


References


6.2 Other synchronization methods

An Asynchronous Pacer Protocol is one method of limiting the possible executions of the photographed protocol in order to achieve bounded memory overhead for the distributed snapshot protocol.

The APP has a drawback when it is used in networks in which link delays have a large variance. In such cases, a message that is delayed for a long time may slow down the entire network: the next pulse is not performed before the messages of the former are received. To be more precise, when a message sent to a node $i$ at $\text{pulse}(n)$ is delayed, this node cannot perform $\text{pulse}(n+1)$, its neighbors cannot perform $\text{pulse}(n+2)$, their neighbors cannot perform $\text{pulse}(n+3)$ and so on.

One way of overcoming this drawback is by allowing a wider range of executions. For example, a network in which message delays are $1, 2, 3, \ldots K$ can be simulated by using distributed schedulers [MR 1]. A distributed scheduler customized for this task is an extended version of a synchronizer, in which a node can perform $\text{pulse}(n)$ if all messages sent to it at $\text{pulse}(n-K)$ have already arrived.

While achieving less sensitivity to variance in the communication delays, the memory overhead of distributed snapshots combined with such schedulers is at least $K$ times larger than the memory overhead of distributed snapshots combined with an APP.
(2) if $N$ is the three nodes network $i \rightarrow j \rightarrow k$ introduced in Sec. 2, and $P$ is the protocol described earlier, in which $j$ and $k$ exchange a token and $j$ sends RCVD_TOKEN messages to $i$, then $Z(P, N) = 1$ again.

(3) if $P$ is the Gallager minimum hop protocol [Gal 1], $Z(P, N) = d$ for any network $N$.

Now, assume that a node $i$ takes a snapshot of its local memory and sends a MARKER message to each of its neighbors at time $T_i = pulse_i(n)$. These MARKER messages are received by the neighbors of $i$ before they perform $pulse(n + 1)$. Each neighbor $j$ of $i$ processes the MARKER message sent from $i$ at $pulse_j(n + 1)$. Thus, for each neighbor $j$, the time when $j$ takes a snapshot of its own memory and sends MARKER messages is $T_j \leq pulse_j(n + 1)$. Therefore, node $i$ processes the MARKER message sent by each of its neighbors no later than at time $pulse_i(n + 2)$.

The messages recorded by node $i$ as being on each link are the messages received on that link and processed after $T_i = pulse_i(n)$ and before a MARKER message has been received from that link. The only messages possibly processed by node $i$ during this time interval are those sent to $i$ at pulses $n - 1$, $n$ and $n + 1$ (these messages are processed by $i$ at $pulse_i(n)$, $pulse_i(n + 1)$, and $pulse_i(n + 2)$ respectively). Thus, the amount of memory needed by the distributed snapshot protocol in order to record messages on incoming links is $3m \times Z(P, N)$ ($m$ is the size of the largest original-protocol message). The memory-overhead of the distributed snapshot protocol contains also the amount of memory required to record the local state of the node (which is $S$) and the amount of memory used by the distributed snapshot protocol itself (which is $O(d)$). The resulting memory overhead is $O(d + S + 3m \times Z(P, N))$.

The amount of memory needed for the stage of collecting the global information is $O(d + S \times |V| + 3m \times Z(P, N) \times |V|)$, since at least one node should be able to store the information recorded by all the nodes.

The APP requires the following memory overhead: a node $i$ may receive during $(pulse_i(n)$, $pulse_i(n + 1))$ messages sent at $pulse(n)$ and messages sent at $pulse(n + 1)$ (see [ShS 2]). Thus, during this time interval, node $i$ has to save at most $2 \times Z(P, N)$ messages. Therefore, each node needs at most $2 \times Z(P, N)$ buffers in order to save messages. Since each buffer is of size $m$ at the most, the amount of memory needed by the APP for saving messages is $2m \times Z(P, N)$. Extra $O(d)$ memory at each node is needed in order to perform the synchronization protocol. Thus, the memory overhead of the APP is $O(d + 2m \times Z(P, N))$.

To sum up, a bounded memory overhead distributed snapshot protocol can be constructed by using a simple distributed snapshot protocol combined with an Asynchronous Pacer Protocol. The total memory overhead of the distributed snapshot and the APP is $O(d + S + 5m \times Z(P, N))$, as opposed to unbounded memory overhead of the distributed snapshot alone.

If there is a need for collecting the recorded global information, the total memory overhead is $O(d + S \times |V| + (2m + 3m|V|) \times Z(P, N))$. 

While processing these messages, a node may send messages to some or all of its neighbors.

Now the process repeats: each node waits to receive an acknowledgement for each $P'$ message, and sends SAFE messages to all neighbors. Meanwhile, $P'$ messages received by each node are saved in buffers. Upon receiving a SAFE message from each neighbor, some of the messages saved in the buffers are processed, and so on.

We denote the time when a node $i$ sends AWAKE messages to all neighbors by $\text{pulse}_i(0)$. We denote the $n$-th time in which node $i$ receives SAFE messages from all neighbors and processes messages held in local buffers by $\text{pulse}_i(n)$. A suffix is added to each $P'$ message sent. The suffix contains the number of the pulse at which the message is sent (in fact, a one bit suffix is sufficient, see [ShS 2]). At time $\text{pulse}_i(n)$, node $i$ does not process all the $P'$ messages held in its buffers, but only those that were sent to $i$ by neighbors when performing $\text{pulse}(n-1)$.

An Asynchronous Pacer Protocol simulates a network in which all messages are delayed for exactly one period of time: let $X'$ be an execution of $P'$ in such a network, and $X''$ be an execution of $P'$ combined with an APP on a network with the same topology but arbitrary delays. In $X''$, the messages of $P'$ sent by a node $i$ at $\text{pulse}_i(n)$ (for any integer $n$), are the same as the messages sent in $X'$ at time $t = n$. The same goes for the local state at the nodes in both executions.

The APP described above uses a synchronization protocol that is similar to the synchronization protocol used by synchronizer $\sigma$. Therefore, its communication complexity overhead is $O(|E|)$ messages per pulse. In fact, an APP can be constructed by using any other known synchronization protocol, such as that of $\beta$ or of $\gamma$, thus reducing the communication complexity overhead for a penalty of increased time complexity overhead. Poly-logarithmic overhead in both complexities can be achieved by using the synchronization protocol of [AP 1].

### 6.1 The memory overhead of distributed snapshots when using an Asynchronous Pacer Protocol

In this section we discuss the memory overhead of distributed snapshots combined with an Asynchronous Pacer Protocol: let $P$ be an asynchronous protocol, $P'$ the combination of $P$ with the distributed snapshot protocol, and $P''$ the combination of $P'$ with the APP.

**Definition 6.1** Let $N = (V, E)$ be a network and $P$, $P'$ and $P''$ be as described above. $Z(P, N)$ is defined as the maximum number of original-protocol messages (i.e. messages belonging to $P$) processed by any node $i \in V$ at any pulse when performing $P''$.

Observe that $Z(P, N)$ is also the maximum number of original-protocol messages received by any node at any integer time, when the protocol is executed on a network $N$ in which all link delays are exactly 1.

For example:

1. if $P$ is the DFS protocol, $Z(P, N) = 1$ for any network $N$. 

since the upper-bound of the amount of memory needed by the protocol for storing the created snapshot depends only on the topology of the network.

5 Alternative protocols

One can also consider using alternative protocols for solving particular problems. For example, fixed-overhead protocols for termination detection in the asynchronous model can be found in [DS 1], [Mis 1], [CM 1], [Erik 1], and fixed-overhead protocols for deadlock detection can be found in [CM 2], [CMH 1], [BT 1], [AM 1], [RBC 1].

In [HJPR 1], a generalized protocol that can detect a fairly wide range of stable properties, is presented. Theoretically, this protocol has unbounded memory overhead, since it uses unbounded counters. Yet, in most practical applications, bounded counters occupying a fairly small amount of memory are sufficient.

6 A bounded memory-overhead distributed snapshot protocol

A different way of bounding the memory overhead of distributed snapshots is by limiting the photographed protocol to executions whose distributed snapshot requires only bounded memory. Limiting the possible executions of a protocol does not affect its correctness: all executions of a correct protocol are correct.

One way of limiting the possible executions of the protocol is by using an Asynchronous Pacer Protocol (APP). An asynchronous Pacer Protocol works similarly to a synchronizer [Awer 1], except that its input is an asynchronous protocol, rather than a synchronous one. In the sequel we show that the use of an APP limits the possible executions of the protocol such that the distributed snapshot requires bounded memory.

In order to understand how an Asynchronous Pacer Protocol (APP) works, we describe an APP that works similarly to synchronizer α [Awer 1]. Let $P'$ denote the asynchronous protocol given as input to the APP. Let $I$ denote the set of nodes that initiate the protocol. The execution of the APP combined with $P'$ is as follows: first, the nodes in $I$ flood the network with AWAKE messages (using a Propagation of Information protocol [Seg 1]). After sending the AWAKE messages, each node in $I$ performs the code of $P'$, sending $P'$ messages to some or all of their neighbors. Each node that receives a message of $P'$ saves it in a local buffer and sends back an acknowledgement. Upon receiving acknowledgments for all messages sent, each node in $I$ sends SAFE messages to all its neighbors. Each node not in $I$ sends SAFE messages to all neighbors immediately upon receiving the first AWAKE message.

Upon receiving a SAFE message from all neighbors, each node processes some of the messages of $P'$ received earlier and saved in the local buffers as if these messages were just received. Note that not all messages are processed, but only those sent at the previous pulse, as defined presently.
memory needed for the distributed-snapshot protocol.

Therefore, it is better to have a distributed snapshot protocol which uses bounded memory in the asynchronous model. Such a distributed snapshot protocol uses bounded memory in real systems as well, independently of the specific characteristics of the links. The following sections suggest some solutions for this problem.

4 Fixed-overhead special-purpose versions of distributed snapshots

In many cases, the distributed snapshot is used for a specific task, like detection of termination, deadlock or some other stable predicate. In such applications, the distributed snapshot protocol creates a global image of the network at one (or more) specific node. The node uses this image of the network state in order to compute the value of the needed predicate. However, it may turn out that not all the information gathered is necessary for computing the value of the predicate. A special-purpose distributed-snapshot protocol, which gathers only the needed information, may be used in this case. Such a special-purpose protocol would require less memory overhead in order to store the created network image.

For example, suppose the reason for the distributed snapshot is that the global image created is used for detecting termination. An asynchronous protocol is said to be terminated if the following two conditions are fulfilled:

1) there are no messages in transit on the links of the network.
2) no node is currently processing a message that has just been received.

Therefore, there is no reason for storing the contents of the messages in each link, or even the number of messages in each link. In fact, one flag per link, saying whether it is empty or not, is sufficient. The amount of memory needed for storing these flags depends only on the topology of the network and not on the photographed protocol. Thus, the created distributed snapshot protocol is a fixed-overhead protocol.

Another example is when distributed snapshots are used for deadlock detection. In this problem, each node can be waiting for some of its neighbors to complete some task. A node that is waiting for a neighbor cannot proceed with its task until this neighbor completes its own task. A snapshot of such a system can be described by a wait-for graph: the wait-for graph is a directed graph \((V, E)\), where \(V\) is the set of the network nodes, and \(E\) contains an edge \(i \rightarrow j\) iff node \(i\) was waiting for node \(j\) when the snapshot took place.

Given a snapshot of the network it is possible to decide whether there are deadlocked nodes by examining the wait-for graph: if the wait-for graph contains a directed cycle, all nodes in the cycle are deadlocked, and so are all the nodes that are waiting for the nodes in the cycle.

Thus, the only information that should be recorded by the distributed snapshot protocol is the topology of the wait-for graph. Such a distributed snapshot protocol is a fixed-overhead protocol.
2 Memory Overhead of Distributed Snapshots

The memory overhead of an instantaneous (non-distributed) snapshot is the following:

1. the stage of collecting the local information at each node: $S$ for recording the local state; $c$ for recording the state of each adjacent link, where $c$ is the total size of the link buffers. Thus, each node needs $O(S + d \times c)$ memory for the first stage.

2. the stage of collecting global information: $S$ for storing the memory image of each node, and $2c$ for storing the state of the two directions of each link. Thus, each node needs $O(S \times |V| + 2c \times |E|)$ memory for the second stage.

On the other hand, as shown in the following example, the memory overhead of distributed snapshots is unbounded.

Consider a network with three nodes: $i$—$j$—$k$, a link connecting nodes $i$ and $j$ and a link connecting nodes $j$ and $k$, and the following protocol: nodes $j$ and $k$ exchange a token indefinitely. Whenever node $k$ receives the token, it sends it to node $j$. Each time node $j$ receives the token, it returns it back to node $k$, and sends a RCVD_TOKEN message, containing some information, to node $i$.

Let node $i$ initiate a distributed snapshot. Node $i$ does it by first recording its local state and then sending a MARKER message to node $j$. Assume that the link leading from node $i$ to node $j$ is temporarily very slow; therefore, the MARKER message is delayed for a long time. Meanwhile, nodes $j$ and $k$ may exchange the token any number of times. Thus, node $j$ may send any number of RCVD_TOKEN messages to node $i$ before receiving the MARKER message. All these RCVD_TOKEN messages must be recorded by $i$ as being in the link from $j$ to $i$. Therefore, node $i$ has to store the contents of all these RCVD_TOKEN messages in its local memory. The amount of memory needed for this task is unbounded.

We conclude that the memory overhead of distributed snapshots is $\infty$. In the following sections we discuss some solutions to this problem.

3 Theory versus Reality

In real system configurations, there usually exists some bound on the delay of the links. If in the network $i$—$j$—$k$ the link delays are bounded, the number of RCVD_TOKEN messages received by node $i$, after sending the MARKER message to $j$ and before receiving a MARKER message from $j$, is bounded. However, in many cases this bound is unbeareably large. This is the case, for example, if the links in the network have a maximal end-to-end delay that is much larger than the time it takes to send one message. Other cases are when the transmission rate over the links in the network is different in each direction, or when one message may be corrupted and retransmitted many times before received correctly, etc. Furthermore, the bound on the transmission delay is usually difficult to measure, and may also change from time to time, making it difficult to calculate the amount of
protocol that brings this information only to the initiators of the snapshot protocol, yielding better communication complexity.

In the rest of this paper we proceed as follows: in Sec. 2 we show that the memory overhead of distributed snapshots is unbounded. Some solutions to this problem are presented in the following sections. Sec. 4 presents fixed-overhead special-purpose versions of distributed snapshots for termination and deadlock detection. In Sec. 5 we show that in some cases, the problem can be solved by using other known protocols instead of using distributed snapshots. Sec. 6 presents synchronization techniques for limiting the photographed protocol to executions whose distributed snapshot requires only bounded memory.
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.

In [ShS 4] we also define fixed-overhead superimposed protocols as superimposed protocols whose memory overhead 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. [ShS 4] discusses the memory overhead of various synchronization techniques and presents lower bounds on the memory overhead of synchronizers as a function of their time complexity.

The present paper discusses the memory overhead of distributed snapshots. The technique of distributed snapshots is introduced in [CL 1]. The idea is to construct a snapshot of the local states of the processors and of the links, working with a given asynchronous protocol, in order to create a consistent image of the global state.

The distributed snapshot protocol, proposed in [CL 1], is as follows: a node that decides to initiate the snapshot, records its local state and sends a MARKER message along each of its adjacent links. Each node records its local state immediately upon receiving the first MARKER message. At that time, it also sends a MARKER message along each of its adjacent links.

In the sequel, we use $T_i$ to denote the time when node $i$ records its local state and sends MARKER messages. We use $T_i(j)$ to denote the time when node $i$ receives a MARKER message sent by node $j$.

In addition to the local state, each node $i$ also records the contents of each of its incoming links. This is done by recording, for each link $(i, j)$, all messages received during the time interval $(T_i, T_i(j))$. A node that performs $T_i$ when receiving a MARKER message from one of its adjacent links, records that link to be empty.

In [CL 1], the authors suggest that after recording the local state and the state of each adjacent link, each node broadcasts this knowledge to all other nodes in the network. [SK 1] presents a
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.

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.

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.

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 [KK 1], [PU 2], [ABL 1], [AKP 1], [ITT 1], and [BTY 1].

In the present paper, we deal with a special type of distributed protocols, referred to as superimposed protocols, namely protocols that have a meaning only when they are combined with other protocols. Examples of superimposed protocols are: termination detection [Fran 1], [DS 1], [Rana 1], [Mis 1], [CM 1], [Erik 1], deadlock detection and resolution [CM 2], [CMH 1], [BT 1], [AM 1], [RBC 1], distributed snapshots [CL 1], [SK 1], synchronizers [Awer 1], [PU 1], [LT 1], [CCGZ 1], [KTZ 1], [ER 1], [AS 1], [ShS 1], [ShS 2], [ShS 3], [ShS 4], distributed schedulers [MR 1], and reset procedures for dynamic networks [Finn 1], [Seg 1], [SH 1], [AAG 1], [Awer 2].

When discussing memory complexity of superimposed 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 [ShS 4] 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 DISTRIBUTED SNAPSHOTS

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

February 7, 1994

Abstract

This paper shows that the memory overhead of distributed snapshots is unbounded. Several techniques are suggested for bounding it: bounded memory-overhead versions of distributed snapshots for specific problems, like termination detection and deadlock detection; use of alternative protocols; or use of synchronizers or schedulers in order to limit the photographed protocol to executions whose distributed snapshot requires bounded memory. Each solution is discussed in detail, and its memory overhead is analyzed.