Dynamic Networks are as fast as static networks (Preliminary Version)

Baruch Awerbuch* Michael Sipser†

Department of Mathematics and
Laboratory for Computer Science,
MIT, Cambridge, MA 02139

Abstract

This paper gives an efficient simulation to show that dynamic networks are as fast as static ones up to a constant multiplicative factor. That is, any task can be performed in a dynamic asynchronous network essentially as fast as in a static synchronous network. The simulation protocol is based on a new approach, perceiving "locality" as the key to fast adaptation to changes in network topology. The heart of our simulation is a new technique, called a *dynamic synchronizer* which achieves "local" simulation of a global "clock" in a dynamic asynchronous network. Using this result we obtain improved solutions to a number of well known problems on dynamic networks. It can also be used to improve the solution to certain static network problems.

1 Introduction

The *Dynamic Asynchronous* network, where links may repeatedly fail and recover, is a realistic model of existing commercial communication networks, such the ARPANET [23]. Design and analysis of protocols for such networks is much more complicated than the corresponding tasks in a *Static Synchronous* network. Unfortunately, the latter model is not a realistic model of a communication network.

Dynamic networks have long been thought to be a much less efficient computational environment then static networks. This common belief has been due to the existence of a large class of problems which could be solved much faster in dynamic networks than in static networks.

This paper dispels this myth, showing that, in fact, any task can be performed in a dynamic asynchronous network as fast as in a static synchronous network. We show this by presenting a simulation of the latter model by the former one requiring only $O(1)$ time overhead. Previous methods required $\Omega(V)$ time, where V denotes the number of nodes in the network [1] [4]. Our result allows us to give sharply improved bounds to well known problems on dynamic networks and even yields improved solutions to certain static network problems where the input to the problem changes dynamically.

The novel simulation protocol is based on a new approach, perceiving "locality" as the key to fast adaptation to changes in network topology. This localization of operations will also prove efficient in handling, for the first time in the literature, situations in which the network is forever unstable. It will also allow to improve the complexity of certain algorithms in *static* networks.

The heart of our simulation is a new technique, called a DYNAMIC SYNCHRONIZER which achieves "local" simulation of a global "clock" in a dynamic asynchronous network.

Our contributions can be summarized as follows:

1. Introducing the notion of "strong-sense" simulation of a static synchronous network by a dynamic asynchronous network.

*Supported by Air Force contract TNDGAFOSR-86-0078, ARO contract DAAL03-86-K-0171 and NSF contract CCR8611442

†Supported by Air Force contract TNDGAFOSR-86-0078, and NSF grant DCR 86-0078.

2. Introducing the notion of dynamic synchronizer.
3. Developing a dynamic synchronizer with $O(1)$ time overhead.
4. Proving that our dynamic synchronizer with $O(1)$ time overhead yields the "strong-sense" simulation of a static synchronous network by dynamic asynchronous network with $O(1)$ time overhead.

The "strong-sense" simulation introduced in this paper is the ultimate goal of dynamic asynchronous protocol design, as it enables us to transform the protocol from the weakest model into a protocol for the hardest model, in a strong sense. This problem has never been considered in the past, to the best of our knowledge. The relevant existing works focused on a weaker notion that we call a "weak-sense" simulation [1][4].

Our results strongly improve solutions for classical problems in distributed computing and communication networking, i.e. TOPOLOGY UPDATE [11], RESOURCE ALLOCATION, [7, 22], etc. Essentially, we can improve solutions of problems which allow local solution in a static synchronous network. It is worth mentioning that those improvements are obtained *automatically*, i.e. we *do not improve the original static solutions*, but rather treat it as a black box; the improvement is solely due to the more efficient simulation.

Our synchronizer possesses a number of additional features of practical and theoretical interest. A virtue from the theoretical standpoint is that our simulation does not employ unbounded counters. In fact, we do not see any way to use unbounded counters to improve upon our result. On the practical side, the code for our synchronizer is very concise and easy to implement. Finally, the disadvantage that we see with this method is that it may greatly increase the communication and space requirements of the protocol. Nevertheless, we feel there are practical situations where one may be willing to pay this price.

To aid the reader we present two dynamic synchronizers, a simple one called ε and a more complicated one called ζ. These both work by notifying all nearby nodes whenever a fault occurs. The simpler one uses a naive broadcast scheme which sends notification messages along all paths in the affected neighborhood. This may generate an exponential amount of communication. The more complicated synchronizer ζ avoids this by only notifying along the shortest path to an affected node. This keeps the total additional communication polynomial in the size of the affected neighborhood. The techniques used in the construction of shortest paths are similar to the ones in [6].

The rest of this paper is structured as follows. We start in Section 2 by explaining the difference between dynamic and static synchronizers. Then, in Section 3 we show the relationship between our results and the "weak-sense" simulations, previously considered in the literature. In Section 4 we show applications of our results to particular network problems. In the following sections, we describe our technical contributions. Section 5 contains the definition of the notion of "strong-sense" simulation. In section 6 we state the definition of dynamic synchronizer. Section 7 shows how to transform a dynamic synchronizer into efficient Reset protocol. In sections 8 and 9 we describe the dynamic synchronizers ε and ζ, respectively.

2 Relation to static synchronizers

We would like to point out that the static synchronizer α of [3] will not work in a dynamic environment. One reason for this is that α cannot run as is in a network where edges may recover, since the endpoints of an edge that recovers can be far off from each other in their pulse numbers [3], namely, up to $V-1$. A second, more subtle problem, is that, even in static networks, the worst-case time in between consecutive pulses of α in [3] is $\Theta(V)$. This phenomenon has a disastrous effect on time performance of a (local) protocol. In a static setting, one can avoid it by assuming that initially all pulse numbers to 0. In a dynamic setting, one can achieve the same effect by *global reset* of all pulse numbers to 0;

however this "reset" operation appears to be inherently *global*, i.e. requires $\Omega(V)$ time.

The natural question here is whether a such a "global reset" can be performed *locally*, say, in $O(1)$ time. Observe that, no matter how local is the underlying computation, we cannot simulate it fast if we have separate clocks for each one of the $O(1)$-neighborhoods in the network, since this would have created congestion in the network.

3 Relationship to Reset Protocols

3.1 Definition of Reset

The essence of "computing" in a dynamic network is that we view network topology as a part of the input to the dynamic network protocol, i.e. changes in topology are represented as changes in the input. A dynamic protocol has dynamic input and consequently, dynamic output, which must "adapt" as the input changes. (For more details, see [1].)

Informally, given a protocol π which achieves a certain task (e.g. finds shortest paths) in a static synchronous network, we say that a dynamic protocol ϕ is an *adaptive version* of π in the *weak sense*, if whenever ϕ operates in a dynamic network that eventually stabilizes, then the output produced by ϕ will eventually stabilize to an output that π would have produced if π were executed in the final network.

An *Asynchronous (Synchronous) Reset* protocol, receives as an input, an arbitrary asynchronous (synchronous) protocol π, and produces as its output, a weak-sense adaptive version ϕ of π, that can run in a dynamic asynchronous network. Any *Synchronous Reset* is also an *Asynchronous Reset*, since any asynchronous protocol can run on a synchronous network. On the other hand, the conjunction of an *Asynchronous Reset* with a *Synchronizer* [3] yields a *Synchronous Reset*.

3.2 The complexity measures

For an arbitrary static synchronous protocol π, denote the communication and time complexities as c_π and t_π and the number of nodes participating in the protocol by v_π.

Notation 3.1 We consider an arbitrary static *synchronous* protocol π. Consider the following upper bounds on an arbitrary execution of that protocol.

c_π: on the number of messages sent

t_π: the time of π, in terms of *number of pulses*.

v_π: The number of nodes participating in the protocol.

Big-scale protocols are the protocols which involve the whole network, e.g. $v_\pi \approx V$. Nevertheless, big-scale protocols often possess strong "locality" properties, namely have $t_\pi \ll V$. There are many important static synchronous protocols that fall in this category; the most familiar examples are [21],[2],[18]. For example, $t_\pi = \log^* V$ for [2]. In contrast, *Small-scale* protocols are protocols with relatively few nodes are participating in the protocol, i.e. $t_\pi \approx v_\pi \ll V$.

For the execution of a protocol in an asynchronous network, the *longest message delay* in that execution is the maximum difference between the arrival and transmission times of a message, in terms of some global clock (which is not accessible to nodes). Following [13, 15, 1, 9], the *normalized* time between two events in an execution is the ratio between the physical time between those two events in terms of the global clock above, and the longest message delay in that execution, i.e. physical time in case that link delays vary between 0 and 1. In the context of asynchronous network, "time" always means "normalized time". In the final paper, we extend this notion of "normalized time" in the spirit of [13] to capture "congestion" on links caused by almost simultaneous transmission of multiple messages over the link.

Notation 3.2 For a dynamic protocol ϕ, denote

Method	Commun.	Time
[3, 1]	$c_\pi + E \cdot t_\pi$	$V + t_\pi$
This paper	$c_\pi + E \cdot t_\pi$	t_π

Figure 1: Applications to Synchronous Resets

Method	Commun.	Time
[1]	$E + c_\pi$	$V + t_\pi$
[1]	$c_\pi + v_\pi \log^2 v_\pi$	$V + v_\pi \log^2 v_\pi$
[4]	$c_\pi + v_\pi \log^2 v_\pi$	$v_\pi \log^2 v_\pi$
This paper	$c_\pi + E \cdot t_\pi$	t_π

Figure 2: Applications to Asynchronous Resets

$\bar{T}_\phi$: the *Quiescence Time* [1]. This is the (normalized) time it takes ϕ to assimilate fully to a topology change, i.e. time from the last topological change until last change in the output of ϕ.

$\bar{C}_\phi$: the *Amortized Communication* [1]. This is is the number of messages sent by ϕ per topological change.

Although our methods will work for any protocol, its advantages are most evident for big-scale synchronous protocols with strong locality properties. In this paper, our goal is to achieve a resulting dynamic protocol with $O(t_\pi)$ time and $O(poly(E))$ communication. as opposed to solutions with time $\Theta(poly(v_\pi))$ since this would be far less efficient for such big-scale protocols.

3.3 Applications of our method to Reset protocols

A possible approach to obtain a *Synchronous Reset* is simply to combine the fastest synchronizer [3], with the fastest *Asynchronous Reset* of [1]. Our results dominate this, both in time and in communication, and both for small-scale and big-scale protocols. See Figure 1.

The design of *Asynchronous Reset* protocols is one of the most well-studied problems in the area [14, 10, 24, 11, 12, 1, 4], and has been around for at least 10 years. This problem is *easier* than the one here of designing a Synchronous Reset, because there we already start with protocols resilient to asynchronicity, and thus do not need a clock.

Nevertheless, the fastest Asynchronous Resets still require $\Omega(V)$ time. The reason for this is that all existing methods [14, 10, 24, 11, 12, 1, 4] use a brute-force "blast-away" technique, introduced by Finn [10]. The essence of this technique is that after each topological change, the previous computations are *destroyed* and a new version of the protocol is started. Since the destruction process starts at one place and must traverse the whole network, any locality properties of the original protocol are destroyed. As a result, the *quiescence time* of the resultant transformed protocol may be much greater than the running time t_π of the original protocol π if $t_\pi \ll v_\pi$. It is worth pointing out that it was conjectured in [12] and subsequently in [1], that in any Reset protocol employing only bounded-counters, "... in general, at least $\Omega(v_\pi)$ time is required from the last topological change to the restart of the static application algorithm".

In contrast, our method allows us to solve the even harder problem, the Strong-Sense Synchronous Reset, where the applications algorithm not only *starts*, but also *ends* in $O(t_\pi)$ time since the last topological change. This demonstrates that the aforementioned conjecture does not hold for protocols with more sophisticated control structure than the one in [12],[1],[4]. Our improvements are summarized in the following Figure 2.

The major reason for this improvement in time is due to the fact that we abandon Finn's blast-away method, where previous computations are destroyed. Instead, we *take advantage* of previous computations in order to speed up the response to a topological change.

On the other hand a disadvantage of our method is that its communication overhead is way too high for the case of small-scale asynchronous protocols. For such protocols, our solution is uncomparable with [1, 4], as it is superior in time but inferior in communication.

4 Applications to particular network problems

4.1 Symmetry-breaking problems

Consider *symmetry breaking* problems such as the *Maximal Independent Set* [21], $\Delta + 1$ *Coloring in Sparse graphs* [2], or Δ^2 *Coloring in general graphs* [18]. Many basic network protocols, e.g. *Call Establish* in circuit switching networks, *Resource Allocation* in distributed operating systems, *Deadlock Resolution* in distributed databases, *Flow Control* in packet-switching networks, etc., employ symmetry-breaking in one form or another. For example, the *Drinking Philosophers problem,* [7, 22] which is a variation of the *Resource Allocation problem*, is essentially $\Delta + 1$ coloring in a dynamic graph.

Symmetry-breaking can be solved very efficiently in a static synchronous network. Using previously known methods, the best known solutions for dynamic asynchronous networks yield very inefficient algorithms, because of the need to establish a synchronization mechanism, e.g. [7, 22]. Even if networks are static, but the input graph is dynamic, it is not obvious how to solve the problem. Readers interested in technical difficulties arising, say, with "purely asynchronous" independent set algorithms, are referred to [16, 19, 20]. Our improvements are summarized in Figure 3.

4.2 Topology Update and Shortest Paths

We consider well-studied problems of *Topology Update*, and all-pairs *Shortest Paths*, which are the heart of routing protocols in existing commercial networks, e.g. the ARPANET [23]. We consider the variations of those problems in which a node only needs to know the topology (shortest paths) in a t_π-neighborhood around that node. Let us consider here what we call *best case* performance, where topological changes occur rarely, compared with *worst case* performance where topological changes are frequent.

It appears that [11] contains two solutions, a "local" solution which requires $\Theta(2^V)$ messages and $O(1)$ time, and a "global" solution which requires $O(E)$ messages and $\Theta(V)$ time, i.e. trades locality for polynomial communication. Gafni [11] points out that the "local" solution is advantageous for the best case, and the "global solution" is advantageous in the worst case. Gafni poses the following question: is there a "natural" way to achieve a solution that will get the "best of both worlds", i.e. fast recovery in the best case and polynomial communication in the worst case.

Our method enables to give positive answer to Gafni's question, thus demonstrating that "locality" and polynomial communication complexity do not have to exclude each other. See the following figure.

5 Strong-Sense Synchronous Resets

5.1 Problem Statement

Informally, given a protocol π which achieves a certain task, we say that a dynamic protocol ϕ is an *adaptive version* of π in the *strong sense*, if whenever ϕ runs in a dynamic network which may never stabilize as a whole, then the output produced by ϕ will eventually stabilize and agree with the output produced by π in the stable parts of the network.

For example, suppose the original protocol π finds the sum of degrees of all nodes at distance 10 from itself. Then, the property of ϕ is that, for each node v such that the 10-neighborhood of v (i.e. the subgraph induced by nodes at distance 10 from v) becomes stable, in the sense that no more topological changes happen in that neighborhood, then node v will eventually know the sum of degrees of all nodes in the final 10-neighborhood. Observe that it makes no sense to produce any output at v, unless the 10-neighborhood of v stabilizes, i.e. it is the *weakest possible* condition under which we can hope to produce an output at v. We now formalize this concept.

Definition 5.1 A node is said to reach *quiescence* at a certain time, if it experiences its last

Problem	Underlying Static Protocol	Resulting Dynamic Protocol	Amortized Communication	Quiescence Time
Maximum Independent Set	[21]	[1]	$E \cdot \log V$	V
		This paper	$E \cdot \log V$	$\log V$
$\Delta+1$ Coloring in sparse networks	[2]	[1]	$E \cdot \log^* V$	V
		This paper	$E \cdot \log^* V$	$\log^* V$
Δ^2 Coloring in general networks	[18]	[1]	$E \cdot \log^* V$	V
		This paper	$E \cdot \log^* V$	$\log^* V$

Figure 3: Applications of our method for symmetry breaking problems

Problem	Underlying Static Protocol	Resulting Dynamic Protocol	Worst-case Amortized Communication	Best-case Quiescence Time
Topology Update in radius t_π	[13]	[11, 1]	$E \cdot t_\pi$	V
		[25, 11]	$E \cdot 2^{t_\pi}$	t_π
		This paper	$E \cdot t_\pi$	t_π
All-pairs Shortest Paths of length t_π	[13]	[1]	$E \cdot t_\pi$	V
		This paper	$E \cdot t_\pi$	t_π

Figure 4: Applications of our method for topology update and shortest paths

topological change at that time, i.e. there are no failures or recoveries from this time and on. Nodes which reach quiescence are called *quiescent*. A link is *quiescent* if it is operating from some time on, and at least one of its end-points is a quiescent node.

Definition 5.2 The *quiescence graph* of a dynamic network is an undirected graph $G_1 = (V_1, E_1)$ where V_1 contain all the quiescent nodes, and whose links E_1 are all quiescent links between quiescent nodes.

Definition 5.3 Let *quiescence decomposition* $\mathcal{V} = \bigcup_{i=1}^{|V|} V_i$ of a dynamic graph be a set of sets $V_i \subset V$ such that

- V_1 is the set of quiescent nodes.
- $V_{i+1} = \{v | v \in V_i$ and $\forall (v,u) \in E_1, u \in V_i\}$.

An example of a quiescent decomposition of a network is given in the following Figure 5.

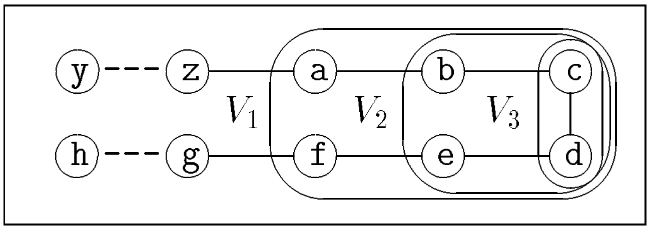

Figure 5: Dynamic network and its quiescence decomposition. Dashed lines represent links that go up and down; solid lines represent links that are up from some time and on. Here, $V_1 = \{a,b,c,d,e,f\}$, $V_2 = \{b,c,d,e\}$, $V_3 = \{c,d\}$. If $t_\pi = 3$, then nodes c, d are capable to produce meaningful output.

Definition 5.4 We say that a dynamic protocol ϕ is an *adaptive version* of π *in the strong sense* if, for *any* dynamic network with quiescence decomposition $\mathcal{V} = \bigcup_{i=0}^{|V|} V_i$, and for any node $v \in V_{t_\pi}$, running ϕ in the dynamic network yields the final output at v as running π in the *static* network $G_1 = (V_1, E_1)$.

Clearly, for arbitrary synchronous protocol π with time complexity t_π, the output of that protocol depends only on the inputs in the t_π neighborhood [18]. If depth of a node is t_π, then t_π-neighborhood of that node becomes stable, so it makes sense to talk about output of π at that node.

Any *Strong-Sense Synchronous (Asynchronous) Reset* is a (regular) *Synchronous (Asynchronous) Reset*. An attractive feature of a Strong-Sense Reset is that it preserves locality by only involving the small neighborhood of the network around a topology change when one occurs. Since it does not restart the entire network for each fault, it yields protocols which produce meaningful output under much weaker assumptions, namely that $O(t_\pi)$ neighborhood of the node stabilizes for (roughly speaking) $O(t_\pi)$ period of time. In order for a weak sense Reset to produce meaningful output, we need a very strong assumption that the *whole* network fully stabilizes for "long" ($\Omega(V)$) periods of time.

5.2 Statement of the Result

Theorem 5.5 (Main Theorem) The combination of SIMULATOR protocol of Section 7 and the DYNAMIC SYNCHRONIZER ζ of Section 9 transforms an arbitrary static protocol π into its strong-sense adaptive version ϕ, whose complexities are

$$\bar{C}_\phi = O(c_\pi + E \cdot t_\pi) \qquad (1)$$

$$\bar{T}_\phi = O(t_\pi) \qquad (2)$$

Proof: postponed to the final version. •
Comment: In practice, no fixed sub-set of a communication network ever permanently stabilizes, no matter how small it is. The practical interpretation of the above Theorem 5.5 is that if a t_π-neighborhood stabilizes for $O(t_\pi)$ time, then the correct output will be produced.

6 Dynamic Synchronizers

6.1 Problem Statement

In this section, we define the notion of a dynamic synchronizer in such a way that it can be "reset" in a "local" way and is robust to topological changes.

The dynamic synchronizer has a parameter t_π, which represents the time complexity of some synchronous protocol π. While the definition of the dynamic synchronizer seems to implicitly require knowledge t_π, in actual implementation we *do not use* this knowledge.

The dynamic synchronizer generates, as an output, a (finite) sequence of *pulses* at each node of the network. Each pulse is labeled with a positive integer not exceeding t_π. The pulse labeled with p is called *pulse p*. The *pulse number* at a node at a given time is the labeling of the last pulse generated at this node by that time, which represents the current "local time" at that node. Unlike "regular" time, "local time" is not necessarily monotonic, i.e. it may decrease as well as increase, similarly to "roll-back" mechanisms [17]. Thus, there may be many pulses at a node labeled with the same pulse number p.

Definition 6.1 A *final* pulse p at a node is a pulse labeled p such that all pulses following it are labeled by $p+1$ at least. A *final* message of pulse p on edge (u, w) is the message sent by u to w upon final pulse p at u. The final pulses and messages must satisfy the specifications stated in Figure 6.

Example of sequence of pulses satisfying the conditions above is given in the following Figure 7.

6.2 Complexity measures

The complexity measures for dynamic synchronizers are defined as follows. Consider the set of executions of the dynamic synchronizer $\mathcal{S}$ in a dynamic network, with k input changes happening during the execution.

Notation 6.2 $\hat{C}_\mathcal{S}$: is the *Amortized Communication Complexity.* This is the total number of messages sent, divided by $k \cdot t_\pi$.

Adaptivity Final input change at a node *precedes* final pulse 1.

Termination Final pulse 1 is performed by each quiescent node.

Liveness For all $1 < p \leq t_\pi$, and for each quiescent node, if node receives final messages of pulse $p-1$ from all its quiescence neighbors, then it generates final pulse p.

Safety For all $1 \leq p \leq t_\pi$, arrival of a final message of pulse p at a node *precedes* final pulse $p+1$ at that node.

Figure 6: Specifications of a dynamic synchronizer

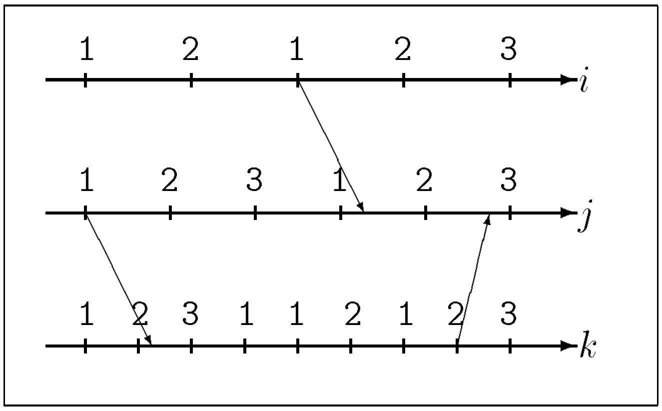

Figure 7: Timing diagram describing pulse sequences at nodes i, j, k produced by a dynamic synchronizer. Arrows represent message sent from one node to another. The last three pulses at each node look "as if" generated by a static synchronizer.

$T_{init}(\mathcal{S})$: is the *Initialization Time*. This is the maximal time from the time a node reaches quiescence until it performs final pulse 1.

$T_{pulse}(\mathcal{S})$: is the *Pulse Time*. This is the maximal difference between the time that all neighbors of a node have performed final pulse $p-1$ and the time a node performs final pulse p.

$\hat{T}_{\mathcal{S}}$: is the *Amortized Time Complexity*. is defined as $\hat{T}_{\mathcal{S}} = \frac{T_{init}(\mathcal{S})}{t_\pi} + T_{pulse}(\mathcal{S}) + 1$.

Comment: That is, $\hat{T}_{\mathcal{S}}$ is the upper bound on the average time to perform a single pulse since the last input change. By the *Liveness* and *Termination* conditions in Figure 6, both $T_{pulse}(\mathcal{S}) < \infty$ and $T_{init}(\mathcal{S}) < \infty$.

Our specifications and complexity measures for dynamic synchronizers are equivalent to the specifications and complexity of the static synchronizer in [3], in the case that there are no topological changes in the network.

7 The Simulator

We introduce the notion of a *Simulator* for synchronous protocols. which transforms *any Dynamic Synchronizer* into *Synchronous Reset*. The *Simulator* receives, as in *input*, the *output* of *Dynamic Synchronizer*, namely sequences of pulses at all the nodes, and a synchronous protocol π. The simulator "attempts" to run π w.r.t. this sequence of pulses, even though it is not really an output of a synchronous clock.

The Simulator operates as follows. Each node maintains a "local state', denoted by q_p. which describes the contents of all local variables at the node after pulse p. It also maintains the incoming (outcoming) message buffers $I_p(e)$ ($O_p(e)$), for each edge e and each pulse number p. We denote by I_p (O_p) the set of all $I_p(e)$ ($O_p(e)$), for all adjacent edges e. We denote by I_p (O_p) the set of all $I_p(e)$ ($O_p(e)$). We view I_p (O_p) as the input (output) of a given node at pulse p. Those buffers are initialized to *nil*.

Upon pulse $p+1$, each node performs $q_{p+1} := \Delta(q_p, I_p)$, where Δ the *state transition* function. In addition, the node sets $O_{p+1} := \lambda(q_{p+1})$, where

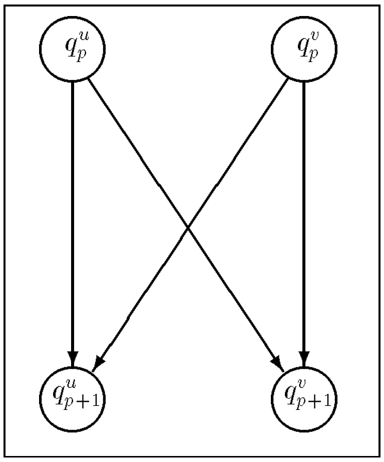

Figure 8: The state transition function computes q_{p+1} from q_p and messages received from neighbors. Arrows show dependencies of states for two adjacent nodes u, v.

λ is the *output* function. Next, for each e, node sends message M on e, where M is the contents of $O_p(e)$ buffer. Messages of each pulse are labeled with the number of that pulse. Once node on the other side of e receives this message, it inserts this message into the incoming messages buffer $I_p(e)$

Observe that the value of q_p at a node depends on the values of q_{p-1} of its neighbors, as shown in Figure 8.

The formal description of the variables, functions, and code of the protocol is given in Figures 9, 10, and 11.

Theorem 7.1 The combination of SIMULATOR protocol in Figure 11, and *any* dynamic synchronizer $\mathcal{S}$ yields a *Strong-Sense Synchronous Reset* $\mathcal{A}$. Applying Reset $\mathcal{A}$ to an arbitrary protocol π yields a dynamic version ϕ of π with complexities

$$\bar{C}_\phi = O(c_\pi + \hat{C}_{\mathcal{S}} \cdot t_\pi) \quad (3)$$

$$\bar{T}_\phi = O(t_\pi \cdot \hat{T}_{\mathcal{S}}) \quad (4)$$

Proof: will appear in full paper. •

Intuitively, the SIMULATOR protocol above simulates the circuit in Figure 12. This circuit corresponds to Figure 5.

`Adjacent`: the set of adjacent edges. May change dynamically with time.

$I_p(e)$: Last message of pulse p received on edge e. Initially, $I_p(e) = nil$ for all e and p.

I_p: The contents of all $I_p(e)$.

$O_p(e)$: Message to be transmitted at pulse p on edge e.

O_p: The contents of all $O_p(e)$.

q_p: local state of the node after pulse p.

Figure 9: Variables of the Simulator

Δ computes the local state in the next pulse from the local state of the present pulse and the messages sent by the neighbors in the current pulse.

λ computes the messages to be sent in the state.

Figure 10: Functions of the static protocol

Upon generation of pulse p:
  $q_p := \Delta(q_{p-1}, I_{p-1})$
  $O_p := \lambda(q_p)$
  for each e $\in$ Adjacent, **send** $O_p(e)$ on e

For message M of pulse p arriving on edge e
  $I_p(e) :=$ M

Figure 11: Normal Form

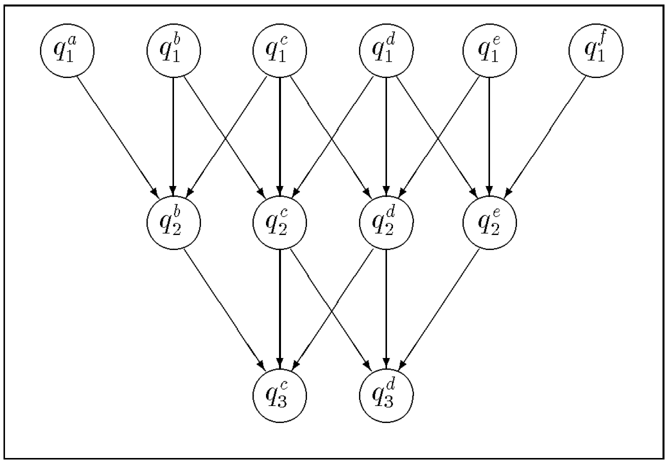

Figure 12: The circuit simulated by the protocol. The level i of the circuit contains the nodes which perform "correctly" pulse i; q_p^i denotes final pulse p at node i. The top and the bottom of the circuit represent the input and the output, respectively.

8 Naive Dynamic Synchronizer

A Dynamic synchronizer with $O(1)$ time can be easily achieved by a "naive" protocol, in which every node tries to raise its pulse number as high as possible, subject to the following two requirements:

1. The pulse number of a node can be increased to p only after all neighbors increase their pulse number to $p-1$.

2. pulse number always drops to 0 after each input change

Observe that pulse drops must be performed to satisfy the *Liveness* condition. To implement 1), consider the following procedure, which will be referred to as the STATIC SYNCHRONIZER α; it is essentially identical to synchronizer α of [3]. The essence of this procedure is that the pulse number of a node is increased to p after all neighbors increase their pulse number to $p-1$. Note that knowledge of t_π is not needed.

The code of the procedure is given below. In order for the Static Synchronizer α to be operational, the variable `State` in this synchronizer should be initialized to *growing*. (The value of

`Adjacent:` the set of adjacent links

`State:` flag receiving values *dropping,growing*. The Synchronizer is operational only while `State = growing`.

`Pulse:` Highest pulse which was performed correctly until now.

`Pulse(e):` Estimate on `Pulse` of the node on the other end of edge e.

`PULSE(p)` message carrying the parameter `Pulse` of the sender.

NEW-PULSE procedure that might start a new pulse.

Figure 13: Static synchronizer α: node variables and procedures

`State` is not changed by the static synchronizer itself.) The reason for introducing this variable into the code of the protocol is that in the future we will use Synchronizer α in the setting in which the value of `State` will be changed by some external protocol.

To implement 2), we add on top of the static synchronizer α a mechanism for dropping pulse numbers; this mechanism will be called *Manager* ε. Basically, it forces every node to drop its pulse number to 0 upon every input change, and report the drop in the pulse number to all neighbors. Upon receipt of this report, a neighbor with pulse number higher than 1 will drop its pulse number to 1, and will report this fact to all neighbors. When the latter report arrives at a node with pulse number higher than 2, it drops it pulse to 2, etc. After each pulse drop, a node executes static synchronizer α again, trying to increase its pulse number.

The code of Manager ε is presented in Figure 15. The combination of Manager ε together with static synchronizer α will be referred to as *Dynamic Synchronizer* ε.

Theorem 8.1 The dynamic synchronizer ε indeed satisfies the *Safety, Adaptivity, Termination,* and *Liveness* properties of a dynamic synchronizer. Its time complexity is $\hat{T}_\varepsilon = O(1)$.

Proof: omitted. •
Comment: We do not claim any bounds on the communication complexity $\hat{C}_\varepsilon$. In the following

section we build a synchronizer whose communication complexity is provably small (linear).

```
Upon invocation:
   Execute Procedure NEW-PULSE

Procedure NEW-PULSE
   while for all e ∈ Adjacent,
   Pulse(e) > Pulse −1 do
      Pulse := Pulse + 1
      Call normalized protocol π
      for all e in Adjacent,
         send PULSE(Pulse) on e

For PULSE(p) on edge e
   Pulse(e) := p
   if State = growing then
      Execute Procedure NEW-PULSE
```

Figure 14: Static synchronizer α: node algorithm

```
For change in node input,
   case of recovery on a link e
      add e to Adjacent
   case of failure on a link e
      delete e from Adjacent
   Pulse := 0
   for all links e ∈ Adjacent
      send DOWN(0) on e
   Invoke STATIC SYNCHRONIZER α

For DOWN(p) on edge e
   Pulse(e) := p
   if p+1 < Pulse then
      Pulse := p+1
      for all links e ∈ Adjacent
      send DOWN(p+1) on e
      Invoke STATIC SYNCHRONIZER α
```

Figure 15: Manager ε

9 Final dynamic synchronizer

9.1 Outline

The Final dynamic synchronizer, referred to as SYNCHRONIZER ζ, uses two subroutines. The first subroutine is the STATIC SYNCHRONIZER α, presented in the previous section. The second subroutine, called MANAGER ζ, can be viewed as a modification of Manager ε. The task of the Manager ζ is to suspend static synchronizer α, delete all outdated information by dropping the value of *Pulse* and then to resume static synchronizer α, which will cause *Pulse* to increase.

It is essential for the communication complexity of the resulting protocol that, roughly speaking, all the pulse drops are performed *before* all pulse increases. In that respect, our strategy resembles the one used in [1]. The implementation itself is different and is in fact based on the method of Spinelli and Gallager [25].

For simplicity, suppose first all the changes are performed "sequentially", i.e. long periods of time separate any two changes. A node which notices a change, originates a special "shock wave", which propagates thru the t_π-neighborhood of the origin of the wave, Every node at distance $1 \le d \le t_\pi$ of the origin will drop its pulse number to d, since all pulses from d until t_π have been performed with outdated information. Once the wave reaches the boundary of the t_π-neighborhood, it "reflects" back, as in [8]. Once all the "reflections" of the wave have been collected at the origin, it is clear that all the pulse numbers have been dropped. At this time, the origin starts propagating the "secondary" wave, which informs all the nodes in the t_π-neighborhood that they can start executing synchronizer α, so that the pulse numbers will start increasing. In order to determine the boundary of the t_π-neighborhood and the precise distance from the origin inside the t_π-neighborhood, we execute Breadth-First-Search (BFS) w.r.t. the origin. In the actual implementation, nodes do not need to know the value of t_π.

Now, if many changes happen in parallel, then each originator "conquers" all the nodes which are closer to it than to some other originator, effectively performing a Breadth First Search with respect to all the originators. During this process, one originator whose tree is growing "fast" may "conquer" nodes which should belong in fact to other originator; however the "true owner" will eventually retaliate, reconquering its nodes. Efficient, but complicated "conquering strategies" have been proposed in [5, 6]. The technique in this paper protocol is a distributed version of Bellman-Ford algorithm, in which nodes update their distances in a "greedy" manner. We additionally impose a mechanism of acknowledgements ("reflections") on top of this algorithm, so that origin can detect the time at which its tree stops growing.

9.2 Detailed Description

Every node keeps the following variables. `Pulse` describes number of the last pulse performed correctly. `In-edge` points to a neighbor which caused the most recent drop in the pulse number; the set of these pointers forms a forest of directed trees, rooted at the nodes which observed input changes. Each such tree will be called a *drop tree*. The variable `State` receives values *dropping, growing*; by changing this variable the MANAGER protocol resumes and suspends the STATIC SYNCHRONIZER α.

Once a node with `Pulse` > 0 notices a change in the input, it drops its `Pulse` to 0, sets `In-edge` to *self* and sets its `State` to *dropping*, thus suspending the operation of STATIC SYNCHRONIZER. Next, it executes procedure BROADCAST-DROP. In that procedure the node broadcasts a special `DROP(Pulse)` message over each adjacent edge e, setting `Ack(e)` := *false*.

Once a node receives a `DROP(p)` message over edge e, it acts as follows. If `Pulse` $\leq p+1$, then it sends back `ACK(p+1)` message. Otherwise, it sets `Pulse` := $p+1$, `In-edge` := e, `State` := *dropping* and executes procedure BROADCAST-DROP.

The meaning of the `ACK` message is that the subtree of the drop tree rooted at that node has stopped growing. Observe that the node may send many `DROP` messages while `State` = *dropping*; however the pulse numbers in those messages are strictly decreasing. Thus, `ACK(p)` messages arriving at the node with $p >$ `Pulse` correspond to previous `DROP` message. Once an acknowledgement `ACK(p)` arrives on edge e, the node acts as follows. If $p \neq$ `Pulse`, then this `ACK` is outdated and thus is discarded. Otherwise, if p =`Pulse`, this is the `ACK` to the latest `DROP` message sent by the node. In the latter case, the node sets `Ack(e)` := *true* and executes procedure CONVERGE.

In procedure CONVERGE, the node checks whether `Ack(e)` = *true* for each adjacent link e. If so, the node will send `ACK(Pulse-1)` message on `In-edge`. Once the root of a drop tree detects that `Ack(e)` = *true* for all incident edges e, it decides that the "cleaning" process is over, and now is the time to resume the operation of synchronizer α.

For that purpose, it executes the procedure BROADCAST-GROW. In the procedure BROADCAST-GROW, the node sends `GROW` message to all neighbors, and sets `State` := *growing*, thus resuming the operation of SYNCHRONIZER α. Once a node receives `GROW` message on its `In-edge`, it executes procedure BROADCAST-GROW.

The Manager ζ protocol is formally presented below in Figures 16, 17. Figure 16 contains the declarations of the variables, messages and procedures used, and Figure 17 describes the node algorithm.

9.3 Properties of dynamic synchronizer ζ

Theorem 9.1 The dynamic synchronizer ζ above indeed satisfies the *Safety, Adaptivity, Termination,* and *Liveness* properties of a dynamic synchronizer. Its complexities in communication and time are

$$\hat{C}_\zeta = O(E) \tag{5}$$

$$\hat{T}_\zeta = O(1) \tag{6}$$

Proof: will appear in the final version. •

Observe that the Theorem 9.1 above and the Theorem 7.1 of Section 7 imply the Theorem 5.5 of Section 5.

10 Acknowledgements

It is a pleasure to thank friends and colleagues for their generous contributions to this research. Gary Miller was the first one to suggest studying synchronous protocols in dynamic networks. Discussions with Michelangelo Grigni and Johan Hastad were extremely helpful in the early stages of this research. The author thanks Oded Goldreich, Hershel Safer and Nir Shavit for contributing to the presentation in this manuscript. Thanks are also due to Serge Plotkin for invaluable help with formatting thsi paper.

References

[1] Yehuda Afek, Baruch Awerbuch, and Eli Gafni. Applying static network protocols to dynamic networks. In *28th Annual Symposium on Foundations of Computer Science*, IEEE, October 1987.

[2] S. Plotkin Andrew V. Goldberg and G. Shannon. Parallel symmetry breaking in sparse graphs. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, page , ACM, May 1987.

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

[4] Baruch Awerbuch. On the effects of feedback in dynamic network protocols. In *29th Annual Symposium on Foundations of Computer Science*, IEEE, October 1988.

[5] Baruch Awerbuch and Robert G. Gallager. Distributed bfs algorithms. In *25th Annual Symposium on Foundations of Computer Science*, IEEE, October 1985.

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

[7] K. Mani Chandy and J. Misra. The drinking philosophers problem. *ACM Trans. Prog. Lang. and Syst.*, 6(4):632–646, 1984.

[8] Edsger W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Info. Proc. Lett.*, 11(1):1–4, August 1980.

[9] Feldman and Micali. Optimal algorithms for byzantine agreement. In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing*, May 1988.

[10] Steven G. Finn. Resynch procedures and a fail-safe network protocol. *IEEE Trans. Comm.*, COM-27(6):840–845, June 1979.

[11] Eli Gafni. Topology resynchronization: a new paradigm for fault tolerance in distributed algorithms. In *Proceedings of the Amsterdam Workshop on Distributed Algorithms*, page , CWI, July 1987.

[12] Eli Gafni and Yehuda Afek. Local fail-safe resynch procedure. April 1987. unpublished manuscript.

[13] Robert G. Gallager. *Distributed Minimum Hop Algorithms.* Technical Report LIDS-P-1175, MIT Lab. for Information and Decision Systems, January 1982.

[14] Robert G. Gallager. *A Shortest path Routing Algorithm with Automatic Resynch.* Technical Report , MIT Lab. for Information and Decision Systems, March 1976.

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

[16] G.H. Godbeer. The computational complexity of the stable configuration problem for connectionist models. September 1987. M.Sc. thesis.

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

[18] Nathan Linial. Locality as an obstacle to distributed computing. In *27th Annual Symposium on Foundations of Computer Science*, page , IEEE, October 1987.

[19] J. Lipscomb. On the computational complexity of finding a connectionist model's stable state vectors. October 1987. M.Sc. thesis.

[20] M. Luby. Unpublished. January 1988. private communication.

[21] Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.*, 15(4):1036–1053, November 1986.

[22] J. Lundelius and N. Lynch. Synthesis of efficient drinking philosophers algorithms. January 1988. unpublished manuscript.

[23] John M. McQuillan, Ira Richer, and Eric C. Rosen. The new routing algorithm for the ARPANET. *IEEE Trans. Comm.*, 28(5):711–719, May 1980.

[24] Adrian Segall. Distributed network protocols. *IEEE Trans. Info. Theory*, IT-29(1):23–35, January 1983. Some details in technical report of same name, MIT Lab. for Info. and Decision Syst., LIDS-P-1015; Technion Dept. EE, Publ. 414, July 1981.

[25] John M. Spinelli and Robert G. Gallager. Broadcasting topology information in computer networks. June 1987. unpublished.

Variables joint with the synchronizer

`Adjacent`: the set of adjacent links.

`State`: flag, updated by the MANAGER protocol. Receives values *dropping, growing*. The SYNCHRONIZER is operational only while `State` = *growing*.

`Pulse`: Highest pulse which was performed correctly until now. Receives values $\{1,2,..\ t-1, t_\pi\}$.

`Pulse(e)`: Estimate on `Pulse` of the node on the other end of edge e. Receives values $\{1,2,..\ t-1, t_\pi\}$.

Additional Variables

`Ack(e)`: Binary flag, indicating whether an `ACK` is expected on the edge e, maintained for all $e \in$ `Adjacent`.

`In-edge`: The parent of a node, maintained only while `State` = *dropping*.

Messages

`DROP(p)`: message carrying the parameter `Pulse` of the sender.

`ACK(p)`: acknowledgement to `DROP` message.

Procedures

BROADCAST-DROP: procedure which broadcasts `DROP` messages.

BROADCAST-GROW: procedure which broadcasts `GROW` messages.

CONVERGE: procedure which converges `ACK` messages towards roots.

Figure 16: Declarations of variables, messages, procedures

```
For change in node input
   case recovery on a link e
      Ack(e) := true
      add e to Adjacent
   case failure on a link e,
      delete e from Adjacent
   if Pulse ≠ 0 then
      Execute Procedure PARENT-SWITCH
      In-edge := nil
      Pulse := 0
      Execute Procedure BROADCAST-DROP

Procedure PARENT-SWITCH
   if In-edge ≠ nil then
      send ACK(Pulse-1) on In-edge

Procedure BROADCAST-DROP
   State := dropping
   for all links e ∈ Adjacent
      send DROP(Pulse) on e
      Ack(e) := false
   Execute CONVERGE

For DROP(p) on edge e
   Pulse(e) := p
   if p+1 ≥ Pulse then
      send ACK(p) on e
   else
      Execute Procedure PARENT-SWITCH
      Pulse := p+1
      In-edge := e
      Execute Procedure BROADCAST-DROP

For ACK(p) message on edge e
   if p = Pulse then
      Ack(e) := true
      Execute Procedure CONVERGE

Procedure CONVERGE
   if for all e ∈ Adjacent, Ack(e) = true then
      if In-edge ≠ nil then
         send ACK(Pulse+1) on In-edge
      else Execute Procedure BROADCAST-GROW

Procedure BROADCAST-GROW
   for all e ∈ Adjacent
      send GROW on e
   State := growing
   Invoke STATIC SYNCHRONIZER α

For GROW message on edge e
   if e = In-edge then
      Execute Procedure BROADCAST-GROW
```

Figure 17: Manager ζ