe with respect to pulse$(n)$. In Lemma 2 we show that all fixed-overhead synchronizers must satisfy this predicate for hypercube (ring, torus) networks and at least one synchronous protocol. Lemma 1 is used for proving Lemma 2.

Lemma 1 Let $Syn$ be a fixed-overhead synchronizer. There exists a synchronous protocol $P$ such that $Syn$ satisfies the following predicate for $P$ and every hypercube, every ring, and every cubic torus: all original-protocol messages sent to any node at pulse$(n - 1)$ arrive at the node before any original-protocol message sent to it at pulse$(n)$.

Sketch of proof (the proof is given in Appendix A): the synchronous protocol $P$ is selected as follows: each node $i$ maintains a variable $A_i$ containing $S$ bits, where $S >> M_{Syn}$. At initialization time, each $A_i$ is given an arbitrary value independently from the others (the initial values of $A_i$
end of the \((2n+1)\)-th and \((2n+2)\)-th execution of the protocol at node \(i\) is denoted by \(t_i(n)\) and \(T_i(n)\) respectively \((n = 0, 1, 2, \ldots)\).

Original-protocol messages are sent by node \(i\) only at times \(t_i(n)\). Messages sent by node \(i\) to a neighbor \(j\) at time \(t_i(n)\) are received by \(i\) after \(T_i(n - 1)\) and before \(T_i(n)\). At time \(T_i(n)\), the original-protocol variables are copied into a backup variables set and then the pulse is performed, without sending messages. The messages are sent at \(t_i(n + 1)\) and are built based on the variables in the backup set (for more details, see \([\text{ShS 1}]\)).

The memory overhead of variable-duplicating synchronizers is composed of two parts: the memory needed for the synchronization protocol and the memory needed for the backup set of variables. The amount of memory occupied by the latter part is exactly \(S\). Therefore, the memory overhead of the variable-duplicating versions of the different synchronizers presented in Sec. 4.2 is:

- synchronizers \(\alpha, \beta, \gamma\) and the simple version of \(\alpha\): \(S + O(d)\).
- the synchronizer for the hypercube: \(S + O(1)\).
- the synchronizer with polylogarithmic overhead: \(S + O(d) + O(polylog|V|)\).

### 4.4 Conclusions

Among all known synchronizers that work for arbitrary networks, the fixed-overhead synchronizers are by far the best in terms of memory overhead.

The memory overhead of variable-duplicating synchronizers cannot be compared with that of message-delaying synchronizers. The reason is that the first depends on the amount of memory used by the synchronous protocol, while the second depends on the length of the synchronous-protocol messages. However, in the worst case, a node sends an original-protocol message that contains a complete image of its local state, and therefore, the length of the synchronous-protocol messages is \(S\) (e.g. the ADD family presented in Sec. 5). Thus, the worst-case memory overhead of message-delaying synchronizers is \((d - 1)\) times larger than the memory-overhead of variable duplicating synchronizers (assuming that \(S >> d\)).
external degree of the nodes to be no more than a given value. When the partition algorithm is
given parameters $k$ and $n$, the memory overhead of $\zeta$ is no more than $\min(d - 1, \frac{|V|}{n}) \times m + O(d)$,
its time and communication complexities are $O(\log_k |V| + n)$ and $O(k|V|)$ respectively.

- **a synchronizer for the hypercube [PU 1]**: The idea of this synchronizer is based on the use of a 3-
spanner with $O(7|V|)$ edges for the hypercube. A 3-spanner for a graph $G$ is defined as a subgraph
$G'$ of $G$ that contains all the nodes and part of the edges of $G$ and satisfies the following predicate:
let $v$ and $v'$ be two arbitrary neighbors in $G$, the distance between $v$ and $v'$ in $G'$ is no more
than 3. The synchronizer uses the $\alpha$ mechanism on the edges of $G'$ three times between each two
consecutive pulses.

The communication and time complexities of this synchronizer are $O(|V|)$ and $O(1)$ respectively.
Again, communication (of three SAFE messages per pulse) occurs between neighbors only. The
amount of edges that participate in the synchronizer protocol at each node is $O(1)$. Thus, the
memory-overhead of this message-delaying synchronizer is $(d - 1)m + d + O(1)$.

- **a simple version of $\alpha$**: appears in [LT 1] and [ER 1]. The idea is to force each node to send messages
to all neighbors at each pulse. Each node waits until it receives messages of the former pulse from
all adjacent links. The memory overhead of this synchronizer is $(d - 1)m + O(d)$.

- **a synchronizer with polylog time and communication overhead [AP 1]**: the synchronization protocol
used by this synchronizer is complicated. We will mention, however, some of the main properties
of this synchronizer: it works only for synchronous protocols in which nodes that do not receive
messages sent at pulse$(n - 1)$ are not active at pulse$(n)$. The communication complexity of the
synchronizer differs from pulse to pulse: assuming that the number of nodes that are active at
pulse$(n)$ is $C(n)$, the average number of messages sent by the synchronizer protocol at pulse$(n)$ is
$O(C(n) \times \log^3 |V|)$. The average time complexity per pulse of the synchronizer is $O(\log^3 |V|)$. The
memory-overhead of the synchronizer, according to the authors, is $(d - 1)m + O(d) + O(polylog|V|)$.

- in [CCGZ 1] and [KTZ 1], the authors present synchronizers for asynchronous bounded-delay net-
works. The memory overhead of the message-delaying versions of these synchronizers is $(d-1)m + d$.
We will not discuss these synchronizers in detail, since they use an asynchronous model that is dif-
ferent from the one treated in this paper.

### 4.3 Variable-duplicating synchronizers

All synchronizers presented in Sec. 4.2 can be transformed into variable-duplicating synchronizers
without affecting the order of their communication or time complexities.

All these synchronizers involve a synchronizer protocol which begins when the nodes are safe
and ends at each node when it knows that all its neighbors are safe. The technique of transform-
ing a synchronizer into a variable-duplicating synchronizer works as follows: variable duplicating
synchronizers perform the synchronizer protocol twice between every two consecutive pulses. The
more than one message per link can be early, the maximum number of delayed messages at any given time is \((d - 1)\). Thus, the amount of memory needed for saving delayed messages is \((d - 1)m + d\). Here \(d\) is the amount of memory needed for the occupied/not-occupied flags which maintain the data structure of the saved messages.

The following synchronizers allow early original-protocol messages. These synchronizers were constructed with the concept of message delaying in mind.

- **synchronizer \(\alpha\) [Awer 1]**: After each pulse, SAFE messages are sent by each node to all its neighbors as soon as it becomes safe. A node which receives SAFE messages from all neighbors, performs the next pulse. The communication and time complexities of \(\alpha\) are \(O(|E|)\) and \(O(1)\) respectively. Since the communication is done only between nodes and their neighbors, the amount of memory needed for the synchronizer-protocol variables is \(O(d)\). The message-delaying version of \(\alpha\) uses \((d - 1)m + d\) memory for saving early messages, and \(O(d)\) memory for the synchronizer-protocol variables. Thus, the memory overhead of the synchronizer is \((d - 1)m + O(d)\).

- **synchronizer \(\beta\) [Awer 1]**: An initialization phase creates a spanning tree for the network graph. After each pulse, SAFE messages are broadcast along the spanning tree to the root node. Then AWAKE messages are broadcast back along the tree, triggering the next pulse at the nodes. The communication and time complexities of \(\beta\) are \(O(|V|)\) and \(O(D)\) respectively. Again, the communication is only between neighbors; thus, the memory overhead of \(\beta\) is \((d - 1)m + O(d)\).

- **synchronizer \(\gamma\) [Awer 1]**: An initialization phase creates a spanning forest for the network graph. The subgraph created by each tree in the forest is called a ‘cluster’. A preferred link is selected between every two neighboring clusters. After each pulse, SAFE messages are converged along each tree to the root. At this point, CLUSTER_SAFE messages are broadcast along each tree edge and preferred links, telling all neighboring clusters that this cluster is safe. The information about neighboring clusters being safe is brought to the root node of each cluster by CLUSTER_READY messages that converge along each tree, and then the root node initiates a broadcast of AWAKE messages along the cluster tree, which cause the nodes in the cluster to perform the next pulse.

The communication and time complexities of \(\gamma\) are \(O(k|V|)\) and \(O(\log_k(|V|))\) respectively (\(k\) is a parameter given at initialization time). For the same reasons as in \(\alpha\) and \(\beta\), the memory overhead of \(\gamma\) is \((d - 1)m + O(d)\).

- **synchronizer \(\zeta\) [ShS 3]**: This is a message-delaying version of \(\gamma\) that enables tradeoff between time complexity and memory overhead. The main idea is to use, inside each cluster, the protocol of \(\epsilon\) instead of the protocol of \(\beta\). This ensures that early original-protocol messages arrive to the nodes only from edges that lead to nodes in other clusters (external edges). Messages received from edges that lead to nodes in the same cluster are always timely.

The number of external edges connected to a node is called the external degree of the node. A special partition algorithm is used with this synchronizer. This partition algorithm limits the
4 The Memory Overhead of known Synchronizers

4.1 Fixed-overhead synchronizers: $\delta$ and $\epsilon$

Recall that a fixed-overhead synchronizer is a synchronizer in which the memory overhead is independent of the specific original-protocol and is a function of the network topology only. We next show that synchronizers $\delta$ [LT 1] and $\epsilon$ [ShS 1] are fixed-overhead synchronizers.

In synchronizer $\delta$, an initialization phase creates a directed spanning tree for the network graph. After performing pulse$(n - 1)$, SAFE messages are sent along the spanning tree from the leaves to the root node: each node that is safe and has received SAFE messages from all its children in the spanning tree, sends a SAFE message to its parent. 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 are ready to perform pulse$(n)$. At this time, the root node sends AWAKE messages to all its neighbors and performs pulse$(n)$. When receiving the first AWAKE message, each other node immediately sends AWAKE messages to all its neighbors and performs pulse$(n)$. This procedure ensures that no node receives an original-protocol message sent at pulse$(n)$ before receiving an AWAKE message and performing pulse$(n)$.

Synchronizer $\delta$ ensures that there are no early original-protocol messages and therefore needs no memory for saving such messages. Thus, $\delta$ needs additional memory only for the synchronizer variables: the type of the last message received from each link and the current status of the node. Therefore, the memory overhead of synchronizer $\delta$ is $O(d)$, where $d$ is the maximal node degree. The time and communication complexities of $\delta$ are $O(D)$ (where $D$ is the diameter of the network) and $O(|E|)$ respectively.

Synchronizer $\epsilon$ [ShS 1] also uses a spanning tree created at the initialization phase. As in $\delta$, after each pulse, SAFE messages are sent along the spanning tree towards the root node. Synchronizer $\epsilon$ differs from $\delta$ in the way the root node causes all other nodes to perform the next pulse. Instead of sending AWAKE messages to all neighbors before performing the pulse, each node sends AWAKE messages only to its children in the spanning tree. This means that nodes may receive original-protocol messages before receiving an AWAKE message. In this case, the node that receives the early message first performs the pulse and only afterwards processes the message. The early message need not be saved, since it is processed by the code activated by its arrival and before any other message is processed. However, the code is performed in the indicated order — the pulse is performed before the message is processed.

For the same reasons as in $\delta$, the memory overhead of $\epsilon$ is $O(d)$. The time and communication complexities of $\epsilon$ are $O(D)$ and $O(|V|)$ respectively.

4.2 Message-delaying synchronizers

The memory overhead of message-delaying synchronizers consists of two parts: the memory needed for the synchronizer protocol and the memory needed for saving the delayed messages. Since no
execution of $P$ and an execution of $P'$ on the same topology and with the same initial local states 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 $t_i(n)$ (where $i$ is the node sending the message), and if the executions are finite, their final outcome is the same.

**Definition 3.8 (implementation)**

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

**Definition 3.9 (proper operation)**

A synchronizer ensures proper operation if combined with any synchronous protocol $P$, it produces an asynchronous protocol $P'$ that is an implementation of $P$.

Another term, that is used in most previous works on synchronizers, is ‘safety’:

**Definition 3.10 (safe)**

In a combined protocol, a node $i$ is said to be safe with respect to $\text{pulse}(n)$, 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.
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 of 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 $\{t_i(n), n \geq 0\}$.

A synchronizer 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 the 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 (satisfy a predicate)
A synchronizer satisfies a predicate, if for every synchronous protocol $P$, every execution of $P$ combined with this synchronizer satisfies this predicate.

Definition 3.4 (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 3.5 (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 3.6 (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 3.7 (equivalent)
Let the asynchronous-protocol $P'$ be the synchronous-protocol $P$ combined with a synchronizer. An
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 has been received.

Similarly to the synchronous model, messages are processed at each node in the received order, even if they were received from different links. A node can process an earlier received message, after processing later received messages, only if it was saved in the local variables when received.

In the sequel, when analyzing the memory demands of a protocol at each node, we do not count the memory used by lower or higher layers (e.g., DLC, transport, etc.).

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.

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.

Correctness of synchronizers is discussed in details in [ShS 1]. We say that a synchronizer is correct if it ensures proper operation. Below we give formal definitions for synchronizers and for the term 'ensures proper operation'.

Definition 3.1 (synchronizer)
available to the code at the time of the pulse, unless saved in the local variables when received.

Notice that this synchronous model is message-and-pulse driven, meaning that code is performed when receiving a message, as well as when a pulse occurs. Another possible synchronous model is the pulse driven model, where received messages are held in buffers and no code is executed until the next pulse. When the next pulse occurs, the code performed at the pulse accesses these buffers and processes them.

In this paper, we use the message-and-pulse driven model because in many cases, it is much more efficient in terms of memory usage. The message-and-pulse driven model is used in previous works on synchronizers, like [Awer 1], [Awer 2], [LT 1], [PU 1], [AP 1], [LTC 1], [AIR 1], [ShS 1], [ShS 2] and [ShS 3].

**Definition 2.1** (pulse(n))

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

**Definition 2.2** (phase(n))

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

**Assumption 2.1** In a synchronous network, at time $pulse(n)$, 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], [ShS 1], [ShS 2] and [ShS 3], 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 2.3** (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 $t$. $\Pi_i[X](t)$ denotes the local state of node $i$ at time $t$ in the execution $X$.

### 2.2 The Asynchronous Model

In the asynchronous model, when a node receives a message from some neighbor or from the outside world, it 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 [BS 1], which in our context means that they arrive in
All known synchronizers use at each node additional variables, over and above those of the original protocol, and thus add memory overhead. In Sec. 4, we discuss the different synchronization techniques mentioned above and their memory overhead. We show that message-delaying synchronizers and variable-duplicating synchronizers use a large amount of memory. In message-delaying synchronizers, most of this memory is used for storing delayed messages, while in variable-duplicating synchronizers, most of this memory is used for maintaining a backup copy of the original-protocol variables.

We define fixed-overhead synchronizers as synchronizers in which the memory overhead is not a function of the specific original-protocol, but only of the network topology. In Sec. 4 it is shown that synchronizers \( \delta \) and \( \epsilon \) are fixed-overhead synchronizers, while variable-duplicating and message-delaysing synchronizers are not. In Sec. 5 we prove that \( \delta \) is the lower bound for the time complexity of fixed-overhead synchronizers. Therefore, synchronizer \( \epsilon \) is the best fixed-overhead synchronizer in terms of time and communication complexities per pulse.

The memory overhead of message-delaying synchronizers cannot be compared with that of variable-duplicating ones. The memory overhead of a message-delaying synchronizer depends on the size of the original-protocol messages as well as on the maximum node degree in the particular network. On the other hand, the memory overhead of variable-duplicating synchronizers is the amount of memory needed to store a backup of the original-protocol variables. However, in Sec. 5 we prove that variable-duplicating synchronizers have optimal memory overhead in the following sense: there exists a family of synchronous protocols which cannot be synchronized with memory overhead that is less than the memory overhead of variable-duplicating synchronizers, and with time complexity less than \( \delta \).

2 The models

2.1 The Synchronous Model

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.

Each node processes each message when it receives it. At each pulse, each node may send 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. There is one additional code-block that is 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 links. The contents of any received message is not
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, 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], [KTZ 1], [ER 1], [ER 2], [RS 1], [AS 1], [ShS 1], [ShS 2] and [ShS 3]. 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] and [MR 1].

All synchronizers ensure that a node may perform a new pulse when it knows that it has received all the 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 all nodes perform simultaneously the pulse, and messages sent at a certain pulse arrive 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.

Several techniques for solving this problem are known:

- **message delaying**: early messages are not processed immediately. Instead, these messages are saved and processed only after the pulse is performed. This technique is discussed and used in [LT 1], [FLS 1], [ER 1], [Awer 1], [KTZ 1], [ShS 1], [ShS 2] and [ShS 3]. The technique can be applied with any synchronizer.

- **variable duplicating**: such synchronizers ensure that there are no early messages. This is done by sending original-protocol messages only some time after the pulse is performed. In order to do that, a copy of the original-protocol variables is saved at the time of the pulse and used when the messages are sent. A special protocol is used to decide the timing of sending the messages. This technique, which is introduced in [ShS 1], can be applied with any synchronizer without affecting the time or communication complexity. Examples of variable-duplicating synchronizers are $\alpha_1'$ and $\gamma_1'$ presented in [ShS 1].

- **synchronizer $\delta$**: this synchronizer also ensures that there are no early messages. It does so by imposing the order in which nodes perform pulses. This synchronizer protocol uses $2|E| + V$ messages and $D$ time units per pulse. The synchronizer is first introduced in [LT 1] and described in more detail in Sec. 4.

- **synchronizer $\epsilon$**: This synchronizer uses a synchronization mechanism whose purpose is to ensure that no node performs pulse$(n)$ before all other nodes have received all the former pulse messages and thus are ready to perform pulse$(n)$. Each node that receives an original-protocol message that is early for pulse$(n)$, first performs pulse$(n)$ and only then processes the message. This synchronizer is introduced in [ShS 1] and uses $|V|$ messages and $D$ time units per pulse. Synchronizer $\epsilon$ is described in more detail in Sec. 4.
network topology parameters: $|V|$, $|E|$, the maximum node degree $d$ over the network, etc.

- $S$: the maximum over the nodes in the network, of the amount of memory necessary to store the original-protocol variables.

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

We define fixed-overhead superimposed protocols as superimposed protocols with memory overhead that does not depend on the original-protocol they are combined with, but only on the topology of the network.

Fixed-overhead superimposed protocols have the following advantages:

1. fixed-overhead superimposed protocols can be implemented as a part of the operating system. The amount of memory needed for the superimposed protocol can be added to the system without knowing the properties of the original-protocol it will be combined with.

2. as seen in the sequel, in many cases, fixed overhead superimposed protocols have better memory overhead than non-fixed overhead ones.

Memory overhead of superimposed protocols is discussed in [ShS 1], [APPS 1], [ShS 3] and [ShS 4]. [ShS 1] presents synchronizers with low memory overhead. [ShS 3] presents a message delaying synchronizer that provides tradeoff between memory overhead and time complexity. [APPS 1] presents a reset procedure that is based on a dynamic synchronizer, and analyzes its memory overhead. The present paper discusses and demonstrates advantages (1) and (2) above for synchronizers. [ShS 4] discusses the memory overhead of distributed snapshots.

The present paper discusses the memory overhead of synchronizers. Synchronizers are tools for transforming protocols written for a synchronous model to protocols that run on an 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 referred to as a time unit. The network contains a global clock that ticks synchronously at all network nodes at time unit intervals. Code can be performed upon receipt of a message and/or at clock ticks. Nodes are allowed to send messages only at clock ticks.

The synchronous protocol which is given to the synchronizer as input will be referred to as the original protocol. 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 and sends messages that are identical to the original-protocol messages. In certain circumstances, slight changes in the pulse code and/or in message format are allowed (see [ShS 1]).

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
1 Introduction

Consider a network described by an undirected graph \( N = (V, E) \), 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. The nodes do not share memory and communicate only by means of messages sent along the communication links.

We are mainly interested in the memory complexity (also known as space complexity) of distributed protocols designed for such networks. The memory complexity of a protocol \( P \) on a network \( N \) is defined as the maximum over the nodes of \( N \), of the amount of memory needed to store the local variables used by \( P \). The memory complexity of a protocol \( P \) on a network \( N \) depends on the topology of the network.

Memory complexity of distributed protocols and tradeoffs between memory, time and communication complexities have been discussed in previous works: tradeoffs between the memory occupied by routing tables and the efficiency of the routes created by them are discussed in [KK 1], [PU 2] and [ABLP 1]. A tradeoff between the number of buffers used by a store-and-forward deadlock prevention scheme and the efficiency of the created routes is discussed in [AKP 1].

In [LTT 1], the authors present tradeoffs between communication and memory complexities of distributed protocols for matrix multiplication, convolution of two polynomials and multiplication of a matrix by a vector. The considered system consists of two processors connected by a communication link. In addition, the authors present problems for which there is no tradeoff between communication and memory complexities. These problems can be solved with optimal communication and memory simultaneously. The results are further formalized and generalized in [BTY 1].

In the present paper, we deal with a special type of distributed protocols, referred to as superimposed protocols, namely protocols that do not have a meaning as stand-alone protocols, but rather are combined with other protocols. Examples of superimposed protocols are: termination detection protocols [Fran 1], [DS 1], [Rana 1], [Mis 1], [CM 1], [Erik 1], deadlock detection and resolution protocols [CM 2], [CMH 1], [BT 1], [AM 1], [RBC 1], distributed snapshots [CL 1], [SK 1], synchronizers (discussed in detail in the sequel), distributed schedulers [MR 1], and reset procedures for dynamic networks [Finn 1], [Seg 1], [SH 1], [AAG 1], [Awer 4].

The memory complexity of superimposed protocols is not defined, since these are not stand-alone protocols. We can only refer to the memory complexity of the combined protocol, created when combining the superimposed protocol with an original protocol. Generally speaking, the memory overhead of a superimposed protocol is the amount of memory required by the combined protocol over and above the memory needed by the original protocol.

The memory overhead of a superimposed protocol for a network \( N \) and an original protocol \( P \) is defined as the maximum over the nodes of \( N \) of the memory needed for the superimposed-protocol variables and buffers, when combined with \( P \). The memory overhead of a superimposed protocol is a function of:
ON THE MEMORY OVERHEAD OF 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

September 7, 1993

Abstract

Memory overhead of superimposed protocols is discussed. The memory overhead depends on the network topology as well as on the protocol these protocols are combined with. Fixed-overhead superimposed protocols are defined as superimposed protocols for which the memory overhead depends only on the network topology.

Memory overhead of synchronizers is discussed in detail: the memory overhead of known synchronizers is analyzed. A tradeoff between the memory overhead of synchronizers and their time and communication complexities is presented.