Adapting to Asynchronous Dynamic Networks

EXTENDED ABSTRACT

Baruch Awerbuch[*] Boaz Patt-Shamir[†] David Peleg [‡] Michael Saks[§]

Abstract

The computational power of different communication models is a fundamental question in the theory of distributed computation. For example, in the synchronous model messages are assumed to be delivered within one time unit, whereas in the asynchronous model message delays may be arbitrary. Another important parameter of the model is the assumptions about the topology. In the dynamic topology model, links are assumed to crash and recover dynamically, but their status is known to the incident node processors. A meaningful computation can be carried out if the topology stabilizes for a sufficiently long period.

In this paper we show that the model of asynchronous, dynamic-topology network is equivalent, up to polylogarithmic factors, to the synchronous, static (i.e., fixed-topology) model. Specifically, we present a simulation methodology of synchronous static protocols that can withstand arbitrary link delays and changing topology at the expense of only polylogarithmic blowup in the running time, the number of messages, and the space requirement. Previous methods entailed a linear blowup in at least one of these resources.

The generality of our method is demonstrated by a series of improvements for important applications, including Breadth First Search, computing compact efficient routing tables, and packet routing on asynchronous networks.

1 Introduction

Communication networks, although common, do not share a robust mathematical model. There is a considerable variety of parameters by which one may model such networks. A crucial question, for example, is what are the assumptions we make about the delivery of the messages. In the *synchronous* model, we assume that messages are delivered within one time unit, and that processors have access to a clock by which they can tell whether messages have arrived or not. By contrast, in the *asynchronous* model the message delays are arbitrary. However, message delivery can be easily verified in this model by explicit acknowledgements, at the cost of high communication overhead.

Another important issue in modeling communication networks is reliability of the links. The simplest model is the *static network* model, based on the assumption that links maintain operational status forever. A slightly more realistic assumption is the *dynamic network* model: links may crash and recover, and the endpoints are constantly notified of the links' status. A meaningful computation is expected to be carried out once the topology remains stable for sufficiently long time. A simple solution to the problem of unknown topology is using a *reset* procedure, that will restart the computation whenever a topological change is detected. The best reset procedures [1, 8], however, have

[*] Dept. of Mathematics and Lab. for Computer Science, MIT, Cambridge, MA 02139. Supported by Air Force Contract TNDGAFOSR-86-0078, ARO contract DAAL03-86-K-0171, NSF contract CCR8611442, and a special grant from IBM.

[†] Lab. for Computer Science, MIT, Cambridge, MA 02139. Supported by ONR contract N00014-85-K-0168, by NSF grants CCR-8915206, and by DARPA contracts N00014-89-J-1988.

[‡] Department of Applied Mathematics, The Weizmann Institute, Rehovot 76100, Israel. Supported in part by an Allon Fellowship, by a Bantrell Fellowship and by a Walter and Elise Haas Career Development Award.

[§] Rutgers University and UCSD. Supported in part by NSF contract CCR-8911388.

high running time.

Clearly, a protocol which is designed to work on a dynamic asynchronous network (dubbed hereafter "dynamic asynchronous protocol"), can run on a static synchronous network without any complexity penalty — the static synchronous model is just a special case of the dynamic asynchronous model. In this paper we show that, perhaps surprisingly, the converse also nearly holds, namely, a synchronous, static protocol can be simulated on an asynchronous dynamic network with polylogarithmic blowup in the time, communication and space complexity. Before we state our results more precisely, it is necessary to give a formal definition of the problem.

The models. The *asynchronous* model we consider is the standard model of a point-to-point communication network. 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. 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 $O(\log n)$ bits, so that a processor's identifier can be accommodated in a message. We consider only protocols in which all nodes are receiving at least one message in the course of execution.

A synchronous network is a variation of the above 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.

In a *static network* all links are operational at all times. This allows us to do some offline pre-processing of the communication graph, that may be used by the actual problems which arrive online. In a *dynamic network* [1], the set of operational links is a function of time, i.e., links may crash and recover arbitrarily. The nodes are assumed to know the status of their incident links at all times. We assume that eventually, the topology ceases to change, and the protocols should produce results with respect to the *final topology*.

The complexity measures are defined as follows. The *communication complexity* of an algorithm π, *Comm*(π), is the worst-case number of messages sent during a run of the algorithm. The *time complexity* of a synchronous algorithm π, *Time*(π), is the number of pulses generated during the run. For an asynchronous algorithm π, *Time*(π) is the largest number of time units to complete a run, assuming that each message incurs a delay of at most one time unit. The *space complexity*, denoted *Mem*(π), is the maximal amount of space per link used at a node by the algorithm, measured in units of size $O(\log n)$. Another measure for algorithms that will prove useful here is the *congestion* of a link, which we define to be the maximal number of messages that traverse that link in any run. The congestion of a protocol π, denoted *Con*(π), is the maximal link congestion over all links. In the dynamic model, we consider only the *quiescence* complexities, i.e., the resource requirements after the last topological change.

On an abstract level, a *dynamic synchronizer* Υ is an algorithm to transform a static synchronous protocol π into an equivalent asynchronous protocol $\Upsilon(\pi)$. The efficiency of a synchronizer Υ is measured by the increase in the communication, time and space complexity of the result of the transformation defined as follows.

$$\begin{aligned} Stretch_{comm}(\Upsilon) &= \max_{\pi} \frac{\mathit{Comm}(\Upsilon(\pi))}{\mathit{Comm}(\pi)} \\ Stretch_{time}(\Upsilon) &= \max_{\pi} \frac{\mathit{Time}(\Upsilon(\pi))}{\mathit{Time}(\pi)} \\ Stretch_{mem}(\Upsilon) &= \max_{\pi} \frac{\mathit{Mem}(\Upsilon(\pi))}{\mathit{Mem}(\pi)} \end{aligned}$$

We refer to $Stretch_{comm}(\Upsilon)$, $Stretch_{time}(\Upsilon)$ and $Stretch_{mem}(\Upsilon)$ as the *communication stretch*, *time stretch* and *space stretch* of a synchronizer Υ, respectively.

We occasionally use the "soft Oh" notation, that absorbs polylogarithmic factors. More precisely, $f(n) = \tilde{O}(g(n))$ if there exist integers k, N such that for all $n \geq N$, we have $f(n) \leq (\log n)^k \cdot g(n)$.

Previous results. The synchronizer methodology was introduced by Awerbuch [3]. Synchronizers, and various aspects or applications thereof, were studied in [15, 4, 12, 25, 14, 17, 11, 23, 13]. The synchronizers fall into two categories, namely static and dynamic synchronizers.

Reference		Space	Comm.	Time
[11]	(det.)	$\tilde{O}(n)$	$\tilde{O}(n)$	$\tilde{O}(1)$
[1, 9]	(det.)	$\tilde{O}(n)$	$\tilde{O}(1)$	$\tilde{O}(n)$
This paper	(rand.)	$\tilde{O}(1)$	$\tilde{O}(1)$	$\tilde{O}(1)$

Figure 1: Comparison of stretch factors of dynamic synchronizers ($\tilde{O}(\cdot)$ absorbs polylog factors).

Among the static synchronizers [3, 23], the best we know of is the protocol introduced by Awerbuch and Peleg [9], which has polylogarithmic time and communication stretch, but needs linear space at some nodes. Another disadvantage of this synchronizer is that the congestion of the protocol may be linear.

The dynamic synchronizers are typically based on repeated computation when a topology change is detected. An optimal-time dynamic synchronizer was introduced by Awerbuch and Sipser in [11]. There, a complete history is maintained at each node, and relevant actions are undone when a topology change occur. This method has constant time stretch, but high costs in both communication and space (see Figure 1). In [1] another method is used. There, it is shown how to adapt a static protocol to run on a dynamic network, but this simulation has a large time stretch. When combined with the best static synchronizer, the reset protocol of [1] yields a dynamic synchronizer, whose complexities are given in Figure 1.

Our results. In this paper we give the first polylog time, space and communication simulation of synchronous, static protocols on dynamic asynchronous networks. As a byproduct, a better static synchronizer is obtained, improving on the best known static synchronizer in terms of congestion and maximum space requirement per edge. Our result can also be interpreted as the best known "from-scratch" synchronizer scheme, i.e., synchronizing when we are not allowed use pre-computed structures. The results are not deterministic, in that there exists a polynomially small probability that the simulation fails. However, this randomization origins at a certain "cover-construction" algorithm used as a subroutine, and there is hope that a deterministic polylog distributed algorithm can replace it in the future.

Specifically, we prove the following result for dynamic synchronization.

Theorem 1.1 There exists a randomized dynamic synchronizer with polylogarithmic time, space, and communication stretch factors, and with probability of success $1 - O(1/n)$.

The following theorem states our result for static synchronization.

Theorem 1.2 There exists a randomized static synchronizer which does not use pre-computed structures, with polylogarithmic time, space, and communication stretch factors, and with probability of success $1 - O(1/n)$.

Applications. In the following corollaries, we give a partial list of some immediate applications of the (static and dynamic) synchronizers of this paper. The results are ordered from basic techniques to more specific applications.

Our most elementary improvement is in the basic task of distributed Breadth First Search. The best known distributed BFS algorithms [6, 7, 9, 2] had $\Omega(n)$ stretch in at least one of the three relevant parameters (communication, time, space), in both the static and the dynamic cases. Here, we obtain a result for a generalized version of BFS, where any subset of the nodes may be the originators (whereas in the "conventional" BFS there is a single originator). Unlike the sequential case, new difficulties arise in distributed multiple-originator BFS, since we must synchronize potentially remote sources.

Corollary 1.3 Distributed Breadth First Search (with single or multiple originators) can be performed by a randomized algorithm on an asynchronous (static or dynamic) network with E edges and final diameter D using $\tilde{O}(1)$ space, in $\tilde{O}(D)$ time and $\tilde{O}(E)$ messages, with probability of success $1 - O(1/n)$.

Sparse covers (see Section 2 for definition) proved to be a useful tool in designing distributed algorithms (see, for example [10]). It is known how to construct sparse covers in the static synchronous model (e.g., the randomized algorithm of [21], or the deterministic one of [5]). However, all known *asynchronous* implementations required $\Omega(n)$ overhead in either communication, time, or space. Using the synchronizer of this paper, we obtain the following result.

Corollary 1.4 It is possible to construct sparse covers for a dynamic asynchronous network with E edges and final diameter D by a randomized algorithm using $\tilde{O}(1)$ space, in $\tilde{O}(D)$ time and $\tilde{O}(E)$ messages, with probability of success $1 - O(1/n)$.

We demonstrate the applicability of Corollary 1.4 with the following result [22, 10].

Corollary 1.5 It is possible to compute routing tables in a dynamic asynchronous network with E edges and final diameter D by a randomized algorithm using $\tilde{O}(1)$ space, in $\tilde{O}(D)$ time and $\tilde{O}(E)$ messages, such that with probability $1 - O(1/n)$, the tables are polylog size and the stretch in the distances is polylogarithmic.

Scheduling packets to be transmitted over communication links is one of the basic issues in parallel processing. Many packet routing schemes have been designed and analyzed for the synchronous setting (see [20] for a comprehensive survey). Using the result of this paper, such synchronous schemes can be applied to the asynchronous setting, with only polylogarithmic increase in the size of the queues and the latency of the packets. For example, the randomized routing protocol of Leighton, Maggs and Rao [19] delivers all packets with $O(\log n)$ multiplicative time overhead, using $O(\log n)$ size queues. Applying the new simulation to this protocol yields an asynchronous packet routing protocol where the queue sizes and the latency of the packets are only a polylog factor worse than optimal.

Corollary 1.6 Given any set of packets and their paths, there exists a randomized asynchronous scheduler with polylogarithmic overhead in communication, time and space, and probability of success $1 - O(1/n)$.

Overview. The idea used here for adapting static protocols to dynamic networks is the *local rollback*. Essentially, the local rollback rule is as follows. Each node maintains a complete history of its execution. Whenever a topological change is detected, the node "undoes" all its actions, and informs its neighbors which of the messages they received are obsolete. Upon receiving such a notice, the neighbors undo their "infected" actions, and propagate the information. This idea is well known [16], and was explicitly used in the dynamic synchronizer of [11], by combining it with a simple static synchronizer. The apparent drawback of this approach, however, is that the size of storage required per link is proportional to the total number of messages that traverse that link in the synchronized version.

The basic intuition for circumventing this problem is that we can save greatly on the number of states that should be stored, if we carefully distribute the message traffic among the nodes, so that only few messages will traverse each link. We shall use the "sparse covers" data structure [10] to accomplish this goal. Thus, the dynamic synchronizer problem is reduced to the problem of how to construct sparse covers "from scratch" on an asynchronous network, while keeping the congestion of all the links small. For the fast randomized algorithm of Linial and Saks [21], it can be shown that number of messages crossing each network link is (with high probability) polylogarithmic, assuming that the network is synchronous. In an asynchronous execution, however, the number of messages crossing a link could be $\Omega(n)$.

This naturally suggests the use of synchronizers. In order to save communication, we would like to use the synchronizer of [9], which has polylogarithmic overhead in time and communication. Unfortunately, this synchronizer scheme (as well as its predecessors [3, 23]), assumes the existence of some pre-computed data-structure in the network. Specifically, [9] requires sparse covers. This is the key difficulty in the algorithm: *the synchronizer requires sparse covers, while the cover construction requires a synchronous network.* This Gordian knot is cut by incorporating a synchronous cover construction with the synchronizer protocol in a mutually-recursive fashion. It turns out that it is possible to invest a constant amount of work in constructing some initial covers, and then let the synchronizer and the cover-construction "bootstrap" each other, advancing step by step to produce the desired result, while incurring only polylog penalty.

A few other modifications of the synchronizer [9] are required, stemming from the following properties of the algorithm. First, some links have linear congestion, and secondly, certain nodes have linear space requirement. Our first step, described in Section 3, is to eliminate these bottlenecks. These improvements, when incorporated with the asynchronous cover construction, yield a new synchronizer for a *static asynchronous* network, without assuming the existence of any pre-computed structure. All of its relevant complexity measures (communication overhead, space overhead, time overhead, congestion) are polylogarithmic, thereby proving Theorem 1.2.

Finally, we show how this construction can be made to work on a dynamic asynchronous network. For the synchronizer initialization part, we use the universal rollback technique. For the actual execution of the protocol, we use replication and reset on independent parts of the network, as explained in Section 6.

Organization of this paper. Many important details are omitted from this abstract due to the lack of space. Rather, we try to give intuition as to what are the main ideas of construction as follows.

In Section 2 we introduce several notational conventions and basic concepts that are essential for the exposition of the algorithm. In Section 3 we explain how to modify the algorithm of [9] to avoid local space and congestion bottlenecks. In Section 4 we show how to efficiently construct sparse covers in an asynchronous network, using a generalized version of the synchronous cover-construction protocol of [21], and the synchronizer of Section 3. In Section 5 we present the universal local rollback technique, that transforms a static protocol to a dynamic one. Finally, in Section 6, we integrate the construction and discuss how to make a given static synchronous protocol to run on a dynamic asynchronous network.

2 Preliminaries

In this section we define some key concepts that we use in the description of the algorithm. We also give specifications of the tools we use in the construction.

Synchronizer problem. A synchronizer operates by generating a sequence of local *pulses* at each processor of the network, satisfying the following property.

Definition 2.1 Pulse p is said to be *safely generated* by a processor u if it is generated after all the messages of the algorithm, sent to u by its neighbors during their pulse $p-1$, were received.

Given a synchronous algorithm π, and its (synchronous) time complexity τ, the task of the synchronizer protocol is to safely generate pulses $0,\ldots,\tau$ in the nodes participating at π. 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.

Simple synchronization strategy. The following synchronization scheme, traditionally called α [3], is a naive scheme with high communication overhead. We shall use this strategy in the new protocol, in a way that will be explained in the sequel. The α protocol is defined by the following simple rule.

Algorithm 2.2 (α synchronizer)
Messages (possibly null messages) are sent from a node to *all* its neighbors at *all* pulses. When a node has received all messages of pulse p, it proceeds to pulse $p+1$.

Note that in this strategy, each edge delivers one message in each direction at every pulse, and hence its communication stretch can be linear in the running time of the algorithm. However, this strategy is optimal in terms of its time and space overheads.

Data structures. Given an execution of a protocol, define the *execution parent* of a node as the first node from which it received a message. The execution parent relation defines the *execution forest* in the natural way. We will assume that the nodes maintain pointers to their parent in the execution forest, i.e., the nodes mark the link from which they get the first message. The cost associated with this is only $O(1)$ space per edge.

Another concept which we use is the graph-theoretic notion of sparse covers. Let $G=(V,E)$ be a graph. The *distance* between two vertices u,w in G, denoted $dist_G(u,w)$, is the number of edges on the shortest paths in G between u and w. (We normally omit the subscript G where no confusion arises.) The *j-neighborhood* of a vertex $v\in V$ is defined as $\mathcal{N}_j(v)=\{w \mid dist(w,v)\leq j\}$.

For a set of vertices $W\subseteq V$, let G_W denote the subgraph induced by W in G. A *cluster* is a pair (W,w) where w is a vertex and W is a subset of nodes containing w such that G_W is connected. The *radius* of the cluster, $Rad(W, w)$ is the maximum distance from w to any other vertex in the cluster. The *tree* of a cluster is the shortest path tree rooted at w, in which the parent of node x is the neighbor of x of highest *ID* among those whose distance to w is minimum. For an integer k, the k-*kernel* of a cluster (W, w) is the set of nodes w whose k-neighborhood is entirely contained in W.

A set $\mathcal{W} = \{(W_1, w_1), (W_2, w_2), \ldots, (W_q, w_q)\}$ is called a *cover* of G if $\cup_i W_i = V$. The radius of the cover, denoted $Rad(\mathcal{W})$, is the maximum radius of all of its clusters. The *overlap* at node v in the cover $\mathcal{W}$, is the number of clusters containing v; the maximum overlap over all nodes, denoted $\deg(\mathcal{W})$, is the *degree* of $\mathcal{W}$.

We now come to the definition of covers, which will serve us as one of the basic data structures.

Definition 2.3
A cover $\mathcal{W} = \{(W_1, w_1), \ldots, (W_q, w_q)\}$ is a k-*cover* if every node is in the k-kernel of at least one of the clusters of $\mathcal{W}$. A k-cover is a *sparse* k-*cover* if the maximum overlap at all nodes is $O(\log n)$. *Complete sparse covers*, sometimes referred to as *sparse covers*, is a collection of sparse 2^i-covers for all $0 \leq i \leq \delta$.

Reset protocol. *Reset* is a procedure that we use in the algorithm as a subroutine. Intuitively, reset may be invoked at any node, and its effect is to output a *restart signal* at all the nodes of the system in a consistent way.

More formally, the reset problem is defined as follows. It involves two actions named *reset request* and *reset signal*. The task of the reset protocol is that after the topological changes stop, if one of the nodes makes a reset request and no node makes infinitely many requests, then (i) in finite time all the nodes in the connected component of a requesting node receive a reset signal, (ii) no node receives infinitely many reset signals, and (iii) if $e = (u, v)$ is a link in the final topology, then the sequence of messages sent from u after the last reset signal at u, is identical to the sequence of messages received at v after the last reset signal at v.

The best known reset protocol on a dynamic network (e.g., [1, 8]) runs in time bounded by the length of the longest simple path in the network, with constant congestion and memory per link. Note that for a general graph, the running time is $O(n)$, but for a tree, the runing time is in the order of the height of a tree.

3 A better static synchronizer

In this section, based on the protocol of [9], we describe an improved synchronizer protocol for static networks, using sparse covers (in the next section we will dispense of this assumption). The communication and time stretch, as in [9], are $O(\log^3 n)$ and $O(\log^3 n)$, respectively. However, the space overhead of the synchronizer is $O(\log^3 n)$ per edge in the worst case, and the congestion on the links is $O(\log^3 n)$, whereas in the protocol of [9], the space overhead is $\Omega(n)$, and $\Omega(n)$ messages may traverse the same link.

In the description that follows, we assume that each node u sends messages only in one pulse, denoted *pulse*(u), and that the number of pulses is bounded by $D(G)$, the diameter of the graph. We shall explain later the case for general protocols. Thus, the synchronizer problem is reduced to deciding, at each node u, when did it get all the messages sent to it from its neighbors on pulses before *pulse*(u).

We begin with a brief intuitive description of the synchronizer. The basic observation is that the next pulse can be safely generated at node u if for all nodes v, *pulse*(u) $-$ *pulse*$(v) \leq$ *dist*(u, v), and the crux of the algorithm is how to disseminate this information efficiently. More specifically, the rule is that pulse $p + 1$ is generated only when the node explicitly "knows" that all nodes in some neighborhood have been acknowledged for their pulse p messages. The size of the neighborhood is a function of the pulse value. Note that for neighborhoods larger than the set of adjacent nodes this condition is more strict than the correctness requirement, thereby incurring a slowdown of the algorithm. However, it is not necessary to verify whether the correctness condition holds in each pulse, thereby saving in the number of messages. With a careful choice of of the neighborhoods and information dispersal mechanism, this strategy results in only polylogarithmic time and communication overhead.

Let us first describe the way to communicate the safety information. We use two distributed data structures, namely the *execution forest* and *sparse covers*. The idea is as follows. A node v safely generates pulse $p + 1$ only after it gets a GO_AHEAD message from one of its ancestors, called *inspector*(v). The node *inspector*(v), in turn, forwards a query to *inspector*(*inspector*(v)), called hereafter *supervisor*(v), informing it also whether it has any active subordinate. The *supervisor* gathers information for a region in the graph of sufficiently large radius, so that all the “close” nodes will be certified to be “non-threatening”, i.e., they did not send any unacknowledged message at pulse p. The data structure used by the algorithm for this purpose is sparse covers, which are assumed to be already existing in the network. Since for each node there is a cluster in which its entire 2^i-neighborhood is contained, it is sufficient for the *supervisor* nodes to collect the safety information along the cluster trees.

We make things concrete in the following definition.

Definition 3.1 Let u be a node with $pulse(u) = p \leq D(G)$. The *level* of p is the number of trailing 0's in the binary representation of p, i.e.,

$$\ell(p) = \begin{cases} i, & \text{if } p = (2j-1)\cdot 2^i \\ \infty, & \text{if } p = 0 \end{cases}$$

The *inspector level* of pulse p is

$$\max\left\{ \tilde{p} \;\middle|\; \begin{array}{l} \ell(\tilde{p}) = \min\{\ell(p)+1, \delta\} \\ \tilde{p} \leq p - 2^{\ell(p)} \end{array} \right\}.$$

The *inspector node* of u is the ancestor of u in the execution tree, whose pulse is the inspector level of pulse p. The node u is said to be a *subordinate* of *inspector*(u). The *supervisor* of u is the inspector of its inspector.

With these notations, we can describe the protocol as follows (see Figure 2). Whenever a regular message (i.e., a message which is not generated as a part of the synchronization procedure) is received, it is acknowledged with an ACK_0 message. Whenever a node receives all the ACK_0 messages, it forwards an ACK_1 message to its inspector up on the execution tree. This message also informs the inspector whether the node sent any messages. When an inspector receives an ACK_1 message indicating that one of its subordinates has sent a message at pulse p, the inspector sends a REG message to all the centers of the $2^{\ell(p)+5}$ clusters in which it is a member, indicating that it has *threatening* subordinates, i.e., nodes that might affect generation of pulses in the informed clusters.

After receiving all the ACK_1 messages, the inspector forwards an ACK_2 message to its inspector, i.e., to the supervisor of the node. When a supervisor has received all the ACK_2 messages, it sends DE-REG messages to all the cluster-centers at distance $2^{\ell(p)+5}$, thereby signaling that none of its subordinates is still threatening. Once the center of a cluster detects that all REG messages were matched by DE-REG messages, it broadcasts OK to all the requesting inspectors, thereby notifying them that the cluster is “safe”. When the supervisor receives OK messages from all the clusters in which it is member, it sends GO_AHEAD messages to its sub-subordinates.

The correctness of the synchronizer at Figure 2 is implied by the following lemma, whose proof is omitted.

Lemma 3.2 Pulse $p+1$ is generated at a node only after all pulse p messages from all the nodes at distance $2^{\ell(p)+3}$ were acknowledged.

Before we can analyze the overhead complexity of the synchronizer, we need to specify the way the information is propagated on the execution forest and the cluster trees. We use the following properties of the definition of level.

Lemma 3.3 Let $i \leq \delta$ be a level number. Then

Upon receipt of non-synchronizer message:
  send ACK_0 back

Upon receipt of all ACK_0 for pulse p messages:
  if no messages were sent
    send tree-AND(ACK_1, dead) to *inspector*(v)
  else send tree-AND(ACK_1, alive) to *inspector*(v)

Upon receipt of first (ACK_1, alive):
  send REG(v) to the center of all $2^{\ell(p)+4}$ clusters

Upon receipt of all ACK_1:
  send tree-AND(ACK_2) to *inspector*(v)

Upon receipt of all ACK_2:
  send DE-REG to the center of all $2^{\ell(p)+4}$ clusters

Upon receipt of all OK(p') where $\ell(p') = \ell(p)+1$
  send GO_AHEAD(p') to subordinates of subordinates

Upon receipt of GO_AHEAD(p):
  send pulse p messages

Figure 2: synchronizer code for node v with *pulse*$(v) = p$

1. $|\{p : \ell(p) = i\}| = \Theta(2^{\delta - i})$.

2. $|\{p' : \ell(p') < i < \mathit{inspector}(p')\}| = O(\delta)$.

Let us now specify the way the synchronizer messages are sent (see Figure 3). First, note that all ACK messages are sent up the execution tree in order to compute global AND of the leaves. The number of messages is reduced by a simple *combining rule*: a node forwards the appropriate ACK message only after it received all the corresponding ACKs from its descendents. The number of ACKs traversing a single link this way is proportional to the number of nodes above the link which serve as inspectors of nodes below the link, and this number is, by Lemma 3.3, $O(\log n)$. We also need $O(\log n)$ additional memory for every link, to mark the arrival of the various ACK messages of the different pulses. This combining rule is denoted in Figure 2 as "tree-AND".

Next, consider the registration messages. The task of the these messages is to let the nodes know whether there are threatening nodes, by first registering the nodes as threatening (the REG message), and then deleting this registration (the DE-REG messages). Only when there are no threatening nodes, the response to queries changes. Thus, we actually compute a global OR of the "threatening" status. Hence we can save on the number of messages required by a different combining rule as follows. When a REG message is sent, it is forwarded along the branches of the cluster-tree, marking its trace on the links with a *(source, destination)* pair, until it hits a mark of a trace of another REG message with the same destination — or until it reaches the destination. When a DE-REG message is sent, it follows the trace, and unmarks it so long as the message origin is the only source of this trace. In other words, upon reaching a node, the trace just traversed is unmarked; if no other incoming marked trace remained, the message continues to the next node, else the procedure terminates. In this way, the center is marked by some trace only if there is some node that has sent a REG message and did not send a DE-REG one. The queries, made by the inspectors, are combined by the tree-AND rule above.

The OK and the GO_AHEAD messages are forwarded by a simple broadcast over the cluster and execution trees, respectively.

The number of messages that traverse a cluster-tree link is, with this procedure, proportional to the number

```
Upon receipt of "tree-AND(m)" from link e:
  mark (e, m)
  if ∀e [mark(e, m)]
     send "tree-AND(m)" on execution-parent

Upon receipt of REG from link e:
  mark (e, REG)
  if ¬mark(parent, REG) send REG on cluster-parent

Upon receipt of DE-REG from e:
  unmark (e, REG)
  if ∀e[¬mark(e, REG)] send DE-REG on cluster-parent
```

Figure 3: Communication protocols for the synchronizer

of tree clusters in which it participates, times the number of pulses in which the cluster center participates in the registration and de-registration process. By Definition 2.3, each node is a member in $O(\log n)$ clusters, implying that the number of messages that traverse a link is $O(\log n)$ per pulse. Lemma 3.3 states that the number of pulses with a given level is $O(\log n)$, implying that the number of pulses in which a cluster-center participates is $O(\log n)$, and therefore the congestion of the cluster-tree links is $O(\log^2 n)$ overall. As before, the combining rule requires to maintain for each link the possible traces, which implies additional $O(\log^2 n)$ space.

We summarize the result of this section in the following theorem.

Theorem 3.4 There exists a deterministic static synchronizer with polylogarithmic time, space, communication, and congestion overhead.

We conclude this section by addressing the problem of nodes which send messages in more than one pulse. First, clearly if some node sends messages in many pulses, then necessarily there are some links with high congestion. The converse, however, does not necessarily hold: even if all nodes send messages only in "a few" pulses, it still may be the case that there are some highly congested links. The difficulty stems from the fact that the number of times a cluster center is involved in the registration process cannot be bounded, in general. The "bad" phenomenon occurs when many of the pulses are executed in a small-diameter region of the graph, thereby incurring large congestion at the links of the local cluster-trees. However, as we show in the next section, the cover construction protocol of [21] features the pleasing property that the congestion at all links is at most polylogarithmic. This non-trivial property is crucial when we come to make the protocol dynamic, in Section 5.

construct 32-cover: apply α for $T(32)$ pulses
for $i \leftarrow 1$ **to** $5 + \log \tau$ **do** { stage i }
  for $j \leftarrow 1$ **to** $\frac{T(2^{i+1})}{2^i}$ **do** { phase j }
    generate next 2^i pulses to construct 2^{i+5}-cover
    send tree-AND$(j \cdot 2^i)$ to root of 2^{i+5} cluster
    wait until pulse $(j \cdot 2^i)$ at all neighbor clusters

Figure 4: asynchronous bootstrap construction of covers

4 Asynchronous construction of sparse covers

In Section 3, we have seen an efficient synchronizer protocol that uses a pre-computed structure, namely sparse covers of the graph. For a given static network, on which many computational tasks are carried out, a pre-processing stage is a reasonable assumption that pays off eventually, since all tasks use the same pre-computed structure. In a dynamically changing network, however, we cannot assume such pre-processing — a transient period of topology changes may occur, making all the pre-computed structures obsolete. A dynamic protocol must be able to withstand such changes, and to adjust itself in reasonable time. As usual, our interpretation for "reasonable" is "within a polylogarithmic factor from optimal". To achieve this goal, we must have an *asynchronous cover-construction protocol.* Let us ignore for the moment the issue of a dynamically changing network. As we shall see later, if we can manage the protocol to deliver only few messages through each link, we can sustain dynamic changes of the topology with only a small penalty in space.

In this section, we present an asynchronous protocol that, for a given k, produces a sparse k-cover of the network in $\tilde{O}(k)$ time, and $\tilde{O}(1)$ congestion and space requirements per link. We remark that cover construction protocols that we know of are either sequential [10], or synchronous [21]. To overcome this difficulty we use a simple but powerful idea: we combine a synchronous protocol with the synchronizer described earlier. This seems to be a circular approach: how can the synchronizer work without the covers?

The solution we propose is to work in stages. Stage i will be started when we have a sparse $O(2^i)$-cover, and will end with a sparse $O(2^{i+1})$-cover. Suppose we are given a synchronous algorithm that constructs a 2^i-cover in $T(2^i)$ time. Our strategy is to generate, in each stage i, $T(2^i)$ safe pulses so that the synchronous cover-construction algorithm is able to deliver its promise. Assuming that the stage begins with an $O(2^i)$-cover, the synchronizer protocol described in Section 3 can safely generate 2^i pulses: for this amount of pulses, a node depends only on its 2^i-neighborhood, which is contained in the appropriate 2^i-cluster. Consequently, our solution is to divide each stage into $T(2^i)/2^i$ phases (see Figure 4). During each phase, the pulses are generated by the synchronizer as described in the previous section. To start the next phase, however, we need an "inter-cluster coordination", i.e., we need to know that all neighboring clusters are not too far behind. We do this by verifying, for each cluster, that all neighboring clusters have finished phase j in order to proceed to phase $j+1$. This can be viewed as coordinating the phases of the clusters by the α synchronizer. We repeat this recursive construction sufficiently many times so that we can generate safely the required number of pulses. More precisely, if our ultimate goal is to generate τ safe pulses, we will construct the $2^{5+\log \tau}$-cover required to generate safely τ pulses.

As for the implementation, we recall that clusters are neighbors only if they overlap. Hence, the α synchronization protocol can be carried out simply by ensuring that all the nodes within a cluster have reached the desired pulse number. This can be done in the "tree-AND" fashion (see Figure 3). When the cluster center concludes that all active nodes in the cluster have reached that pulse number, it broadcasts a RELEASE message over the cluster tree.

We also need to enable the first stage to work, i.e., we need to construct an initial 2^5-cover required to produce the synchronizer to generate $2^0 = 1$ safe pulse. We will do this by generating $T(32)$ safe pulses using the α synchronizer (Algorithm 2.2).

The scheme is given in Figure 4, and its correctness is implied by the correctness of the underlying synchronizer protocol, in conjunction with the following lemma.

Lemma 4.1 By the beginning of phase j of stage i at some node v, all nodes in distance at most 2^{i+5} from v have already generated pulse $j \cdot 2^i$.

The complexity of the cover construction protocol can be abstractly summarized as follows, assuming that $T(k)$ is linear in k. The time complexity is bounded by

$$\sum_{i=1}^{\log\tau} \sum_{j=1}^{\frac{T(2^{i+1})}{2^i}} \tilde{O}\left(T(2^i)\right) = \sum_{i=1}^{\log\tau} \tilde{O}\left(T(2^{i+1})\right) = = \tilde{O}\left(T(\tau)\right) .$$

The space complexity remains unchanged, namely $O(\log^3 n)$. and the number of messages at a link incurred by the initialization (line 1) and the periodic coordination (lines 5 and 6) is in the order of

$$T(32) + \sum_{i=1}^{\log\tau} \sum_{j=1}^{\frac{T(2^{i+1})}{2^i}} 1 = T(32) + \sum_{i=1}^{\log\tau} \frac{T(2^{i+1})}{2^i} = = T(32) + O(\log\tau \cdot \frac{T(\tau)}{\tau}) .$$

However, to bound the total number of messages at the links, we must count the messages sent by the synchronizer (line 4). For this, we turn to the specific cover-construction algorithm, viz. the randomized cover construction algorithm of Linial and Saks [21]. This algorithm proceeds as follows. Each node u picks a random value $R_u = (ID_u, r_u)$ as follows. The first component, ID_u, is picked uniformly from $[1..n^4]$, and the second component, r_u, has values in the range $[1..\lceil\log n\rceil]$ with *truncated geometric distribution*, i.e., $\Pr[r_u = k] = 2^{-k}$ for $0 < k < \lceil\log n\rceil$, and $\Pr[r_u = \lceil\log n\rceil] = 2^{-\lceil\log n\rceil+1}$. Notice that with high probability, the R_u's are distinct, and we will identify a node u with its R_u. Intuitively, the role of r_u is to determine the distance to which ID_u will be broadcast. The goal is that each node will get a message from the node with the highest *ID* that can reach it. More formally, say that a node u with $R_u = (ID_u, r_u)$ is *dominated* by a node v with $R_v = (ID_v, r_v)$ if $ID_v \geq ID_u$, and $r_v \geq r_u$. The algorithm proceeds in $\lceil\log n\rceil$ pulses as follows. Whenever a message (ID, r), which is dominated by one of the previous messages is received, it is discarded. If (ID, r) is not dominated by any previous message, then its origin is marked; if r is greater than the current pulse number, it is re-broadcast to the neighbors. Note that this procedure guarantees that each node u will get a message containing $\max\{ID_v : dist(u, v) \leq r_v\}$. The node which is the origin of this message is considered to be the center of the cluster, and the cluster tree is the paths along which the messages propagated. In [21] it is shown that in $O(\log^2 n)$ pulses, this procedure produces 1-cover, with probability $1 - O(1/n^2)$, using $O(\log n)$ messages per link and $O(\log n)$ space per node. Given $0 \leq i \leq \delta$, this procedure can be extended to generate 2^i-cover as follows. We let r_u range over the values $\{1 \cdot 2^i, 2 \cdot 2^i, \ldots, \log n \cdot 2^i\}$, with $\Pr[r_u = m \cdot 2^i] = 2^{-i}$ for $0 < m < \lceil\log n\rceil$, and $\Pr[r_u = \lceil\log n\rceil \cdot 2^i] = 2^{-\lceil\log n\rceil+1}$. Thus we have a protocol that produces k-covers with probability $1 - O(n^2)$ in time $T(k) = O(k\log^2 n)$. Applying the bootstrap scheme of Figure 4 in a *synchronous* network, this protocol yields k-sparse covers for $k = 2^0, 2^1, \ldots, 2^{\log\tau}$ with probability $1 - O(1/n)$ in $O(\tau\log^2 n)$ synchronous pulses, with maximal link congestion $O(\log^2 n)$ messages, using $O(\log^2 n)$ space at each node.

We need to show that this protocol for cover construction, has low congestion and space requirement when composed with the synchronizer described in Section 3. For this, we have the following key lemma.

Lemma 4.2 The worst-case number of messages which arrive at any node in the course of asynchronous construction of a k-cover is $\tilde{O}(1)$, with probability $1 - O(1/n)$.

Proof Sketch: Since the number of messages a node gets in the synchronous construction is $O(\log^2 n)$, it follows that the number of ACK messages crossing execution-tree links is $O(\log^3 n)$. The problem is how to bound the number of times a single cluster-center is queried for threatening nodes. It suffices to prove that given any set of nodes S with diameter $d\log n$, the number of pulses active in this set, and divisible by d, is $O(\log^3 n)$ w.h.p. We do this as follows.

Pick any $x \in S$, and consider the ball B_0 of radius $r = d\log n$ around x. Clearly, $S \subseteq B_0$. Consider the messages whose origins are inside S. These messages are active in S only within the pulse range $[0, r]$, and hence their contribution to number of active pulses in S divisible by d is at most $\log n$. Next, we turn our attention to the number of message entering B_0 from outside. We shall prove that this quantity is small w.h.p.

Consider the set of balls $\{B_i\}$ centered at x, where the radius of B_i is $2ir$, for $1 \leq i \leq \log n$. We concentrate on the balls B_{2i} (the same reasoning can be applied for the balls B_{2i-1}). Let C_i denote the ball whose radius is $4ir$, i,e., B_{2i}. Notice that messages traverse shortest paths, and hence, the messages which originated in $C_i \setminus C_{i-1}$ may incur only $O(\log n)$ active pulses in S which are divisible by d. Note also that by the triangle inequality, we have that if the messages sent by a node $u \in C_i$ are dominated by some node in C_{i-1}, then no message from u will enter B_0.

For each $1 \leq i \leq \log n$, let Y_i be a Bernouli random variable defined by $Y_i = 1$ iff there exists $u \in C_i \setminus C_{i-1}$ such that u is not dominated by any node in C_{i-1}. Notice that Y_i's are mutually independent. Next, we prove the following crucial property.

Claim 4.3 Let $R_i = |C_i|/|C_{i-1}|$. Then

$$\Pr[Y_i = 1] \leq \min\{\ln n \cdot (R_i - 1), 1\} .$$

Proof Sketch: By definition, $\Pr[Y_i = 1]$ is bounded by $(|C_i| - |C_{i-1}|)$ times the probability that a fixed point in C_i is not dominated by any point in C_{i-1}. The probability that a given (ID_u, r_u) pair is not dominated can be bounded as follows. List R_v for all nodes $v \in C_{i-1}$, sorted in decreasing order of *ID*s, and insert R_v in the position determined by ID_u. Let X_j be a Bernouli random variable whose value is 1 iff R_u is not dominated, given that ID_u is ranked j in the list. Since the *ID*s are i.i.d., we have that $\Pr[X_j = 1] = 1/j$, and that the probability of ID_u being ranked j in the list is $\frac{1}{|C_{i-1}|+1}$. Hence, the probability that R_u is not dominated by any node in C_{i-1} is

$$\sum_{j=1}^{|C_{i-1}|+1} X_j = \sum_{j=1}^{|C_{i-1}|+1} \frac{1}{j} \cdot \frac{1}{|C_{i-1}|+1} \leq \frac{\ln n}{|C_{i-1}|} ,$$

and therefore

$$\Pr[Y_i = 1] \leq (|C_i| - |C_{i-1}|) \cdot \frac{\ln n}{|C_{i-1}|} = (R_i - 1) \ln n .$$

∎

Corollary 4.4 $E[\sum_i Y_i] \leq \ln^2 n$.

Proof: By the claim,

$$E[\sum_i Y_i] \leq \ln n \sum_i \min\{R_i - 1, 1\} .$$

Now, for all $x \geq 1$, $\min\{x - 1, 1\} \leq \ln x$, and hence

$$\sum_i \min\{R_i - 1, 1\} \leq \sum_i \ln R_i = \ln(\prod_i R_i) .$$

Finally, since $\prod_i R_i \leq n$, we get $E[\sum_i (Y_i)] \leq \ln^2 n$, as desired. ∎

To complete the proof of Lemma 4.2, we apply the Chernoff bound (Raghavan-Spenser variant: [24], Theorem 1) and obtain $\Pr[\sum_i Y_i \geq \log^3 n] \leq \frac{1}{n^3}$. Repeating the argument for the odd balls, we conclude that with high probability, only $O(\log^3 n)$ of the $C_i \setminus C_{i-1}$ sets send messages that reach S, and since each such set incurs only $O(\log n)$ pulses divisible by d in S, we are done. ∎

Lemma 4.2 completes the proof of Theorem 1.2.

5 Local rollback technique

In this section we explain the *local rollback technique*, the technique that enables us to upgrade an arbitrary static asynchronous protocol to a dynamic protocol. The basic idea was used in [11] for synchronous protocols, and here we generalize the construction to the general asynchronous model.

We begin with the well known notion of causality chain [18].

Definition 5.1 Given an execution of an asynchronous protocol $\langle a_1, a_2, a_3, \ldots \rangle$, where a_i is either a send or receive action for all $i \geq 1$, we say that a_i is *causally dependent* on a_j if one of the following holds.

1. a_j is a receive action, a_i is a send action, both a_i and a_j are taken at the same node, and $i > j$.

2. There exists e message such that a_j is its send action and a_i is its receive action.

3. There exists k such that a_i is causally dependent on a_k, and a_k is causally dependent on a_i.

A sequence of actions $\langle a_{i_1}, a_{i_2}, \ldots \rangle$ is a *causal chain* if for all j, $a_{i_{j+1}}$ is causally dependent on a_{i_j}.

Intuitively, one may think of the causal dependence relation as the transitive closure of a directed graph, where the nodes are actions, and the arcs connect either a receive to later send actions occurring in the same node, or message send to its delivery. Clearly, this dependency graph is a DAG. Another important property, for synchronous protocols, is the following.

Lemma 5.2 Then length of the logest causal chain of a synchronous protocol is its time complexity.

It turns out that the length of the logest causal chain plays an important role in the complexity of adjusting to changing environment. For a given protocol π, let $Len(\pi)$ denote the length of the longest causal chain in any execution of π.

Theorem 5.3 Let π be a static asynchronous protocol with maximal congestion $Con(\pi)$, time complexity $Time(\pi)$, longest causal chain of length $Len(\pi)$, and space per edge $Mem(\pi)$. Then there exists a dynamic version ϕ of π such that

$$\begin{aligned} Comm(\phi) &= O(Comm(\pi)) \\ Time(\phi) &= Time(\pi) + Len(\pi) \\ Mem(\phi) &= Mem(\pi) + Con(\pi) \end{aligned}$$

The proof of the theorem consists of the local rollback algorithm. The essence of the technique is as follows. Throughout the execution of the algorithm, each node maintains a complete journal of the local history, i.e., an ordered list of all messages received and sent, including origins and destinations, respectively.

When a topology change occurs, the protocol "undoes" all actions that are causally dependent on this change. Specifically, both endpoints of the crashed/recovered edge send messages to all the nodes they sent messages before, saying "roll back my messages". When a node receives such a message, it looks through its journal, and resets itself to the state in which the first obsolete message was received. In addition, it looks in its journal to find to which nodes did it send messages that are now obsolete, and informs them. It is easy to see that in this way all "infected" actions are undone, and the computation can resume as if the initial state was correct.

However, implementing the local rollback mechanism in a straightforward way may have catastrophic consequences. Consider the case where after receiving a rollback message, the node resets itself and resumes operation (i.e., starts sending messages) immediately. Then there is a schedule that may cause an exponential blowup in the number of messages after the last topological change, as illustrated by the following scenario.

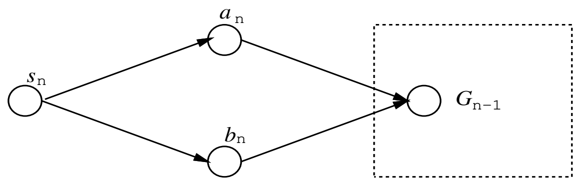

Figure 5: The Graph G_n.

Consider the following series of schedules.

1. G_0 is a single edge (s_0, t) that corresponds to a single message delivery from s_0 to t_0.

2. G_{n+1} has four additional message deliveries (see Figure 5). Messages are sent from s_n to a_n and b_n, and are delivered at s_{n-1} first from a_n and then from b_n.

Suppose that s_n initiates a rollback, and that the rollback messages are delivered to the source of s_{n-1} in reverse order, i.e., first from b_n, and then from a_n.

It is easy to verify, by induction on n, that the number of messages sent in (s_0, t) is 2^n.

The remedy for this flaw is ensuring that the nodes resume regular processing only after all rollback messages have been delivered. For this we use "3 phase rollback", similar to the technique in [11], as follows. The rollback messages are tagged with the accumulated distance from the origin of the topology change. When a node receives a rollback message, it resets its journal accordingly, marks the source of the message, and propagates the information further. A node sends an acknowledgement rollback only if its journal is already in a correct state, and after all the nodes to which a rolloback message was sent have replied with acknowledgement. This way, the rollback messages propagate in a tree-like fashion, and the acknowledgements are returned starting from the leaves. When a node receives another rollback message, it either acknowledges it immediately (if its origin is not closer than the previously seen message) or it modifies its parent pointer to the last message, and sends immediately an acknowledgement to the previous parent (if the origin of the new message is closer than the previously seen message). When the initiator receives all the ack's, it broadcasts a RELEASE message on the tree. When a node receives a RELEASE message from the node marked as the parent of the rollback, it broadcasts RELEASE and goes back to normal mode of operation, starting at the state specified by its current journal.

The protocol outlined above has the important property that after the last topological change, all rollback messages are delivered before any node resumes normal operation, hence every message is undone at most once, and the result follows.

6 Simulating an arbitrary protocol

In this section we describe how, given a static synchronous protocol π, can one construct an equivalent version of π that can run on a dynamic asynchronous network.

By Theorem 1.2, given a static synchronous protocol π whose time complexity is τ, we can construct a synchronizer in $\tilde{O}(\tau)$ time, using only polylog space. An important property of our construction was that the *congestion* was low, and hence, as an immediate corollary of Theorem 5.3, we obtain dynamic version of the synchronizer initialization stage (i.e., the covers construction) with only polylogarithmic space requirement. Moreover, since the underlying cover-construction algorithm is synchronous, the length of the longest causality chain of the initialization phase can be shown to be $\tilde{O}(\tau)$, by proving that the length of the logest causal chain is independent of the actual asynchronous schedule. Thus, superimposing the rollback mechanism over this construction, we obtain a synchronizer that initializes itself in $\tilde{O}(\tau)$ time, after the last topological change. In other words, we have the following.

Theorem 6.1 It is possible to construct sparse covers of τ-neighborhoods in a dynamic asynchronous network, by a randomized algorithm with probability of success $1 - O(1/n)$, using $\tilde{O}(1)$ space per edge, in time $\tilde{O}(\tau)$, and communication stretch $\tilde{O}(1)$.

Nevertheless, we are not quite done yet. We now must confront the problem of topological change after the initialization stage is over, i.e., when the actual algorithm π is already running. To our dismay, the rollback technique is not helpful now: π may have high congestion (e.g., $\Omega(n)$), and there is nothing we can do about it. We must therefore conclude that the rollback technique is out of the question at this stage, if we are to keep the space requirement low. The reset procedure is not applicable directly either, since it might incur $\Omega(n)$ time overhead. This problem, namely what to do if a topological change occurs while π is running, brings us to the last twist of the algorithm, which goes as follows. The first rule to follow is that we run π in parallel on each cluster of the τ-cover, restricted to the subgraph induced by the cluster nodes. Output is produced only by nodes whose complete τ-neighborhood is contained in the cluster ("kernel nodes"). Notice that for each node there exists such a cluster, which can be identified by the *ID* of its center. Furthermore, it is easy to see, by induction on τ, that the output is correct. Now, if a topological change occurs, reset is applied to the affected clusters *over the cluster trees*. By the assumption on the running time of the algorithm, the reset needs not be propagated to other clusters, and since it is applied over the cluster trees, it will terminate in time proportional to the height of these trees, i.e., $\tilde{O}(\tau)$. The correctness follows from the fact that for each node there is its "home cluster", and if this cluster is not reset, then the node will produce correct output.

Let us finally consider the complexity cost of this idea. For the space, notice that the storage of each node is replicated the number of clusters in which it is member, i.e., $\tilde{O}(1)$. The time to perform a reset on a τ-cluster is additional $\tilde{O}(\tau)$, and the message complexity is increases ny a factor of $\tilde{O}(1)$ messages per link. With this, the proof of Theorem 1.1 is completed.

References

[1] Yehuda Afek, Baruch Awerbuch, and Eli Gafni. Applying static network protocols to dynamic networks. In *Proc. 28th IEEE Symp. on Foundations of Computer Science*, pages 358–370, October 1987.

[2] Yehuda Afek and Moty Ricklin. Sparser: A paradigm for running distributed algorithms. unpublished manuscript, 1990.

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

[4] 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.

[5] Baruch Awerbuch, Bonnie Berger, Lenore Cowen, and David Peleg. Fast deterministic cover algorthms. Unpublished manuscript, November 1991.

[6] Baruch Awerbuch and Robert G. Gallager. Distributed BFS algorithms. In *Proc. 26th IEEE Symp. on Foundations of Computer Science*, October 1985.

[7] 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.

[8] Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Self-stabilization by local checking and correction. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, pages 268–277, October 1991.

[9] Baruch Awerbuch and David Peleg. Network synchronization with polylogarithmic overhead. In *Proc. 31st IEEE Symp. on Foundations of Computer Science*, pages 514–522, 1990.

[10] Baruch Awerbuch and David Peleg. Sparse partitions. In *Proc. 31st IEEE Symp. on Foundations of Computer Science*, pages 503–513, 1990.

[11] Baruch Awerbuch and Michael Sipser. Dynamic networks are as fast as static networks. In *Proc. 29th IEEE Symp. on Foundations of Computer Science*, pages 206–220, October 1988.

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

[13] 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.

[14] 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.

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

[16] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Trans. on Software Eng.*, SE-13(1):23–31, January 1987.

[17] 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.

[18] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. of the ACM*, 21(7):558–565, July 1978.

[19] T. Leighton, B. Maggs, and S. Rao. Universal packet routing algorithms. In *Proc. 29th IEEE Symp. on Foundations of Computer Science*, pages 256–271. IEEE, October 1988.

[20] Tom Leighton. *Introduction to Parallel Algorithms and Architectures*. Morgan-Kaufman, 1991.

[21] N. Linial and M. Saks. Decomposing graphs into regions of small diameter. In *Proc. 2nd ACM-SIAM Symp. on Discrete Algorithms*, pages 320–330. ACM/SIAM, January 1991.

[22] D. Peleg and E. Upfal. A tradeoff between size and efficiency for routing tables. *J. of the ACM*, 36:510–530, 1989.

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

[24] Prabhakar Raghavan. Probabilistic construction of deterministic algorithms: approximating packing integer programs. In *Proc. 27th IEEE Symp. on Foundations of Computer Science*, pages 10–18, May 1986.

[25] 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.