Cost-Sensitive Analysis of Communication Protocols (Extended Abstract)

Baruch Awerbuch * Alan Baratz † David Peleg ‡

Abstract

This paper introduces the notion of cost-sensitive communication complexity and exemplifies it on the following basic communication problems: computing a global function, network synchronization, clock synchronization, controlling protocols' worst-case execution, connected components, spanning tree, etc., constructing a minimum spanning tree, constructing a shortest path tree.

1 Introduction

Traffic load is one of the major factors affecting the behavior of a communication network. This fact is well recognized, and is the reason why most models for communication networks and most algorithms for routing, traffic analysis etc. model the network using a *weight function* on the edges, capturing this factor. In this model, the weight of an edge reflects the estimated delay for a message transmitted on this edge, and thus also the cost for using this edge. The significance of the load factor has also motivated the intense study of efficient methods for performing basic network tasks such as computing shortest paths and constructing minimum weight spanning trees (with length / weight defined with respect to the weight function).

However, in most of the previous work on *distributed algorithms* for these and other tasks, the design and analysis of the algorithms themselves completely disregards this weight function. That is, transmission over all the edges is assumed to be equally costly and completed within the same time bound. Such assumptions are made even when the task performed by the algorithm is directly related to the edge costs, and the algorithm has to be executed over the same network, and thus suffer the same delays. This seems to contradict the very purpose towards which the tasks are performed. It is sometimes argued that it is not crucial to take the weights into account when considering such "network service" algorithms, since these algorithms occupy only a thin slice of the network's bandwidth. Nonetheless, it is clear that an algorithm that can do well in that respect is preferable to one that ignores the issue.

This paper proposes an approach enabling us to take traffic loads into account in the design of distributed algorithms. This issue is addressed by introducing *cost-sensitive* complexity measures

*Dept. of Mathematics and Lab. for Computer Science, M.I.T., Cambridge, MA 02139. ARPA: baruch@theory.lcs.mit.edu. Supported by Air Force Contract TNDGAFOSR-86-0078, ARO contract DAAL03-86-K-0171, NSF contract CCR8611442, DARPA contract N00014-89-J-1988, and a special grant from IBM. Part of the work was done while visiting IBM T.J. Watson Research Center.

†IBM T.J. Watson Research Center, Yorktown Heights, NY 10598.

‡Dept. of Applied Mathematics, The Weizmann Institute, Rehovot 76100, Israel. BITNET: peleg@wisdom. Supported in part by an Allon Fellowship, by a Bantrell Fellowship and by a Haas Career Development Award. Part of the work was done while visiting MIT and IBM T.J. Watson Research Center.

for analysis of distributed protocols. We consider weighted analogs for both communication and time complexity. We then examine a host of basic network problems, such as connectivity, computing global functions, network synchronization, controlling the worst-case execution of protocols, and constructing minimum spanning trees and shortest path trees. For each of these problems we seek to establish some lower bounds and propose some efficient algorithms with respect to the new complexity measures.

We feel that the approach proposed in this paper may serve as a basis for a more accurate account of the behavior of distributed algorithms in communication networks.

1.1 The model

We consider the standard model of (static) asynchronous communication networks. We consider a communication graph $G = (V, E, w)$, where a weight $w(e)$ is associated with each (undirected) edge of the network. We denote $n = |V|$, $m = |E|$. We also denote by W the maximal weight $w(e)$ of a network edge, $W = \max_{(u,v)\in E} w(u,v)$. We make the assumption that $W = poly(n)$, and thus $\log W = O(\log n)$. For any subgraph $G' = (V', E', w)$ of G, let $w(G')$ denote the total weight of G', i.e., $w(G') = \sum_{e\in E'} w(e)$. Let $dist(u, v, G')$ be the weighted distance from u to v in G', i.e., the minimum of $w(p)$ over all paths p from u to v in G'. Let $Diam(G')$ denote the *diameter* of G', i.e., $\max_{u,v\in V'} dist(u, v, G')$. Given a tree T and two vertices x, y in it, denote by $Path(x, y, T)$ the path from x to y in T.

1.2 The complexity measures

This paper introduces *weighted* complexity measures analogous to the traditional time and communication measures. We define the *cost* of transmitting a message over an edge e as $w(e)$. The *communication complexity* of a protocol π, denoted c_π, is the sum of all transmission costs of all messages sent during the execution of π. The *time complexity* of the protocol π, denoted t_π, is the maximal physical time it takes π to complete its execution, assuming that the delay on an edge e varies between 0 and $w(e)$. The classical complexity measures correspond to the case where $w(e) = 1$ for all $e \in E$.

Traditionally, communication protocols are evaluated in terms of E, V, D, which denote, respectively, the number of edges, the number of vertices, and the unweighted (hop based) diameter of the network. It turns out that it is convenient to evaluate the weighted complexity of protocols using the "weighted analogs" of E, V, D, denoted by $\mathcal{E}, \mathcal{V}, \mathcal{D}$, which are defined as follows:

$$\begin{aligned} \mathcal{E} &= w(G) \left(= \sum_{e\in E} w(e) \right) \\ \mathcal{V} &= w(T) \text{ where } T \text{ is an MST of } G \\ \mathcal{D} &= Diam(G) \end{aligned}$$

The analogy between these parameters and their unweighted counterparts is manifested in the fact that $\mathcal{E}$ equals the total cost of transmitting a single message over all the edges of the network, $\mathcal{V}$ is the minimal cost of reaching (or, disseminating a message to) all vertices, and $\mathcal{D}$ is the maximal cost of transmitting a message between a pair of network nodes.

In the sequel we express the complexity of our algorithms in terms of $\mathcal{E}, \mathcal{V}$ and $\mathcal{D}$. This gives results that are conveniently similar in appearance to the results of the unweighted case, as follows from the statement of results in the following subsection.

1.3 Problems and results

1.3.1 Global function computation

The problem: We are concerned here with computing global functions in a network. We assume that the structure of the network is known to all the vertices (including the edge weights). The only unknowns are the values of the n arguments of the function, which are initially stored at different vertices of the network, one at each vertex. The outputs must be be produced at all the vertices.

We restrict ourselves to the family of functions called *symmetric compact* in [GS86]. The functions $f_n : X^n \to X$ in this family are symmetric (i.e., any two arguments can be switched)

Global function computation		
	Communication	Time
Upper bound	$\mathcal{V}$	$\mathcal{D}$
Lower bound	$\Omega(\mathcal{V})$	$\Omega(\mathcal{D})$

Figure 1: Lower and upper bounds for global functions.

and compact, in the sense that the contribution of any subset of arguments can be represented in "compact form" by a string of size $\log_2 |X|$. The latter condition is formalized by assuming that there exists a function $g : X^2 \rightarrow X$ such that for any $k < n$, $f(x_1, x_2 \ldots x_n) = g(f_k(x_1, x_2 \ldots x_k), f_{n-k}(x_{k+1}, x_{k+2} \ldots x_n))$.

Computing such functions is quite a basic task in the area of network protocols. Many functions belong to this family, e.g. maximum, sum, basic boolean functions (XOR, AND, OR). Many other tasks, e.g. broadcasting a message from a given node to the rest of the network, termination detection, global synchronization, etc. can be represented as computing a symmetric compact function. A similar class of functions is considered in [ALSY88].

The results: We show that the computation of global functions requires $\Theta(\mathcal{V})$ messages and $\Theta(\mathcal{D})$ time.

The upper bound is derived as follows. Define a spanning tree as *shallow-light tree* (SLT) if its diameter is $O(\mathcal{D})$ and its weight is $O(\mathcal{V})$. We then show that SLT trees are effectively constructible, which implies that computing the value of our global function can be performed (optimally) with $O(\mathcal{V})$ messages and $O(\mathcal{D})$ time.

We are also concerned with efficient distributed constructions of SLT trees, or, in short, SLT algorithms. We present a specific SLT algorithm that requires $O(\mathcal{V} \cdot n^2)$ communication and $O(\mathcal{D} \cdot n^2)$ time.

1.3.2 Clock Synchronization

Problem: The purpose of the clock synchronization is to generate at each node a sequence of pulses, such that pulse p at a node is generated *after* (in the "causal" sense [Lam78]) all neighbors generate pulse $p - 1$.

As argued by Even and Raijsbaum [ER90], the relevant complexity measure here is the "pulse delay", which is the maximal time delay in between two successive pulses at a node. Let us denote $d = \max_{(u,v) \in E} dist(u, v)$, i.e., d is largest distance between neighbors in the network. Clearly $d \leq W$, and the problem is interesting when $d \ll W$. A lower bound of $\Omega(d)$, and an upper bound of $O(W)$ are derived in [ER90]. (It is worth pointing out that the main emphasis of [ER90] is on somewhat different "directed" version of this problem.)

Results: In this paper, we show that one can achieve a pulse delay of $O(d \cdot \log^2 n)$, i.e. leave a gap of $\log^2 n$ between the lower and upper bounds. This result relies heavily on a number of existing techniques, like the "Network Partition" of [AP89], and the "Synchronizer γ" of [Awe85a].

1.3.3 Network Synchronization

The problem: Asynchronous algorithms are in many cases substantially inferior in terms of their complexity to corresponding synchronous algorithms, and their design and analysis are more complicated. This motivates the development of a general simulation technique, known as the *synchronizer*, that allows users to write their algorithms as if they are run in a synchronous network. Implicitly, such techniques were proposed already in [Jaf80] and [Gal82]. The first explicit statement of the problem was given in [Awe85a]. Even better construction is known for special networks, like hypercubes [PU89]. Our goal is to extend the concept of the synchronizer to the weighted case and provide an appropriate construction.

On a conceptual level, the synchronizer (as well as the *controller*, described in following sections) is a protocol transformer, transforming a protocol π into a protocol ϕ that is equivalent to π in some sense but enjoys some additional desirable properties. Recall that c_π and t_π denote the communication and time complexity of the protocol π, and similarly for c_ϕ and t_ϕ. Our purpose is to guarantee that the transformation maintains c_ϕ

and t_ϕ small compared to c_π and t_π.

The synchronizer can be viewed as a way to remove variations from link delays in an asynchronous network. In the "unweighted" case, this means that we want to "force" all link delays to be exactly 1. In the "weighted" case, the most natural and most useful generalization of this concept is to force the delay on each link e to be exactly $w(e)$. In a sense, the synchronizer enables to simulate a "weighted" synchronous network $G(V,E,w)$ with each link e having a delay of exactly $w(e)$ by a "weighted" asynchronous network $G(V,E,w)$. Such simulations may be useful for various applications, for which the absence of variations in edge delays significantly simplifies the tasks in hand, e.g., shortest paths [Awe89], constructing routing tables [ABNLP89], and others. However, in addition to simplifying protocol design and analysis, synchronizers actually lead to complexity improvements for concrete algorithms. For example, the algorithm $\mathtt{SPT}_{synch}$ derived via a synchronizer (see full paper), is the best known shortest path algorithm for certain values of $\mathcal{V},\mathcal{D},\mathcal{E}$.

We define the *amortized* costs of a synchronizer ξ (i.e., the overhead per pulse) in communication and time as follows.

$$\mathcal{C}_p(\xi) = \frac{c_\phi - c_\pi}{t_\pi}$$
$$\mathcal{T}_p(\xi) = \frac{t_\phi}{t_\pi}$$

At first sight, the clock synchronization problem from Subsection 1.3.2 seems to resemble the problem of simulating an "unweighted" synchronous network $G'(V,E)$ (with all link delays being exactly 1) by a "weighted" asynchronous network $G(V,E,w)$. The main difference is in the fact that the only goal of the network synchronizer is to simulate a particular protocol, whereas the purpose of the clock synchronizer is to generate pulses. In general, it would be ineffective to use clock synchronizers for network synchronization, and vice versa. Even though the methods that we use to handle both problems have certain techniques in common, the differences are quite substantial.

The results: We construct a synchronizer γ_w, which is an analog of synchronizer γ of [Awe85a], such that

Synchronizers		
	Communication	Time
Upper bound	$kn\log n$	$\log_k n \cdot \log n$
Lower bound	$\Omega(kn)$	$\Omega(\log_k n)$

Figure 2: Lower and upper bounds for network synchronization.

$$\mathcal{C}_p(\gamma_w) = O(kn\cdot\log n)$$
$$\mathcal{T}_p(\gamma_w) = O(\log_k n\cdot\log n)$$

1.3.4 Controllers

Problem: The controller [AAPS87] is a protocol transformer transforming a protocol π into a protocol ϕ that is equivalent to π in terms of its input-output relation on a static network, but is more "robust" than π in the sense that it has "reasonable" complexity even if it operates on "wrong" data.

Results: In the unweighted case, [AAPS87] presents a controller guaranteeing $t_\phi = c_\phi = O(c_\pi\cdot\log^2 c_\pi)$. We show that the same bounds hold for the weighted case as well.

1.3.5 Connected components, spanning tree

The problems: The problems considered here are finding connected components and constructing a (not necessarily minimum) spanning tree [Seg83, AGPV89]. These problems are equivalent to each other.

The results: We show that performing any of the above tasks requires $\Theta(\min\{\mathcal{E}, n\cdot\mathcal{V}\})$ communication by providing matching upper and lower bounds. To be more precise, we prove that

1. For every distributed connectivity algorithm A and for any n there exists a family of n vertex graphs G on which A requires communication complexity $\Omega(n\cdot\mathcal{V})$ and a family

Connectivity		
	Communication	Time
DFS	$\mathcal{E}$	$\mathcal{E}$
$\texttt{CON}_{flood}$	$\mathcal{E}$	$\mathcal{D}$
$\texttt{CON}_{hybrid}$	$\min\{\mathcal{E}, n \cdot \mathcal{V}\}$	$\min\{\mathcal{E}, n \cdot \mathcal{V}\}$
Lower bound	$\Omega(\min\{\mathcal{E}, n \cdot \mathcal{V}\})$	$\Omega(\mathcal{D})$

Figure 3: Our Connectivity algorithms.

Minimum Spanning Trees (MST)		
Algorithm	Communication	Time
$\texttt{MST}_{ghs}$	$\mathcal{E} + \mathcal{V} \cdot \log n$	$\mathcal{E} + \mathcal{V} \cdot \log n$
$\texttt{MST}_{centr}$	$n \cdot \mathcal{V}$	$n^2\mathcal{D}$.
$\texttt{MST}_{fast}$	$\mathcal{E} \cdot \log n \log \mathcal{V}$	$\mathcal{D} \cdot n \cdot \log n \cdot \log \mathcal{V}$.
$\texttt{MST}_{hybrid}$	$\min\{\mathcal{E}, n \cdot \mathcal{V} \cdot \log n\}$	$\min\{\mathcal{E}, n \cdot \mathcal{V} \cdot \log n\}$
Lower bound	$\Omega(\min\{\mathcal{E}, n \cdot \mathcal{V}\})$	$\Omega(\mathcal{D})$

Figure 4: Our MST algorithms.

of n vertex graphs G on which A requires communication complexity $\Omega(\mathcal{E})$.

2. There is a distributed connectivity algorithm with communication complexity $O(\min\{\mathcal{E}, n \cdot \mathcal{V}\})$ on any graph G.

1.3.6 Constructing minimum spanning trees

Problem: The *minimum spanning tree (MST)* of the graph G is a tree of minimum weight spanning G.

Results: We develop a number of MST algorithms, based on modifications of the algorithms of [GHS83, Awe87].

1. An algorithm with communication complexity $O(\min\{\mathcal{E} + \mathcal{V} \cdot \log n,\ n \cdot \mathcal{V}\})$.

2. An algorithm with communication complexity $O(\mathcal{E} \cdot \log n \log \mathcal{V})$ and time complexity $O(\mathcal{D} \cdot n \cdot \log n \cdot \log \mathcal{V})$.

Shortest Path Trees (SPT)		
Algorithm	Communication	Time
$\texttt{SPT}_{centr}$	$n^2 \cdot \mathcal{V}$	$\mathcal{D} \cdot n$
$\texttt{SPT}_{recur}$	$\mathcal{E}^{1+\epsilon}$	$\mathcal{D}^{1+\epsilon}$
$\texttt{SPT}_{synch}$	$\mathcal{E} + \mathcal{D} \cdot kn \cdot \log n$	$\mathcal{D} \cdot \frac{\log^2 n}{\log k}$
$\texttt{SPT}_{hybrid}$	$\min\{\mathcal{E} + \mathcal{D} \cdot kn \cdot \log n), \mathcal{E}^{1+\epsilon}\}$	$\mathcal{D}^{1+\epsilon}$
Lower bound	$\Omega(\min\{\mathcal{E}, n \cdot \mathcal{V}\})$	$\Omega(\mathcal{D})$

Figure 5: Our SPT algorithms.

1.3.7 Constructing shortest path trees

Problem: The *shortest paths tree (SPT)* of the graph G with respect to a source vertex $s \in V$ is a tree defined by the collection of shortest paths from s to all other vertices in G.

Results: We develop a number of SPT algorithms:

1. An algorithm with communication complexity $O(\mathcal{E}^{1+\epsilon})$ and time complexity $O(\mathcal{D}^{1+\epsilon})$. This is analogous to the result of [Awe89], which achieves same result for the unweighted case.

2. An algorithm with communication complexity $O(\mathcal{E} + \mathcal{D} \cdot kn \cdot \log n)$ and time complexity $O(\mathcal{D} \cdot \log_k n \log n)$.

1.4 Structure of this paper

The paper proceeds as follows. Section 2 gives tight upper and lower bounds on the computation of global functions. Section 3 contains clock synchronization algorithms. In Section 4, we give upper and lower bounds for network synchronizers. Finally, in Section 5 we give a lower bound for the problems of constructing connected components, spanning tree, and others.

The rest of the results will appear in the full paper.

2 Optimal computation of global functions

2.1 The lower bound

Theorem 2.1 The computation of global symmetric compact functions requires $\Omega(\mathcal{V})$ communication and $\Omega(\mathcal{D})$ time.

Proof: Suppose that the value of the function has been computed at the vertex v. Since the value of a global function depends on the value of all of its arguments, there must be some information flow from each of the vertices to v. Thus the subgraph $G'(V, E')$, defined by the set of edges E' traversed by messages of the protocol, must contain a path from v to any other vertex in V, i.e., it must contain some spanning tree of G.

Observe that the distance $dist(u, v, G')$ from v to any other vertex $u \in V$ is a lower bound on time complexity of the protocol. Picking a pair of vertices u, v realizing $\mathcal{D}$ (i.e., maximizing the distance $dist(u, v, G)$) and noting that $dist(u, v, G') \geq dist(u, v, G) = \mathcal{D}$, we get that $\mathcal{D}$ is a lower bound on the time complexity of the protocol.

Furthermore, the total weight of the edges of G', $w(G')$, is a lower bound on the communication complexity of the computation. Now, since G' contains a spanning tree of G, its total weight satisfies $w(G') \geq \mathcal{V}$. Thus $\mathcal{V}$ lower bounds the communication complexity of the protocol. ∎

2.2 The upper bound

It is easy to see that given a spanning tree T for the network, a global function can be computed with communication complexity $w(T)$ and time complexity $Diam(T)$. Clearly, any shortest path tree T_S has small depth, namely $Diam(T_S) = O(\mathcal{D})$, but its weight may be as big as $w(T_S) = \Omega(n \cdot \mathcal{V})$. Analogously, any minimum spanning tree T_M has small weight, namely $w(T_M) = \mathcal{V}$; but its depth may be as high as $Diam(T_M) = \Omega(n \cdot \mathcal{D})$.

Recall that a spanning tree is *shallow-light tree* (SLT) if its diameter is $O(\mathcal{D})$ and its weight is $O(\mathcal{V})$. Such trees minimize simultaneously both weight and depth; existence of such tree would imply that in any graph, one can compute global functions with communication complexity $O(\mathcal{V})$ and $O(\mathcal{D})$ time. However, it is not clear that such trees exist. In the next subsection we establish

Theorem 2.2 Every graph has a shallow-light spanning tree.

Corollary 2.3 The computation of global symmetric compact functions can be performed with communication complexity $O(\mathcal{V})$ and $O(\mathcal{D})$ time. ∎

The shallow-light tree algorithm

We next provide an algorithm (hereafter referred to as the SLT algorithm) for constructing an SLT for an arbitrary graph, thus proving Theorem 2.2.

1. Construct an MST T_M and an SPT T_S for G.

2. Traverse T_M in a DFS fashion. Observe that DFS defines a "tour" through the tree, in which each tree edge is traversed exactly twice. Define the "mileage" of the center of activity of the DFS to be the number of tree edge traversals (forward and backward) up to this time. Denote by $\nu(i)$ $(0 \leq i \leq 2(n-1))$ the location of the center of activity of the DFS at the time its mileage is exactly i. For example, $\nu(0) = \nu(2n-2) = s$, where s is the source of the DFS.

3. Construct the "line-version" L of T_M, which is a (weighted) path graph containing vertices $0, 1, \ldots 2(n-1)$. A vertex i on the path corresponds to $\nu(i)$. We assign each edge $e = (i, i+1)$ on the line L the weight of the corresponding edge $(\nu(i), \nu(i+1))$ in the graph G. Observe that the total weight of the line is at most twice the total weight of the MST T_M, i.e., $w(L) \leq 2\mathcal{V}$.

4. Fix a parameter $q > 0$. Construct "break-points" B_i on L by scanning it from left to right according to the following rules.

   (a) Break-point B_1 is vertex 0 on the line L.

Construct a minimum spanning tree T_M for G.
Construct a shortest path tree T_S for G.
Construct L based on T_M as described above.
Assign each edge e of L the same weight as $\nu(e)$ in G.
$E' \leftarrow T_M$
$X \leftarrow 0; Y \leftarrow 0$
repeat
  repeat $Y \leftarrow Y + 1$
  until $dist(X,Y,L) > q \cdot dist(X,Y,T_S)$
    $E' \leftarrow E' \bigcup Path(X,Y,T_S)$
    $X \leftarrow Y$
until $Y = n$
Construct a shortest path tree T in $G' = (V,E')$
output T

Figure 6: The SLT algorithm

(b) Break-point B_{i+1} is the first point to the right of B_i such that $dist(B_i, B_{i+1}, L) > q \cdot dist(\nu(B_i), \nu(B_{i+1}), T_S)$, meaning that the distance from B_i to B_{i+1} in T_M exceeds that in T_S by a factor of at least q.

5. Create a subgraph G' of G by taking T_M and adding $Path(\nu(B_i), \nu(B_{i+1}), T_S)$ for all break-points B_i, $i > 1$ (i.e., shortest paths connecting $\nu(B_i)$ to $\nu(B_{i-1})$).

6. Construct a shortest path tree T in the resulting graph G'.

7. Output T.

The algorithm is presented formally in Figure 6. In the algorithm, T denotes the set of edges selected to the shallow-light tree and X, Y are pointers on the line L.

2.3 Analysis

Claim 2.4 The tree T constructed by the algorithm satisfies $w(T) \leq 2 \cdot (1 + \frac{1}{q}) \cdot \mathcal{V}$. ∎

Claim 2.5 The tree T constructed by the algorithm satisfies $d(T) \leq (2q+1) \cdot \mathcal{D}$. ∎

Corollary 2.6 The tree T constructed by the algorithm is an SLT.

As for a distributed construction of shallow-light trees, we show in the full paper

Theorem 2.7 There is a distributed algorithm for constructing an SLT requiring $O(\mathcal{V} \cdot n^2)$ communication and $O(\mathcal{D} \cdot n^2)$ time.

3 Clock synchronization

In this section we describe three methods of clock synchronization, called synchronizer α^*, β^* and γ^*. These are modifications of synchronizers α, β and γ of [Awe85a].

3.1 Clock synchronizer α^*

As pointed out in [ER90], the most natural approach to clock synchronization is to use the following synchronization mechanism, called *synchronizer* α^*.

Synchronizer α^*: whenever a node generates pulse p, is send messages to all neighbors, and when it receives messages of pulse p from all neighbors, it generates pulse $p+1$.

This method clearly requires time proportional to the highest edge weight, namely $O(W)$. Our goal is to approach the lower bound, which is $O(d)$ (recall that d is largest distance between neighbors).

The naive way to improve the delay is to construct a shortest path $Path(u,v)$ for all $(u,v) \in E$ and to communicate with each neighbor over such path. The problem with this method is that a particular edge may belong to many paths (up to E), and thus the resulting congestion will slow down the communication time by the corresponding factor (up to E).

3.2 Clock synchronizer β^*

In order to minimize congestion, we may try the following method, called *synchronizer* β^*.

Preprocessing: We construct a spanning tree T of the network, and select a "leader" to be the root of this tree.

Pulse generation: Information about the completion of the current pulse is gathered up the tree by means of a communication pattern referred to as *convergecast*, which is started at the leaves of the tree and terminates at the root. Namely, whenever a node learns that it is done with this pulse and all its descendants in the tree are done with it as well, it reports this fact to its parent. Thus within finite time after the execution of the pulse, the leader eventually learns that all the nodes in the network are done. At that time it broadcasts a message along the tree, notifying all the nodes that they may generate a new pulse.

The time complexity of Synchronizer β^* is $\Omega(\mathcal{D})$, because the entire convergecast and broadcast process is performed along a spanning tree, whose depth is at least the diameter of the network.

3.3 Clock synchronizer γ^*

Our final synchronizer, called *synchronizer* γ^*, combines synchronizer γ of [Awe85a]) with the network partitions of [AP89].

Definition 3.1 Given an n-vertex weighted graph $G(V, E, w)$, a *tree edge-cover* for G is a collection $\mathcal{M}$ of trees, such that

1. every edge of G is shared by at most $O(\log n)$ trees of $\mathcal{M}$,
2. the depth of each tree in $\mathcal{M}$ is at most $O(\log n \cdot d)$, and
3. for each edge, there exists at least one tree containing both endpoints.

Lemma 3.2 For every n-vertex weighted graph $G(V, E, w)$, it is possible to construct a *tree edge-cover*.

Proof: The desired collection of trees can be constructed using the techniques of [AP89]. Details will be given in the full paper. ∎

Preprocessing: Construct a tree edge-cover for G. Inside each tree, a *leader* is chosen to coordinate the operations of tree. We call two trees *neighboring* if they share a node.

Pulse generation: The process is performed in two phases. In the first phase, Synchronizer β is applied separately in each tree. Whenever the leader of a tree learns that its tree is done, it reports this fact to all the nodes in the tree which relay it to the leaders of all the neighboring trees. Now, the nodes of the tree enter the second phase, in which they wait until all the neighboring trees are known to be done and then generate the next pulse (as if Synchronizer α^* is applied among trees). More details will be given in the full paper.

Complexity: The "congestion" caused by the fact that messages of different trees cross the same edge, adds at most an $O(\log n)$ multiplicative factor to the time overhead. Since the height of each tree is $O(d \log n)$, it follows that the time to simulate one pulse is $O(d \cdot \log^2 n)$.

4 Synchronizers

The lower bound of [Awe85a] for the unweighted case holds here as well.

Lemma 4.1 [Awe85a] For any k, any synchronizer ξ satisfying $C_\xi = O(kn)$, must have $T_\xi = \Omega(\log_k n)$. ∎

In the rest of this section, we develop our synchronizer in a number of steps.

The synchronizers discussed in this section operate by generating sequences of "clock-pulses" at each vertex of the network, satisfying the following property: pulse p is generated at a vertex only after it receives all the messages of the synchronous algorithm that arrive at that vertex prior to pulse p. This property ensures that the network behaves as a synchronous one from the point of view of the particular synchronous algorithm.

The problem arising with synchronizer design is that a vertex cannot know which messages were sent to it by its neighbors and there are no bounds on edge delays. Thus, the above property cannot be achieved simply by waiting "enough time" before generating the next pulse, as may be possible in a network with bounded delays. However, it

may be achieved if additional messages are sent for the purpose of synchronization.

In the unweighted synchronizers of [Awe85a], incoming links are "cleaned" from transient messages in between any two consecutive pulses, similar to the clock synchronizers in Section 3. In our (weighted) case, this would be very inefficient since cleaning the links requires time proportional to the maximal link weight $W \gg 1$, which would therefore dictate the multiplicative overhead of the synchronization. The idea for overcoming this problem is that links of high weight should be cleaned less frequently, thus enabling to amortize the cost of cleaning them over longer time intervals.

We need to state a number of definitions first.

Definition 4.2 Given a synchronous protocol π running on a synchronous weighted network $G(V, E, w)$, we say that π is *in synch* with G if π transmits a message on edge e only at times that are divisible by $w(e)$.

Definition 4.3 A weighted network $G(V, E, w)$ is said to be *normalized* if all weights $w(e)$ are powers of 2.

Informally, our solution proceeds according to the following plan.

1. Design a synchronizer for normalized networks and protocols that are in synch with the networks on which they are run.
2. Show that one can transform an arbitrary synchronous protocol π and synchronous network G, so that the above assumptions are satisfied, without significantly increasing the complexities.

These two steps are described in the following two subsections.

4.1 Synchronizer γ_w

We assume now that the weights of all network edges are powers of 2, and messages are sent on an edge of weight 2^i only at times divisible by 2^i.

Let $\delta = \log W$. We define a collection of subnetworks $\{G_i(V, E_i) \mid 0 \leq i \leq \delta\}$, by defining E_i to be the set of edges whose weights are divisible by 2^i. (Note that an edge e with weight $w(e) = 2^j$ occurs in all graphs G_i for $j \geq i$.)

The idea is that pulses divisible by 2^i are handled by a so-called synchronizer γ_i, which is exactly synchronizer γ of [Awe85a], applied to the graph G_i. The synchronizer γ_i treats pulse $p \cdot 2^i$ as "super-pulse" p. It guarantees that super-pulse p is executed only *after* all messages sent along edges in E_i at super-pulse $(p-1)$ have arrived.

A vertex has to satisfy all δ synchronizers in order to proceed with a pulse. More specifically, consider a pulse $p = 2^j \cdot (2r+1)$, i.e., such that 2^j is the maximal power of 2 dividing p. Then pulse p is postponed until super-pulse $(2r+1) \cdot 2^{j-i}$ of synchronizer γ_i is executed. For example, pulse $24 = 3 \cdot 2^3$ is completed only after the synchronizers γ_0, γ_1, γ_2, and γ_3 are done carryng their pulses 24, 12, 6 and 3, respectively.

Lemma 4.4 Synchronizer γ_w is correct.

Proof Sketch: We need to show that under synchronizer γ_w, a vertex v generates pulse p only after receiving all messages it would receive by pulse p were the protocol executed on a synchronous network. This follows from the fact that the set of messages it would get by pulse p, i.e., the set of messages affecting this pulse, includes messages sent on edges belonging to G_i sent at pulse $p - 2^i$, and the arrival of these messages is guaranteed by synchronizer γ_i. ∎

4.2 Designing the protocol transformation

In order to justify the assumptions of the previous subsection we need to prove the following claim.

Lemma 4.5 Given a synchronous protocol π running on a synchronous weighted network $G(V, E, w)$, there exist a synchronous protocol π' and a synchronous network $G'(V, E, w')$ with the following properties:

1. G' is normalized.
2. The protocol π' is in synch with G'.

3. The output of π' on G' is identical to the output of π on G.

4. The time and communication complexities of a run of π' on G' are at most twice higher than the complexities of the corresponding run of π on G.

The proof is postponed to the full paper.

4.3 Complexity

Lemma 4.6 The synchronizer γ_w described above has the following complexities:

$$\begin{aligned} \mathcal{C}_p(\gamma_w) &= O(k \cdot n \cdot \log W) = O(k \cdot n \cdot \log n) \\ \mathcal{T}_p(\gamma_w) &= O(\log_k n \cdot \log W) = O(\log_k n \cdot \log n) \end{aligned}$$

Proof: Synchronizer γ_i is invoked on the graph G_i once every 2^i time units. This costs us $O(2^i \cdot n \cdot k)$ in communication and $O(2^i \cdot \log_k n)$ time. This waste is amortized over 2^i time units, and then summed over all $0 \leq i \leq \log W$ graphs G_i. ∎

5 Connected components and spanning tree construction

In this section, we prove matching upper and lower bounds on the communication complexity of performing the tasks of finding connected components and constructing a spanning tree.

5.1 Lower bounds

Let us first point out that an $\Omega(\mathcal{E})$ lower bound on communication is given in [AGPV89] for the case where all edge weights are unity. In the rest of this subsection, we prove an $\Omega(n \cdot \mathcal{V})$ lower bound on the communication complexity.

Consider the family of graphs $G_n = (V, E, w)$ defined as follows. $V = \{1, \ldots, n\}$. The set of edges is composed of two subsets, $E = E_p \cup E_b$, where the first subset creates a *path*, $E_p = \{(i, i+1) \mid 1 \leq i \leq n-1\}$, and the second subset consists of *bypassing edges*, $E_b = \{(i, n+1-i) \mid 1 \leq i \leq n/2\}$. The weights are defined as

$$w(e) = \begin{cases} X, & e \in E_p, \\ X^4, & e \in E_b, \end{cases}$$

where X is some large value, say $X \geq n$.

Note that the MST for G is the subgraph (V, E_p) based on the path alone, so $\mathcal{V} = nX$.

We make some assumptions similar to those of [AGPV89] regarding the model. In particular, we assume that the only operation one can do with ID's is comparisons; this can be extended also to general operations in case the ID's are allowed to be sufficiently large.

Let A be a deterministic algorithm that succeeds in computing a spanning tree on every input graph and whose communication complexity is $f(n) = o(n^4)$. In particular, this means that there exists a constant n_0 such that for every $n > n_0$, the algorithm A completes the construction of tree on G_n with communication cost less than n^4. Clearly, then, the algorithm does not send any messages over any bypassing edge in these graphs, since using such an edge immediately incurs a cost of n^4. Henceforce we restrict attention to graphs G_n for $n > n_0$.

Lemma 5.1 Algorithm A requires $\Omega(n\mathcal{V})$ messages. ∎

5.2 An upper bound

Claim 5.2 Algorithm $\texttt{CON}_{hybrid}$ (presented in the full paper) requires $O(\min\{\mathcal{E}, n \cdot \mathcal{V}\})$ messages.

Acknowledgments

We warmly thank Oded Goldreich for his illuminating comments on a previous draft.

References

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

[ABNLP89] Baruch Awerbuch, Amotz Bar-Noy, Nati Linial, and David Peleg. Compact distributed data structures for adaptive network routing. In *Proc. 21st ACM Symp. on Theory of Computing*, pages 230–240. ACM SIGACT, ACM, May 1989.

[AGPV89] Baruch Awerbuch, Oded Goldreich, David Peleg, and Ronen Vainish. A tradeoff between information and communication in broadcast protocols. *J. of the ACM*, 1989. to appear.

[ALSY88] Y. Afek, G.M. Landau, B. Schieber, and M. Yung. The power of multimedia: combining point-to-point and multiaccess networks. In *Proc. of the 7th ACM Symp. on Principles of Distributed Computing*, pages 90–104, Toronto, Canada, August 1988.

[AP89] Baruch Awerbuch and David Peleg. Routing with polynomial communication-space trade-off. Technical Memo TM-411, MIT, Lab. for Computer Science, September 1989.

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

[Awe85b] Baruch Awerbuch. A new distributed depth-first-search algorithm. *Info. Process. Letters*, 20:147–150, April 1985.

[Awe87] Baruch Awerbuch. Optimal distributed algorithms for minimum weight spanning tree, counting, leader election and related problems. In *Proceedings of the* 19^{th} *Annual ACM Symposium on Theory of Computing*, pages 230–240. ACM, May 1987.

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

[DS80] Edsger W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Info. Process. Letters*, 11(1):1–4, August 1980.

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

[Eve79] Shimon Even. *Graph Algorithms*. Computer Science Press, 1979.

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

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

[GS86] O. Goldreich and L. Shrira. The effects of link failures on computations in asynchronous rings. In *Proc. 5th ACM Symp. on Principles of Distributed Computing*, pages 174–186. ACM, August 1986.

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

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. of the ACM*, 21(7):558–565, July 1978.

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

[Seg83] Adrian Segall. Distributed network protocols. *IEEE Trans. on 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.