Network synchronization with polylogarithmic overhead

Baruch Awerbuch * David Peleg †

Abstract

The synchronizer is a simulation methodology for simulating a synchronous network by an asynchronous one, thus enabling the execution of a synchronous algorithm on an asynchronous network. Previously known synchronizers require each processor in the entire network $G(V, E)$ to participate in each pulse of the synchronization process. As a result, the communication overhead of existing synchronizers depends linearly on the number n of the network nodes.

This paper presents a novel type of synchronizer, whose overhead is only poly-logarithmically dependent on n. This synchronizer can also be realized with $polylog(n)$ space. This polylog-overhead synchronizer is based on involving only the "relevant" portions of the network in the synchronization process.

1 Introduction

1.1 Motivation

The *synchronizer* is a simulation methodology introduced in [Awe85a] for simulating a synchronous network by an asynchronous one, thus enabling the execution of a synchronous algorithm on an asynchronous network. The availability of such a tool enables us to design distributed network algorithms intended to be run in an asynchronous environment under the assumption of a synchronous environment. This is desirable since it is generally easier to comprehend and reason about the behavior of distributed protocols in the synchronous model, hence this model is more convenient for designing and analyzing network algorithms. Synchronizers, and various aspects or applications of them, were studied in [Jaf80, Awe85b, CGZ86, SM86, FLS87, LT87, AS88, PU89, ER90].

The synchronizer operates by generating a sequence of local clock pulses at each processor of the network, satisfying the following property:

> *pulse number p is generated by a processor only after it received all the messages of the algorithm sent to it by its neighbors during their pulse number $p - 1$.*

Intuitively, the problem lies in the fact that in case processor v did not send any message to its neighbor u at a certain pulse, u cannot deduce this by simply waiting for a fixed period of time, as link delays in the asynchronous network are unpredictable.

1.2 The model

We consider the standard model of an asynchronous point-to-point communication network (e.g. [GHS83]). The network is described by an undirected graph $G = (V, E)$, $|V| = n$. The nodes of the graph represent the processors of the network and the edges represent bidirectional communication channels between the processors. We occasionally use V (respectively, E) also to denote the *number* of vertices (resp., edges). The diameter of the network is denoted by $D(G)$, and we denote $\delta = \lceil \log_2 D(G) \rceil$.

All the processors have distinct identities. There is no common memory, and algorithms are event-driven (i.e., processors cannot access a global clock in order to decide on their action). Messages sent from a processor to its neighbor arrive within some finite but unpredictable time. Each message contains a fixed number of bits, and therefore carries only a bounded amount of information.

A synchronous network is a variation of the above

*Department of Mathematics and Lab. for Computer Science, M.I.T., Cambridge, MA 02139; ARPANET: baruch@theory.lcs.mit.edu. Supported by Air Force Contract TNDGAFOSR-86-0078, ARO contract DAAL03-86-K-0171, NSF contract CCR8611442, and a special grant from IBM.

†Department of Applied Mathematics, The Weizmann Institute, Rehovot 76100, Israel; BITNET: peleg@wisdom. Supported in part by an Allon Fellowship, by a Walter and Elise Haas Career Development Award and by a Bantrell Fellowship.

model in which all link delays are bounded. More precisely, each processor keeps a local clock, whose pulses must satisfy the following property. A message sent from a processor v to its neighbor u at pulse p of v must arrive at u before pulse $p+1$ is generated by u.

1.3 Complexity measures

Our complexity measures are defined as follows. The *communication complexity* of an algorithm π, $\mathcal{C}(\pi)$, is the worst-case total number of messages sent during the run of the algorithm. The *time complexity* of an algorithm π, $\mathcal{T}(\pi)$, is defined as follows. For a synchronous algorithm, $\mathcal{T}(\pi)$ is the number of pulses generated during the run. For an asynchronous algorithm, $\mathcal{T}(\pi)$ is the worst-case number of time units from the start of the run to its completion, assuming that each message incurs a delay of at most one time unit. This assumption is used only for performance evaluation, and does not imply that there is a bound on delay in asynchronous networks.

On an abstract level, the synchronizer $\mathcal{S}$ transforms a synchronous protocol π into equivalent asynchronous protocol $\phi(\mathcal{S},\pi)$. The complexity of the synchronizer $\mathcal{S}$ is measured by the increase in the communication and time complexity of the transformation, namely by the following two "stretch factors":

$$\begin{aligned} Stretch_{com}(\mathcal{S}) &= \max_{\pi} \frac{\mathcal{C}(\phi(\pi,\mathcal{S}))}{\mathcal{C}(\pi)} \\ Stretch_{time}(\mathcal{S}) &= \max_{\pi} \frac{\mathcal{T}(\phi(\pi,\mathcal{S}))}{\mathcal{T}(\pi)} \end{aligned}$$

We refer to $Stretch_{com}(\mathcal{S})$ and $Stretch_{time}(\mathcal{S})$ as the *communication stretch* and the *time stretch* of the synchronizer $\mathcal{S}$, respectively.

Observe that these definitions do not take the complexity of initializing the synchronizer into account. The issue of initialization is ignored throughout the remainder of this abstract. Let us comment, though, that this initialization can be performed using the distributed coarsening algorithms of [AP90].

1.4 Previous work

Several synchronizers have been suggested in the literature [Awe85a, PU89]. All of these synchronizers share two characteristic properties in common. First, they are *memoryless*, in the sense that they process each pulse separately, and do not attempt to capitalize on information about the past. Secondly, they are *local*, in the sense that the communication is performed only between neighbors, or nodes that are at logarithmic distance from each other. Essentially, existing synchronizers require each processor to verify that all its neighbors are done with the current pulse before generating the next pulse. This verification phase is thus responsible for the $\Omega(n)$ additional communication overhead per each pulse of the protocol. We refer to this class of synchronizers as *oblivious* synchronizers. It has been proved in [Awe85a] that synchronizers of this class inherently require each processor in the *entire* network to participate in the collective synchronization effort in *each pulse* of the process. This implies that the communication stretch increases linearly with the size n of the network. Moreover, [Awe85a] presents an oblivious synchronizer γ that is communication-optimal for general networks, and whose complexities are:

$$\begin{aligned} Stretch_{com}(\gamma) &= \Theta(n) \\ Stretch_{time}(\gamma) &= O(\log n) \end{aligned}$$

As pointed out in [Awe85b], in many important network algorithms, such as Breadth First Search (BFS), a node participates in very few pulses of the synchronous protocol. In fact, in various protocols, some nodes never participate at all. Since the overhead of existing synchronizers depends linearly on the size of the network, compiling synchronous algorithms like BFS, with linear ($O(|E|)$) complexity, on such synchronizers, results in asynchronous algorithms with quadratic (($O(n^2)$) complexity. In order to reduce this complexity (e.g. aiming towards $O(E+n\cdot polylog(n))$ BFS), it is desirable to detect termination of the computation, and stop "bothering" nodes after they quit the computation.

1.5 Contributions of this work

This paper takes a different approach to the problem. We abandon the method of simulating each pulse separately, which is doomed to $\Omega(n)$ multiplicative overhead. Rather, we do our simulation "incrementally", taking advantage of the information of the past, and involving only nodes that actually take part in the computation. That is, instead of minimizing the worst-case overhead per pulse, we try to minimize the cumulative overhead over the whole execution, or *amortized* overhead per pulse.

Our major objective is to achieve a "smooth" operation of the simulator, by preventing a node from "prematurely" performing operations of a certain pulse, namely without receiving all the relevant information needed to perform this pulse. However, we want to accomplish this without communicating between each pair of neighbors in between any two successive pulses.

Unfortunately, if only part of the network nodes are involved in the synchronization, then necessarily some branches of the computation are proceeding faster than others. The computation will inevitably reach an inconsistent state, unless different branches are sufficiently separated from each other. One of the contributions of the paper is a characterization of "inconsistent" states of the network, based on the position of each node in the computation, and the distances between those nodes. To avoid inconsistencies, we must "slow down" fast branches of the computation if slow branches are present in the vicinity.

For example, suppose node v believes that its part in pulse p is completed. This belief is erroneous if v has a neighbor u that has not yet communicated to v the outcome of its computation at pulse $p-1$. More generally, consider a path $P = w \rightsquigarrow v$, of length d, carrying information flow that is started by w at pulse $p-d$. This information affects the execution of pulse p at v, and thus v cannot start pulse $p+1$ before this information arrives.

The crux of the algorithm is verifying, for each single pulse p, that the pulse p is "performed" (in the sense that local registers are changed, clock are advanced, and messages are sent) only when no relevant information is in transit, i.e. "errors" never occur. Towards this goal, each network node should maintain a (partial) picture of the global state, or the "snapshot" [CL85] of the network. While it is perfectly clear that, given such "snapshots", the problem is easy, it appears unlikely that maintaining global snapshots can be cheaper than local communication among neighbors, which, as we already know, requires linear overhead.

The contribution of this paper is a novel data structure, which accomplishes this task of efficient maintenance of the global network snapshots. Throughout the whole execution, the total overhead in communication, time, and space is only polylogarithmically dependent on the size of the network. This enables to develop a synchronizer $\mathcal{S}$ such that

$$\begin{aligned} Stretch_{com}(\mathcal{S}) &= O(\log^3 n), \\ Stretch_{time}(\mathcal{S}) &= O(\log^3 n). \end{aligned}$$

These complexities are at most a $polylog(n)$ factor away from the optimum.

The class of algorithms for which our synchronization methodology applies is restricted by the following two requirements. First, we require that the algorithms handled are event-driven. That is, our solution excludes the class of algorithms allowing the use of explicit "time-outs." (Such algorithms can be implemented by replacing a timeout for t time units by a self-loop of messages for that number of times.) Secondly, it is required that in the beginning of the execution, the originators are known to all the nodes of the network. (In other words, if the originators are "spontaneous", they are required to first "register" as originators throughout the network.)

2 Basic concepts

2.1 A generalized concept of safety

All oblivious synchronizers are based on requiring every processor receiving a message from a neighbor to send back an acknowledgment. This enables every processor to learn, within finite time, that all the messages it sent during a particular pulse have arrived. Such a processor is said to be *safe* with respect to that pulse. Note that the acknowledgments do not increase the message complexity or the time complexity of the algorithm by more than a constant factor.

Definition 2.1 The processor v is said to be *safe* with respect to pulse p, denoted $Safe(v,p)$, if either it does not execute this pulse, or all the messages it sent during pulse p have arrived.

The following trivial fact will be used later without reference.

Fact 2.2 If v satisfies $Safe(v,p)$ then also $Safe(v,q)$ for every $q < p$.

We characterize a processor's readiness to increase its pulse in terms of its neighbors' safety. In order to start next pulse, each processor is required to verify that all its neighbors are done with respect to the current pulse.

Definition 2.3 The processor w is *ready* for pulse p (i.e., it is ready to increase its counter to pulse p), denoted $Ready(w,p)$, if all the messages sent to w from its neighbors at pulse $p-1$ have already arrived.

Fact 2.4 A processor w satisfies $Ready(w,p)$ if each of its neighbors v satisfies $Safe(v,p-1)$.

The second and main phase of all previous synchronizers involves delivering this "safety" information. The main problem with these synchronizers is that safety must be detected separately for each node at each pulse, namely, the synchronizers are *memoryless*.

Our work is based on a generalization of the concept of safety, called *generalized readiness*, which characterizes the ability of the node to perform the operations required in pulse p based on the global state of the network.

Definition 2.5 A processor v satisfies $Gen\_Ready(v,p)$ if, for all $d>0$, each node $u \in \mathcal{N}_d(v)$ satisfies $Safe(u,p-d)$.

The algorithm is based on the following observation.

Fact 2.6 If v satisfies $Gen\_Ready(v,p)$, then it satisfies $Ready(v,p)$.

Consequently, in order to make the decision to perform the operations of pulse p it suffices to check condition $Safe(u,p-d)$ for each node $u \in \mathcal{N}_d(v)$, for every d.

Now, checking explicitly, in each pulse, that condition $Safe(u,p-d)$ is met for each node $u \in \mathcal{N}_d(v)$ boils down to computing the global state, or snapshot [CL85] of the computation. Direct implementation of this is too expensive. Instead, the algorithm checks a variation on this condition, which can efficiently implemented with the help of the data structure described in the sequel.

2.2 Basic data structures

We say that a node v is *active* in pulse p of the protocol if it sends a message in that pulse. For simplicity of presentation, we assume that each network node is active in at most a single pulse of the protocol. In section 8 we describe how to modify the solution for the general case, after removing this assumption.

Let $pulse(v)$ be the pulse at which v is active; for nodes that are never active, $pulse(v)=\infty$. Observe that originators of the protocol have $pulse(v)=0$. Also, node v is assumed to receive messages only in pulse $pulse(v)-1$.

In our simulation, for each node v with $pulse(v)=p$, there exists a node u such that a message sent from u at pulse $p-1$ is the first message arriving at v. In this case v remembers the sender u of the message as its *parent*, and sets $parent(v)=u$. (For originators v, $parent(v)=nil$). The *parent* relation defines a directed subgraph called the *execution forest*, consisting of trees rooted at the originators. Let *child* be the inverse relation, i.e., $u \in child(v)$ iff $v=parent(u)$. Observe that when node u wakes up it knows $parent(u)$, and it is straightforward to ensure that the parent v knows that u enters $child(v)$ at this pulse.

For technical reasons, it is convenient to imagine that for every originator v of the protocol there exists an infinite "prehistoric execution path" going through nodes v_{-i} and leading to the node v_0, with $parent(v_{-i})=v_{-(i+1)}$. In the particular implementation that we use, all of these imaginary nodes v_{-i} actually reside in the node v itself, and their tasks in the synchronization are carried out by v. In our accounting, we will assume that communication from v_0 to v_{-i} costs $O(i)$ communication and time, which is clearly an overcount, but simplifies our calculations.

A second data structure that is maintained during the execution is the *destination forest* (c.f. [AAPS87]). Throughout the paper we associate with every pulse number p a *level* number $\ell(p)$, defined as follows. For $p=0$, $\ell(p)=\infty$. For $p \neq 0$, let $i \geq 0$, $j \geq 1$ be integers such that $|p|=(2j-1)\cdot 2^i$. Then $\ell(p)=\min\{i,\delta\}$. Hence there are $\delta+1=\lceil \log_2 D(G)\rceil + 1$ possible levels. We define now

$$prev(p) = \max\Big\{\tilde{p} \mid \ell(\tilde{p}) = \min\{\ell(p)+1,\delta\} \bigwedge \tilde{p} \leq p - 2^{\ell(p)}\Big\}.$$

That is, $prev(p)$ as the highest value of $\tilde{p}$, such that $\ell(\tilde{p})=\ell(p)+1$ (but no larger than δ) and $\tilde{p}$ is "far" enough from p.

For a particular node v at layer p in the tree, we define $host(v)$ to be the ancestor of v on layer $prev(p)$ of the execution tree. (Note that $prev(p)=\tilde{p}$ may be negative, in which case $host(v)$ would be the imaginary node $w_{\tilde{p}}$ residing at the originator w who is v's ancestor at pulse 0.) We also define $root(v)=host(host(v))$. Intuitively, $host(v)$ and $root(v)$ are "responsible" for the synchronization of node v. We may sometimes denote vertices by h or r for emphasizing their roles as hosts or roots, respectively.

We use the following notation.

Notation 2.7

$$\begin{array}{rcl}
\texttt{Nodes}(p) &=& \{v \mid pulse(v)=p\}\\
\texttt{Trunc}(p) &=& \{w \mid w \text{ has no descendents}\\
&& \quad v \in \texttt{Nodes}(p)\}\\
\texttt{Hosts}(p) &=& \{h \mid \exists v\ , pulse(v)=p,\ h=host(v)\}\\
\texttt{Hosts}_i &=& \{h \mid \exists p,\ \ell(p)=i,\ h \in \texttt{Hosts}(p)\}\\
\texttt{Roots}(p) &=& \texttt{Hosts}(prev(p))\\
\mathcal{N}_d(v) &=& \{u \mid dist(v,u) \leq d\}\\
\texttt{Near_Nodes}(v) &=& \texttt{Nodes}(p) \cap \mathcal{N}_{2^{\ell(p)+3}}(v),\\
&& \quad \text{for } p = pulse(v)\\
\texttt{Near_Hosts}(h) &=& \{h' \mid \exists v,v',\ h=host(v), h'=host(v'),\\
&& \quad v' \in \texttt{Near_Nodes}(v)\}\\
\texttt{Near_Roots}(h) &=& \texttt{Near_Hosts}(host(h))
\end{array}$$

2.3 Properties of data structures

The following observation is adopted from [AAPS87].

Fact 2.8 If $pulse(v) = p$ and $i = \ell(p)$, then

$$\begin{aligned} dist(v, host(v)) &\leq 3 \cdot 2^i \\ |\texttt{Hosts}_i| &\leq \frac{\mathcal{C}(\pi)}{2^i} \end{aligned}$$

Claim 2.9 Let $v \in \texttt{Nodes}(p)$, $v' \in \texttt{Near\_Nodes}(v)$, $h = host(v)$, $h' = host(v')$. Then

1. $dist(v, h') \leq 2^{\ell(p)+4}$
2. $dist(h, h') \leq 2^{\ell(p)+4}$

Corollary 2.10 For all $h \in \texttt{Hosts}(p)$ and $r' \in \texttt{Near\_Roots}(h)$, $dist(h, r') \leq 2^{\ell(p)+5}$.

Corollary 2.11 For all nodes $h, h' \in \texttt{Hosts}(p)$, $r = host(h)$ and $r' = host(h')$, if $h' \in \texttt{Near\_Hosts}(h)$ then

1. $h' \in \texttt{Near\_Nodes}(h)$,
2. $r' \in \texttt{Near\_Hosts}(r)$, and
3. $r' \in \texttt{Near\_Roots}(h)$.

3 High-level algorithm

3.1 Outline

A processor v that receives a message carrying pulse number p considers itself "awake" at pulse $p+1$; this condition is denoted as $Wake(v, p+1)$. It would be premature for a processor that becomes "awake" at pulse $p+1$ to perform actions of pulse $p+1$ right away, as other messages of pulse p may be in transit to that node, and operations of pulse $p+1$ may depend on the contents of those messages. The nodes proceed to execute the algorithm below in order to find out the "right time" to execute a given pulse $p+1$.

As discussed in the previous section, collecting "safety" information can be an expensive task. The way to reduce costs is by organizing the nodes into logical groups and handling groups collectively, using centralized registration centers. For this purpose, we break the execution tree into subtrees according to pulse levels, and appoint the hosts (namely, the roots of the subtrees) to coordinate safety checking and registration for their respective subtree nodes.

Conceptually, it is convenient to think of the "safety tracking" process as done by an abstract registration mechanism, termed $\mathcal{REGISTER}$ in the sequel. Hosts are responsible for registering in this mechanism. Consequently, checking whether the pulse number can be increased safely requires verifying that various "neighboring" hosts are already registered as safe too.

The entire process is thus based on registration in the $\mathcal{REGISTER}$ mechanism and checking for various conditions concerning the registration status of various nodes. Let us now specify the particular conditions used throughout our algorithm.

- **Condition** $Tree\_Trunc(v, p)$**:**
  v has no descendants $v' \in \texttt{Nodes}(p)$.
- **Condition** $Tree\_Safe(v, p)$**:**
  v has descendants $v' \in \texttt{Nodes}(p)$, and $Safe(v', p)$ holds for all of them.
- **Condition** $Marked\_Safe(h, p)$**:**
  either $h \in \texttt{Hosts}(p)$ and h is *registered* as satisfying $Tree\_Safe(h, p)$,
  or $h \in \texttt{Nodes}(prev(p)) \cap \texttt{Trunc}(p)$ and h satisfies $Tree\_Trunc(h, p)$.
- **Condition** $Tree\_Clean(v, p)$**:**
  $Marked\_Safe(h, p)$ holds for all descendants $h \in \texttt{Nodes}(prev(p))$ of v.
- **Condition** $Marked\_Clean(r, p)$**:**
  the root r is registered as satisfying $Tree\_Clean(r, p)$.

The algorithm itself is described in the next subsection.

3.2 Detailed description

The algorithm proceeds according to the following steps.

Whenever a node receives a message of the algorithm it sends an acknowledgment back. Whenever a node $v \in \texttt{Nodes}(p)$ receives all acks for its messages, it detects condition $Safe(v, p)$. This also implies that all $v' \in child(v)$ satisfy $Wake(v', p+1)$.

A node $v \in \texttt{Nodes}(p-1)$ that has no children in the execution tree reports itself as satisfying $Tree\_Trunc(v, p)$. A node $v \in \texttt{Nodes}(p-1)$ that has children v' in the execution tree waits until all of them report satisfying $Safe(v', p)$ and then reports itself as satisfying $Tree\_Safe(v, p)$. These reports are propagated upwards in the execution tree using the following "convergecast" process. A node v in the execution tree waits until it gets reports from all its children v'. If all the reports are $Tree\_Trunc(v', p)$ then v forwards a report $Tree\_Trunc(v, p)$ to its parent. If, on the other hand, at least one of the reports is $Tree\_Safe(v', p)$, then v forwards a report $Tree\_Safe(v, p)$ to its parent. Thus at each pulse, each node of the execution tree knows which of its children are still "active" (i.e., have "leaves" on layer p) and which are terminated. The goal of this "convergecast" process is to collect this information at the nodes $v \in \texttt{Nodes}(prev(p))$ (i.e., to inform it that $Tree\_Safe(v, p)$ or $Tree\_Trunc(v, p)$).

Whenever a host $h \in \texttt{Hosts}(p)$ detects that it satisfies condition *Tree_Safe*(h,p), h invokes procedure MARK_SAFE$^h(p)$. The task of this procedure is to register h in the $\mathcal{REGISTER}$ mechanism as satisfying *Tree_Safe*(h,p). Thus termination of this procedure guarantees condition *Marked_Safe*(h,p). Note that a node $v \in \texttt{Nodes}(prev(p))$ that is not in $\texttt{Hosts}(p)$, i.e., which satisfies *Tree_Trunc*(v,p), satisfies *Marked_Safe*(v,p) automatically, and does not need to invoke procedure MARK_SAFE and register itself in the $\mathcal{REGISTER}$ mechanism.

Using a "convergecast" process similar to that of the first stage, each root node $r \in \texttt{Roots}(p)$ can detect that all its descendants $h \in \texttt{Nodes}(prev(p))$ satisfy *Marked_Safe*(h,p), (i.e., either they are in $\texttt{Hosts}(p)$ and registered, or they are in $\texttt{Trunc}(p)$), and thus, that r itself satisfies condition *Tree_Clean*(r,p).

Whenever $r \in \texttt{Roots}(p)$ detects *Tree_Clean*(r,p), r invokes procedure MARK_CLEAN$^r(p)$. In this procedure, r registers in the $\mathcal{REGISTER}$ mechanism as satisfying *Tree_Safe*(r,p). Thus termination of this procedure guarantees condition *Marked_Clean*(r,p).

Whenever $h \in \texttt{Hosts}(p)$ detects that $r_h = host(v)$ satisfies *Marked_Clean*(r_h,p), h executes procedure SCAN$^h(p)$. The role of this procedure is to wait until it can verify the following condition:

- **Condition** *Scan_End*(h,p)**:**
  condition *Tree_Clean*(r,p) holds for every root $r \in \texttt{Near\_Roots}(h)$ that satisfied *Marked_Safe*$(r, prev(p))$ upon invocation of the procedure.

For $v \in \texttt{Nodes}(p+1)$, once its host $h = host(v)$ satisfies condition *Scan_End*(h,p), v can start procedure *Raise*(v,p) and perform operations of pulse $p+1$.

3.3 Formal specification

Formal specification of the procedures used in our algorithm is given below.

"*Safe*" convergecast

Invoked at: $h \in \texttt{Hosts}(p)$ and their descendants, $v \in \texttt{Nodes}(p)$ such that $h = host(v)$

Action: collect *Safe*(v,p) information up the execution tree.

Postcondition: *Tree_Safe*(h,p)

Procedure MARK_SAFE$^h(p)$

Invoked at: $h \in \texttt{Hosts}(p)$

Precondition: *Tree_Safe*(h,p)

Action: register as satisfying *Tree_Safe*(h,p) in the $\mathcal{REGISTER}$ mechanism.

Postcondition: *Marked_Safe*(h,p)

"*Clean*" convergecast

Invoked at: $r \in \texttt{Roots}(p)$ and their descendants v

Action: collect *Tree_Clean*(v,p) information up the execution tree.

Postcondition: *Tree_Clean*(r,p)

Procedure MARK_CLEAN$^r(p)$

Invoked at: $r \in \texttt{Roots}(p)$

Precondition: *Tree_Clean*(r,p)

Action: register as satisfying *Tree_Clean*(r,p) in the $\mathcal{REGISTER}$ mechanism.

Postcondition: *Marked_Clean*(r,p)

Procedure SCAN$^h(p)$

Invoked at: $h \in \texttt{Hosts}(p)$

Precondition: *Marked_Clean*$(host(h),p)$

Action: await postcondition.

Postcondition: *Scan_End*(h,p)

Procedure *Raise*(v,p)

Invoked at: $v \in \texttt{Nodes}(p+1)$

Precondition: *Scan_End*$(host(v),p)$

Action: Execute operation of pulse $p+1$

The only questions that remain to be handled are finding an efficient implementation of the above *Tree_Safe* and *Tree_Clean* conditions, and mostly, finding a way to enable a host $h \in \texttt{Hosts}(p)$ to detect the fact that all $r \in \texttt{Near\_Roots}(h)$ satisfy condition *Tree_Clean*(r,p). This issue is addressed in Section 5.

4 Correctness of the algorithm

In this section we prove that the high-level algorithm presented in the previous section satisfies the correctness requirements on a synchronizer. The proof is centered around the following crucial lemma.

Lemma 4.1 Consider pulses p and $\tilde{p} = prev(p)$, and consider the nodes $h \in \texttt{Hosts}(p)$ and $r = host(h) \in \texttt{Roots}(p)$.

1. Upon *invocation* of $\textsc{Scan}^h(p)$, condition $Marked\_Safe(r', \tilde{p})$ holds $\forall\ r' \in \texttt{Near\_Roots}(h)$.

2. Upon *termination* of $\textsc{Scan}^h(p)$, condition $Tree\_Clean(r', p)$ holds $\forall\ r' \in \texttt{Near\_Roots}(h)$.

3. Upon *termination* of $\textsc{Scan}^h(p)$, condition $Marked\_Safe(h', p)$ holds $\forall\ h' \in \texttt{Near\_Hosts}(h)$.

Claim 4.2 Consider nodes $v \in \texttt{Nodes}(p)$ and $h = host(v)$. For all nodes $v' \in \texttt{Near\_Nodes}(v)$, $Safe(v', p)$ holds upon *termination* of $\textsc{Scan}^h(p)$.

Corollary 4.3 A node x enters pulse $p+1$ only after $Safe(w, p)$ holds for all neighbors w of x such that $w \in \texttt{Nodes}(p)$.

5 Technical implementation

5.1 Regional matchings

The basic components of our construction are a read set $\texttt{Read}(v) \subseteq V$ and a write set $\texttt{Write}(v) \subseteq V$, defined for every vertex v. Consider the collection $\mathcal{RW}_m$ of all pairs of sets, namely
$\mathcal{RW}_m = \{\ \texttt{Read}_m(v),\ \texttt{Write}_m(v) \mid v \in V\ \}$.

Definition 5.1 The collection $\mathcal{RW}_m$ is an *m-regional matching* (for some integer $m \geq 1$) if for all $v, u \in V$ such that $dist(u, v) \leq m$, $\texttt{Write}_m(v) \bigcap \texttt{Read}_m(u) \neq \emptyset$.

For any m-regional matching $\mathcal{RW}_m$ define the following parameters:
$Deg_{read}(\mathcal{RW}_m) = \max_{v \in V} |\texttt{Read}_m(v)|$,
$Rad_{read}(\mathcal{RW}_m) = \frac{1}{m} \cdot \max_{v \in V}\{dist(v, \texttt{Read}_m(v)\}$,
and $Deg_{write}(\mathcal{RW}_m)$, $Rad_{write}(\mathcal{RW}_m)$ are defined analogously based on the sets $\texttt{Write}_m(v)$.

We shall make use of the following lemma of [AP].

Lemma 5.2 [AP] For all $m \geq 1$, there exists an m-regional matching $\mathcal{RW}_m$ with

$$\begin{aligned} Deg_{read}(\mathcal{RW}_m) &\leq 4 \log n \\ Rad_{read}(\mathcal{RW}_m) &\leq 2 \log n + 1 \\ Deg_{write}(\mathcal{RW}_m) &= 1 \\ Rad_{write}(\mathcal{RW}_m) &\leq 2 \cdot \log n + 1 \end{aligned}$$

For every level $0 \leq i \leq \delta$, we construct a 2^i-regional matching $\mathcal{RW}_{2^i}$ as in Lemma 5.2.

Definition 5.3 For any node v, with $\ell(pulse(v)) = i$, we define

$$\begin{aligned} \texttt{Write}(v) &= \texttt{Write}_{2^\alpha}(v), \\ \texttt{Read}(v) &= \texttt{Read}_{2^\beta}(v), \end{aligned}$$

where $\alpha = \min\{i+3, \delta\}$ and $\beta = \min\{i+4, \delta\}$.

5.2 Details of the implementation

We now describe an implementation of the registration mechanism $\mathcal{REGISTER}$ and the procedures $\textsc{Mark\_Safe}$, $\textsc{Mark\_Clean}$ and $\textsc{Scan}$, using regional matchings. In the implementation, every node w in every set $\texttt{Write}(v)$ of every node v maintains lists $\texttt{Safe\_List}^w(p)$ (resp., $\texttt{Clean\_List}^w(p)$) for every pulse p. These lists keep track of nodes marked as satisfying *Marked_Safe* and *Marked_Clean* with respect to the appropriate pulses. The procedures are implemented as follows.

1. Procedure $\textsc{Mark\_Safe}^h(p)$:
   The procedure registers a host $h \in \texttt{Hosts}(p)$ as satisfying condition $Tree\_Safe(h, p)$ by listing it in all the lists $\texttt{Safe\_List}^w(p)$ at all nodes $w \in \texttt{Write}(h)$. Thus the *actual* condition guaranteed upon termination of this procedure is:

   - **Actual-Postcondition** $Marked\_Safe(h, p)$**:** $h \in \texttt{Safe\_List}^w(p)$ at all $w \in \texttt{Write}(h)$.

2. Procedure $\textsc{Mark\_Clean}^r(p)$:
   The procedure registers a root $r \in \texttt{Roots}(p)$ as satisfying condition $Tree\_Clean(r, p)$ by listing it in all the lists $\texttt{Clean\_List}^w(p)$ at all nodes $w \in \texttt{Write}(r)$. Thus the actual condition guaranteed upon termination of this procedure is:

   - **Actual-Postcondition** $Marked\_Clean(r, p)$**:** $r \in \texttt{Clean\_List}^w(p)$ for all $w \in \texttt{Write}(r)$.

3. Procedure $\textsc{Scan}^h(p)$:
   The procedure $\textsc{Scan}^h(p)$ is invoked by a host $h \in \texttt{Hosts}(p)$ whenever it detects that its host, $host(h)$, satisfies $Marked\_Clean(host(h), p)$. The procedure $\textsc{Scan}^h(p)$, in turn, calls sub-procedures $\textsc{Verify}^w(h, p)$ for all $w \in \texttt{Read}(h)$. Sub-procedure $\textsc{Verify}^w(h, p)$ terminates once the lists $\texttt{Safe\_List}^w(prev(p))$ and $\texttt{Clean\_List}^w(p)$ at w satisfy $\texttt{Safe\_List}^w(prev(p)) \subseteq \texttt{Clean\_List}^w(p)$. Once all sub-procedures $\textsc{Verify}^w(h, p)$ terminate, $\textsc{Scan}^h(p)$ is completed. Thus when $\textsc{Scan}^h(p)$ terminates, the actual condition provided directly by the procedure is

   - **Actual-Postcondition** $Actual\_Scan\_End(h, p)$**:**
     $\forall w \in \texttt{Read}(h)$,
     $\texttt{Safe\_List}^w(prev(p)) \subseteq \texttt{Clean\_List}^w(p)$.

   However, the postcondition required by the high-level specification of this procedure is somewhat different, and requires:

- **Required-Postcondition** $Scan\_End(h,p)$**:** condition $Tree\_Clean(r,p)$ holds for every root $r \in \texttt{Near\_Roots}(h)$ that satisfied $r \in \texttt{Safe\_List}^w(prev(p))$ at all $w \in \texttt{Write}(r)$ upon invocation of the procedure.

We shall later show that the actual postcondition implies the required one.

5.3 Formal implementation specification

Formal specification of the procedures in our implementation is given below.

Procedure $\text{MARK\_SAFE}^h(p)$

Invoked at: $h \in \texttt{Hosts}(p)$

Precondition: $Tree\_Safe(h,p)$
/* $\equiv \ \forall v \in \texttt{Nodes}(p),\ h = host(v) \Rightarrow Safe(v,p)$ */

Action: Set $\texttt{Safe\_List}^w(p) \leftarrow \texttt{Safe\_List}^w(p) \cup \{h\}$ at every $w \in \texttt{Write}(h)$

Postcondition: $Marked\_Safe(h,p)$
/* $\equiv \ \forall w \in \texttt{Write}(h), h \in \texttt{Safe\_List}^w(p)$ */

Procedure $\text{MARK\_CLEAN}^r(p)$

Invoked at: $r \in \texttt{Roots}(p)$

Precondition: $Tree\_Clean(r,p)$
/* $\equiv \ \forall h \in \texttt{Hosts}(p),\ r = host(h) \Rightarrow Marked\_Safe(h,p)$ */

Action: Set $\texttt{Clean\_List}^w(p) \leftarrow \texttt{Clean\_List}^w(p) \cup \{r\}$ at every $w \in \texttt{Write}(r)$.

Postcondition: $Marked\_Clean(r,p)$
/* $\equiv \ \forall w \in \texttt{Write}(r), r \in \texttt{Clean\_List}^w(p)$ */

Procedure $\text{SCAN}^h(p)$

Invoked at: $h \in \texttt{Hosts}(p)$

Precondition: $Marked\_Clean(host(h),p)$

Action: invoke procedure $\text{VERIFY}^w(h,p)$ at every $w \in \texttt{Read}(h)$

Actual Postcondition: $Actual\_Scan\_End(h,p)$
/* $\equiv \ \forall w \in \texttt{Read}(h), \texttt{Safe\_List}^w(prev(p)) \subseteq \texttt{Clean\_List}^w(p)$ */

Required-Postcondition: $Scan\_End(h,p)$
/* $\equiv$ condition $Tree\_Clean(r,p)$ holds for every root $r \in \texttt{Near\_Roots}(h)$ that satisfied $r \in \texttt{Safe\_List}^w(prev(p))$ at all $w \in \texttt{Write}(r)$ upon invocation of the procedure. */

Procedure $\text{VERIFY}^w(h,p)$

Invoked at: $w \in \texttt{Read}(h)$ for $h \in \texttt{Hosts}(p)$

Precondition: $true$

Action: await Postcondition

Postcondition:
$\texttt{Safe\_List}^w(prev(p)) \subseteq \texttt{Clean\_List}^w(p)$

6 Implementation correctness

In this section we prove that the implementation presented in the previous section satisfies the specification of the conditions defined earlier.

In order to prove the correctness, it is required to prove that all three procedures (a) enjoy partial correctness with respect to their specifications, and (b) do not get "deadlocked." Both properties are trivial for the registration procedures MARK_SAFE and MARK_CLEAN, and therefore we now concentrate on the behavior of procedure SCAN.

Lemma 6.1 Upon termination of procedure $\text{SCAN}^h(p)$, the required postcondition $Scan\_End(h,p)$ is satisfied.

Lemma 6.2 Procedure $\text{SCAN}^h(p)$ does not deadlock.

7 Complexity analysis

In this section we prove our third main theorem.

Theorem 7.1 The above implementation of synchronizer $\mathcal{S}$ has communication and time complexities $Stretch_{com}(\mathcal{S}) = Stretch_{time}(\mathcal{S}) = O(\log^3 n)$.

Observing the protocol of the new synchronizer, we state the following key observation.

Lemma 7.2 In a synchronizer based on an m-regional matching $\mathcal{RW}$ as described, the total communication and time overhead per pulse are
$Stretch_{com}(\mathcal{S}) = Stretch_{time}(\mathcal{S}) =$
$O(\delta \cdot (Rad_{read} + Rad_{write}) \cdot (Deg_{read} + Deg_{write}))$

Combining Lemma 5.2 with Lemma 7.2, we deduce

Corollary 7.3 $Stretch_{com}(\mathcal{S}) = Stretch_{time}(\mathcal{S}) = O(\delta \cdot \log^2 n) = O(\log^3 n)$.

8 The general case

Our solution so far was based on the assumption that each network node is active in at most a single pulse of the protocol. Let us now remove this restriction, and allow nodes to be active more than once during the execution.

We create an "auxiliary" network as follows. Each node v is represented by $\tau = \mathcal{T}(\pi)$ auxiliary nodes $v_1, v_2 \ldots v_\tau$. Each network edge $(v, u) \in E$ (including all self-loops (v, v), for every $v \in V$) is represented by τ^2 auxiliary edges $(v_i - u_j)$ for all $i, j \leq \tau$. The auxiliary node v_i simulates the operation of the network node v at pulse i. By definition, in this simulation each auxiliary node is active in at most a single pulse of the protocol.

Note that the auxiliary network is needed only for managing the execution tree; the data structures of the `Read` and `Write` sets are still constructed on the original network, hence the complexity bounds do not change.

9 Conclusions

As presented in the abstract, the synchronizer $\mathcal{S}$ requires high space complexity. This is because some nodes are required to maintain long lists concerning the status of many other nodes. However, using the results of [AP], we can achieve an $O(polylog(n))$-space implementation with $O(polylog(n))$ additional overhead in time and communication.

The synchronizer proposed in this paper can be used to significantly improve the solutions of classical network problems, like Breadth First Search, for which existing asynchronous solutions are inefficient [Gal82, Fre85, Awe85a, Awe85b, AG85, AG87, Awe89].

Indeed, observe that there exists a distributed synchronous Breadth First Search (BFS) algorithm π with $\mathcal{C}(\pi) = O(E)$ and $\mathcal{T}(\pi) = O(D)$. Plugging this algorithm directly into our synchronizer $\mathcal{S}$, yields an asynchronous BFS algorithm ϕ with communication and time complexities $\mathcal{C}(\phi) = O(E + n \cdot \log^3 n)$ and $\mathcal{T}(\phi) = O(D \cdot \log^3 n)$.

The synchronizer can be implemented also for the "weighted complexity" model of [ABP90], which assumes the existence of edge weights representing traffic delays. In this model, we can obtain an asynchronous Shortest Paths algorithm ϕ with the same communication and time complexities.

References

[AAPS87] Yehuda Afek, Baruch Awerbuch, Serge A. Plotkin, and Michael Saks. Local management of a global resource in a communication network. In 28^{th} *Annual Symposium on Foundations of Computer Science.* IEEE, October 1987.

[ABP90] Baruch Awerbuch, Alan Baratz, and David Peleg. Cost-sensitive analysis of communication protocols. In *Proc. 9th ACM Symp. on Principles of Distributed Computing*, 1990. To appear.

[AG85] Baruch Awerbuch and Robert G. Gallager. Distributed bfs algorithms. In 26^{th} *Annual Symposium on Foundations of Computer Science.* IEEE, October 1985.

[AG87] Baruch Awerbuch and Robert G. Gallager. A new distributed algorithm to find breadth first search trees. *IEEE Trans. on Info. Theory*, IT-33(3):315–322, May 1987.

[AP] Baruch Awerbuch and David Peleg. Sparse partitions. These proceedings.

[AP90] Baruch Awerbuch and David Peleg. Efficient distributed construction of sparse covers. Unpubpished manuscript, April 1990.

[AS88] Baruch Awerbuch and Michael Sipser. Dynamic networks are as fast as static networks. In 29^{th} *Annual Symposium on Foundations of Computer Science*, pages 206–220. IEEE, October 1988.

[Awe85a] Baruch Awerbuch. Complexity of network synchronization. *J. of the ACM*, 32(4):804–823, October 1985.

[Awe85b] Baruch Awerbuch. Reducing complexities of the distributed max-flow and breadth-first-search algorithms by means of network synchronization. *Networks*, 15(4):425–437, Winter 1985.

[Awe89] Baruch Awerbuch. Distributed shortest paths algorithms. In *Proc. 21st ACM Symp. on Theory of Computing*, pages 230–240. ACM SIGACT, ACM, May 1989.

[CGZ86] C.T. Chou, I.S. Gopal, and S. Zaks. Synchronizing asynchronous bounded-delay networks. Research Report RC-12274, IBM Yorktown, October 1986.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. on Comput. Syst.*, 3(1):63–75, February 1985.

[ER90] Shimon Even and Sergio Rijsbaum. The use of a synchronizer yields maximum computation rate in distributed networks. In *Proc. 22nd ACM Symp. on Theory of Computing.* ACM SIGACT, ACM, May 1990.

[FLS87] A. Fekete, N. Lynch, and L. Shrira. A modular proof of correctness for a network synchronizer. In *Proceedings of the Amsterdam Workshop on Distributed Algorithms.* CWI, July 1987.

[Fre85] Greg N. Frederickson. A single source shortest path algorithm for a planar distributed network. In *Proceedings of 2nd Symp. on Theoretical Aspects of Computer Science*, January 1985.

[Gal82] Robert G. Gallager. Distributed minimum hop algorithms. Technical Report LIDS-P-1175, MIT, Lab. for Information and Decision Systems, January 1982.

[GHS83] Robert G. Gallager, Pierre A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. on Programming Lang. and Syst.*, 5(1):66–77, January 1983.

[Jaf80] Jeffrey Jaffe. Using signalling messages instead of clocks. Unpublished manuscript., 1980.

[LT87] K.B. Lakshmanan and K. Thulasiraman. On the use of synchronizers for asynchronous communication networks. In *Proceedings of the Amsterdam Workshop on Distributed Algorithms*. CWI, July 1987.

[PU89] David Peleg and Jeffrey D. Ullman. An optimal synchronizer for the hypercube. *SIAM J. on Comput.*, 18(2):740–747, 1989.

[SM86] Baruch Schieber and Shlomo Moran. Slowing sequential algorithms for obtaining fast distributed and parallel algorithms: Maximum matchings. In *Proceedings of the* 5^{th} *Annual ACM Symposium on Principles of Distributed Computing*, pages 282–292. ACM, August 1986.