MIT/LCS/TM-268

# COMPLEXITY OF NETWORK SYNCHRONIZATION

Baruch Awerbuch

January 1985

# COMPLEXITY OF NETWORK SYNCHRONIZATION [1]

by

Baruch Awerbuch[2]

M.I.T.

*Abstract*

In this paper we investigate the problem of simulation of the synchronous network by the asynchronous one. We propose a new simulation technique, referred to as "Synchronizer" which is a new, simple methodology for designing efficient distributed algorithms in asynchronous networks. Our Synchronizer exhibits a trade-off between its communication and time complexities, which is proved to be within a constant factor of the lower bound.

[1] An earlier version of this paper has been presented at the ACM Symposium on Theory of Computing, Washington D.C., May 1984.

[2] The author has been supported by the Chaim Weizmann Postdoctoral Fellowship.

## 1. Introduction

The asynchronous algorithms are in many cases substantially inferior in terms of their complexity to corresponding synchronous algorithms, and their design and analysis are much more complicated. Thus, it makes sense to develop a general simulation technique, referred to as "Synchronizer", which will allow the user to write his algorithm as if it is run in a synchronous network. Such technique is proposed in this paper. Essentially, this is a new, simple methodology for designing efficient distributed algorithms in asynchronous networks. No such methodology was previously proposed in the literature for our model; some of the related works are mentioned in the summary.

We also prove existence of certain trade-off between communication and time requirements of *any* Synchronizer. It turns out that our Synchronizer achieves this lower bound within a constant factor.

For problems for which there are good synchronous algorithms, our Synchronizer allows simple construction of low complexity algorithms. We demonstrate its power on the distributed *Maximum Flow* and *Breadth–First–Search* (BFS) algorithms. These are very fundamental graph-theoretic problems. Both of them can be solved by fairly simple and elegant algorithms in a synchronous parallel computation model (*PRAM*), as shown in [SV-82] and [EC-77]. These algorithms have been easily modified in [A-83] for operation in a distributed synchronous network. The complexities of the resulting algorithms are summarized in Table 1. For precise definition of complexity measures used, see Section 2. Applying our Synchronizer to the algorithms of Table 1 yields new asynchronous algorithms, which improve the best existing ones both in terms of communication and time. Our improvements are summarized in Table 2. In the rest of the paper we proceed as follows. In Section 2 we describe the two models we are dealing with, namely the asynchronous and the synchronous networks, define pre-

Table 1: The Complexities of Synchronous Algorithms

| Problem | Adapted from PRAM algorithm of | Communication Complexity | Time Complexity |
|---|---|---|---|
| Breadth First Search | $[EC-77]$ | $\lvert E \rvert$ | $\lvert V \rvert$ |
| Maximum Flow | $[SV-82]$ | $\lvert V \rvert^3$ | $\lvert V \rvert^2$ |

Table 2: The Complexities of Asynchronous Algorithms

| Problem | Reference &Author | Communication Complexity | Time Complexity | Values of Parameters |
|---|---|---|---|---|
| Breadth First Search | [G-82] Gallager | $\lvert V \rvert \lvert E \rvert$ | $\lvert V \rvert$ | |
| | [G-82] Gallager | $\lvert V \rvert^{2+x}$ | $\lvert V \rvert^{2-2x}$ | $0 \leq x \leq 0.25$ |
| | This paper | $k \lvert V \rvert^2$ | $\lvert V \rvert \frac{\log_2 \lvert V \rvert}{\log_2 k}$ | $2 \leq k < \lvert V \rvert$ |
| Maximum Flow | [S-82] Segall | $\lvert V \rvert \lvert E \rvert^2$ | $\lvert V \rvert^2 \lvert E \rvert$ | |
| | This paper | $k \lvert V \rvert^3$ | $\lvert V \rvert^2 \frac{\log_2 \lvert V \rvert}{\log_2 k}$ | $2 \leq k < \lvert V \rvert$ |

cisely the complexity measures for the above models, and state precisely the problem of synchronization. In Section 3, the solution (the Synchronizer) is presented. It is also shown that in order to implement efficiently this Synchronizer, one should solve a certain combinatorial graph problem, referred to as *partition problem* . An algorithmic solution to this problem is given in Section 4. In Section 5, we present the lower bound on complexity of synchronization. Finally, in Section 6, we summarize our results and compare them with the existing ones.

## 2. The problem

### 2.1. The Model

In this paper, we are dealing with distributed algorithms in two network models. The *asynchronous* network is a point-to-point (store-and-forward) communication network, described by an undirected *communication graph* $(V,E)$ where the set of nodes $V$ represents processors of the network and the set of links $E$ represents bidirectional non-interfering communication channels operating between them. No common memory is shared by the node's processors and each node has a distinct identity. Each node processes messages received from its neighbors, performs local computations, and sends messages to its neighbors. All these actions are assumed to be performed in *zero time*. All the messages have a fixed length, and may carry only a bounded amount of information. Each message sent by a node to its neighbor arrives to it within some finite but unpredictable time. This model appears also in [S-82],[S-83],[G-82], etc.

In the *synchronous* network, messages are allowed to be sent only at integer times, or *pulses*, of a global *clock*. Each node has an access to this clock. At most one message can be sent over a given link at a certain pulse. The delay of each link is at most one time unit of the global clock.[3]

The following complexity measures are used to evaluate performances of algorithms operating in the two network models above. The *Communication Complexity, C* is the total number of messages sent during the algorithm. The *Time Complexity, T* of a *synchronous* algorithm is the number of pulses, passed since its starting time until its termination. *The Time Complexity, T* of an *asynchronous* algorithm is the worst-case number of time units from start to the completion of the algorithm, assuming that the propagation

[3] it can be easily seen that the network where all delays are exactly *exactly* 1 is as powerful as the one where delays are *at most* 1.

delay[4] and the inter-message delay[5] of each link is *at most* one time unit. This assumption is introduced only for the purpose of performance evaluation; the algorithm must operate correctly with arbitrary delays.

A typical phenomena in communication networks is the trade-off between communication and time.

## 2.2. The Goal

Our main goal is to design an efficient *Synchronizer* which enables any synchronous algorithm to run in any asynchronous network. For that purpose, the Synchronizer generates sequences of "clock-pulses" at each node of the network, satisfying the following property: a new pulse is generated at a node only after it receives all the messages of the synchronous algorithm, sent to that node by its neighbors at the previous pulses. 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 node cannot know which messages were sent to it by its neighbors and there are no bounds on link delays. Thus, the above property cannot be achieved simply by waiting "enough time" before generating the next pulse, as it might be possible in a network with bounded delays. However, it may be achieved if additional messages are sent for the purpose of synchronization.

The total complexity of the resulting algorithm depends on the overhead introduced by the Synchronizer. Let us denote the communication and time requirements added by a Synchronizer $v$ per each pulse of the synchronous algorithm by $C(v)$ and $T(v)$, respectively. Synchronizers may need an initialization phase, which must be taken into account in case that the algorithm is performed only once. Let us denote by $C_{init}(v)$, $T_{init}(v)$, the complexities of the initialization phase of the Synchronizer $v$. In summary, the complexities

[4] difference between arrival time and transmission time

[5] difference between transmission times of two consecutive messages on the same link

of an original synchronous algorithm $S$ and the asynchronous algorithm $A$ resulting from the combination of $S$ with the Synchronizer, are related as: $C_A = C_S + T_S \cdot C(\upsilon) + C_{init}(\upsilon)$ and $T_A = T_S \cdot T(\upsilon) + T_{init}(\upsilon)$, where $C_A, T_A$ and $C_S, T_S$ are the communication and time complexities of algorithms $A$ and $S$, respectively. A Synchronizer $\upsilon$ is "efficient" if all the parameters $C(\upsilon)$, $T(\upsilon)$, $C_{init}(\upsilon)$, $T_{init}(\upsilon)$ are "small enough". The first two parameters are much more important since they represent the overhead per pulse.

## 3. The Solution

### 3.1. Outline of a number of Synchronizers

The main result of this Section is denoted as Synchronizer $\gamma$. It is a combination of two simple Synchronizers, denoted as Synchronizer $\alpha$ and Synchronizer $\beta$, which are, in fact, generalizations of the techniques of [G-82]. Synchronizer $\alpha$ is efficient in terms of time but wasteful in communication while Synchronizer $\beta$ is efficient in communication but wasteful in time. However, we manage to combine these Synchronizers in such a way that the resulting Synchronizer is efficient *both* in time and communication. Before describing these Synchronizers, we introduce the concept of safety.

A node is said to be *safe* w.r.t. a certain pulse if each message of the synchronous algorithm sent by that node at that pulse has already arrived to its destination. (Remember that messages are sent only to neighbors.) After execution of a certain pulse, each node eventually becomes safe (w.r.t. that pulse). If we require that an acknowledgement is sent back whenever a message of the algorithm is received from a neighbor, then each node may detect that it is safe whenever all its messages have been acknowledged. Observe that the acknowledgements do not increase the asymptotic communication complexity, and each node learns that it is safe a constant time after it entered the new pulse.

A new pulse may be generated at a node whenever it is guaranteed that no message sent at the previous pulses of the synchronous algorithm may arrive at that node in the future. Certainly, this is the case whenever all the neighbors of that node are known to be safe w.r.t. the previous pulse. It only remains to find a way to deliver this information to each node with small communication and time costs. We present now the Synchronizers $\alpha$, $\beta$ and $\gamma$ mentioned above.

*Synchronizer $\alpha$:* Using the acknowledgement mechanism described above, each node detects eventually that it is safe, and then reports this fact to all its neighbors. Whenever a node learns that all its neighbors are safe, a new pulse is generated.

Complexities of Synchronizer $\alpha$ in communication and time are $C(\alpha)=O(|E|)=O(|V|^2)$ and $T(\alpha)=O(1)$, respectively, since one (additional) message is sent over each link in each direction, and all the communication is performed between neighbors.

*Synchronizer $\beta$:* This Synchronizer needs an initialization phase, in which a leader $s$ is chosen in the network and a spanning tree of the network, rooted at $s$, is constructed. The Synchronizer itself operates as follows. After execution of a certain pulse, the leader will eventually learn that all the nodes in the network are safe; at that time it broadcasts a certain message along the tree, notifying all the nodes that they may generate a new pulse. The above time is detected by means of a certain communication pattern, referred to in future as *convergecast,* which is started at the leaves of the tree and terminates at the root. Namely, whenever a node learns that it is safe and all its descendants in the tree are safe, it reports this fact to its father.

Complexities of Synchronizer $\beta$ are $C(\beta)=O(|V|)$ and $T(\beta)=O(|V|)$, because all the process is performed along the spanning tree. Actually, the time is proportional only to the height of this tree which may reach $|V|-1$ in the worst case.

*Synchronizer $\gamma$:* This Synchronizer needs an initialization phase, in which the network is partitioned into *clusters*. The partition is defined by any spanning forest of the communication graph $(V,E)$ of the network. Each tree of the forest defines a cluster of nodes, and will be referred to as an *intra-cluster tree*. Between each two neighboring clusters, one *preferred link* is chosen, which will serve for communication between these clusters. Inside each cluster, a *leader* is chosen which will coordinate the operations of the cluster via the intra-cluster tree. We say that *a cluster is safe* if all its nodes are known to be safe.

Synchronizer $\gamma$ is performed in two phases. In the first phase, Synchronizer $\beta$ is applied separately in each cluster along the intra-cluster trees. Whenever the leader of a cluster learns that its cluster is safe, it reports this fact to all the nodes in the cluster as well as to all the leaders of the neighboring clusters. Now, the nodes of the cluster enter the second phase, in which they wait until all the neighboring clusters are known to be safe and then generate the next pulse (as if Synchronizer $\alpha$ is applied among clusters).

Let us, however, give a more detailed description of this Synchronizer. In order to start a new pulse, a cluster leader broadcasts along the tree a *PULSE* message, which triggers the nodes which receive it to enter the new pulse. After terminating its part in the algorithm, a node enters the first phase of the Synchronizer, in which *SAFE* messages are convergecasted along each intra-cluster tree, like in Synchronizer $\beta$. This process is started by the leaves which send *SAFE* message to fathers whenever they detect that they are safe. Whenever a non-leaf node detects that it is safe and the above message *SAFE* has been received from each of its sons, then, in case it is not the leader, it sends *SAFE* to its father. Otherwise, if it is the leader, it learns that its cluster is safe and reports this fact to all neighboring clusters by starting the broadcast of the *CLUSTER−SAFE* message. Each node forwards this message it to all its sons and along all incident preferred links.

Now, the nodes of the cluster enter the second phase. In order to determine the time, when all the neighboring clusters are known to be safe, a standard convergecast process is performed. Namely, a node sends *READY* message to its father whenever all the clusters neighboring with it or with any of its descendants are known to be safe. This situation is detected by a node whenever it receives *READY* messages from all its sons and *CLUSTER−SAFE* messages from all the incident preferred links and from its father[6].

This process is started at the leaves and is finished whenever the above conditions are satisfied at the leader of the cluster. At that time, the leader of the cluster knows that all the neighboring clusters as well as its own cluster, are safe. Now, it informs the nodes of its cluster that they can generate the next pulse by starting the broadcast of the *PULSE* message. The precise algorithm performed by each node is given in the next sub-section.

*Complexities of the Synchronizer* $\gamma$*:* Let us denote by $E_p$ the set of all the tree links and all the preferred links in a partition $P$. Also, denote by $H_p$ the maximum height of a tree in the forest of $P$. It is easy to see that at most 4 messages of the Synchronizer are sent over each link of $E_p$; thus $C(\gamma)=O(|E_p|)$. It requires $O(H_p)$ time for each cluster to verify that it is safe and additional $O(H_p)$ time to verify that all the neighboring clusters are safe; thus $T(\gamma)=O(H_p)$. This observation motivates the following combinatorial **Partition Problem**:

*Find a partition $P$ with* **both** *$E_p$ and $H_p$* being small.

Observe that the above parameters depend only on the structure of the forest. It does not really matter how the preferred links are chosen in the partition, since their total number equals to the total number of pairs of neighboring trees.

[6] Message from father is needed to ensure that the cluster, to which the node belongs, is safe

Solution of the above problem turns out to be a non-trivial task even for a centralized algorithm. We may mention that it is relatively easy to find partitions with one of the parameters being small. Say, if each node forms a cluster, then $H_p = 0$ and $E_p = |E|$. Also, by taking the whole graph to be a single cluster, whose intra-cluster tree is a BFS tree w.r.t. some node, we achieve $E_p = |V|$ and $H_p = \Theta(D)$, where $D$ is the diameter of the network; in the worst-case $D = |V| - 1$. With these partitions, we actually obtain the aforementioned Synchronizers $\alpha$ and $\beta$, respectively.

Using the partition algorithm of the next section, we achieve $E_p \leq k\,|V|$ and $H_p \leq \frac{\log_2 |V|}{\log_2 k}$. Here, $k$ is a parameter of the partition algorithm, may be chosen arbitrarily in the range $2 \leq k < |V|$. By increasing $k$ in the range from 2 to $|V|^{\frac{1}{10}}$, $C(\gamma)$ increases from $O(|V|)$ to $O(V^{1.1})$ while $T(\gamma)$ decreases from $O(\log_2 |V|)$ to $O(10)$. The particular choice of $k$ is up to the user, depending on relative importance of saving in communication and time in a particular network. This choice may also depend on the topology of the network, since, in fact, no matter which partition is used, $C(\gamma) \leq O(|E|)$ and also $T(\gamma) \leq O(D)$, provided that each intra-cluster tree is a BFS tree w.r.t. some node. For example, in a sparse network, where $|E| = O(|V|)$, we choose $k = |V|$, while in a full network, where $D = 1$, we choose $k = 2$. This is because in a sparse (full) network communication (time) is small anyway.

The distributed implementation of the partition algorithm requires $C_{init}(\gamma) = O(k\,|V|^2)$ and $T_{init}(\gamma) = O(|V| \frac{\log_2 |V|}{\log_2 k})$. Applying Synchronizer $\gamma$ to the synchronous BFS and Maximum Flow algorithms of Table 1 yields new efficient asynchronous algorithms whose complexities are mentioned in Table 2; they include the overhead of the above distributed partition algorithm.

## 3.2. Formal description of the Synchronizer $\gamma$

Here, we give a formal algorithm performed by each node $i$ of the network. The algorithm specifies the actions taken by node $i$ in response to messages arriving to it from its neighbors. Say, "For *PULSE* from $q$ do....." means: "After receipt of *PULSE* message from neighbor $q$, perform".

*Messages of the algorithm*

| | |
|---|---|
| *ACK* = | acknowledgement, sent in response to the message of the synchronous algorithm. |
| *PULSE* = | the message, which triggers the 'clock-pulse'. |
| *SAFE* = | message sent by a node to its father when all the descendants are known to be safe. |
| *CLUSTER −SAFE* = | message sent by a node to its sons and over preferred links whenever its cluster is known to be safe. |
| *READY* = | message, sent by a node to its father whenever all the clusters connected by preferred links to descendants of the node are known to be safe. |

*Variables kept at node i*

*The variables provided by the partition algorithm:*

| | |
|---|---|
| $Neighbors(i)$ = | The set of neighbors of node $i$. |
| $Father(i)$ = | The father of $i$ in the intra-cluster spanning tree. For the leader of the cluster, $Father(i) = i$. |
| $Sons(i)$ = | The sons of $i$ in the intra-cluster spanning tree. |
| $Preferred(i)$ = | Set of pointers to preferred links incident to $i$. For each such link $(i-j)$, node $j$ is included in $Preferred(i)$. For convenience, we assume that $Father(i) \in Preferred(i)$. |

*Variables used in the algorithm:*

| | |
|---|---|
| $Safe(i,q)$ = | A binary flag, kept for all $q \in Sons$ which equals 1 if the *SAFE* message from $q$ was received in the present pulse. ($Safe(i,q)$=0,1). |
| $Ready(i,q)$ = | A binary flag, kept for all $q \in Sons(i)$ which equals 1 if the *READY* message from $q$ was received at the present pulse. ($Ready(i,q)$=0,1). |
| $Dif(i,j)$ = | The counter, kept for each $j \in Neighbors(i)$. It shows the difference between the number of messages of the synchronous algorithm sent from $i$ to $j$ and the number of acknowledgements *ACK* received from $j$ at $i$. At the beginning of a pulse, $Dif(i,j)$=0. ($Dif(i,j)$=0,1,2...) |
| $cluster-safe(i,j)$ = | A binary flag, kept for each $j \in Sons(i) \cup Father(i)$, which equals 1 if the *CLUSTER −SAFE* message was received from $j$ at the present pulse.($cluster-safe(i,j)$=0,1). |

*Procedures used in the algorithm*

*Safe −Propagation* : Procedure which convergecasts the *SAFE* messages.
*Ready −Propagation* : Procedure which convergecasts the *READY* messages.

**The algorithm for node i**

```
For PULSE message do
    Trigger execution of the next pulse of the synchronized protocol P
```

```
        for all q ∈ Sons(i) do
            safe ( i,q ) ← 0 /* wait for SAFE from q */
            send PULSE to q
        end
        for all j ∈ Neighbors(i), set Dif(i,k) ← 0
        for all k ∈ Preferred(i), set cluster-safe(i,k) ← 0
end

For message of the synchronized algorithm S sent from i to j do
        Dif(i,j) ← Dif(i,j) + 1
end

For message of the protocol P arriving at i from j do
        send ACK to j
end

For ACK from j do
        Dif(i,j) ← Dif(i,j) - 1
        Call Safe-Propagation
end

Whenever the actions performed at a certain pulse have been completed, do
        Call Safe-Propagation
end

Safe-Propagation: Procedure
/* This procedure is called whenever there is a chance that node i as well as all its
descendants are safe. In this case, SAFE message is sent to father */
        if Dif(i,j) = 0 for all j ∈ Neighbors(i) and safe(i,q) = 1 for all q ∈ Sons (i)
        then do
            if Leader(i) ≠ i then send SAFE to Father(i)
            else send CLUSTER-SAFE to itself
            /* cluster leader i learned that its cluster is safe and starts broadcast
            of CLUSTER -SAFE message */
        end
end

For SAFE from q do
        safe (i,q) ← 1
        Call Safe-Propagation
end

For CLUSTER-SAFE message from j do
        if j ∈ Preferred(i) then cluster-safe (i,j) ←1
        /* The cluster to which j belongs is safe */
        if j ∈ Father(i) then do
        /* The cluster to which i itself belongs is safe */
            for all q ∈ Sons (i) do
                send CLUSTER-SAFE to q
                ready (i,q) ← 0
                /* wait for READY from q */
            end
            for all k ∈ Preferred(i), sent CLUSTER-SAFE to k
            /* inform the neighboring cluster that your cluster is safe */
        end
        Call Ready-Propagation
end

For READY from q do
```

```
        ready (i,q) ← 1
        Call Ready-Propagation
end

Ready-Propagation: Procedure
/* This procedure is called whenever there is a chance that all the clusters neigh-
boring with node i and all its descendants are safe */
        if cluster-safe(i,j)=1 for all j ∈ Preferred(i) and ready (i,q) = 1
        for all q ∈ Sons (i) then do
                if Leader (i)≠i then send READY to Father (i)
                /* i is not a leader */
                else send PULSE to itself
                /* i is a leader and it has learned that its own cluster as well as all the
                neighboring clusters are safe. Thus, it triggers the execution of the
                next pulse of the synchronous algorithm in its cluster */
        end
end
```

## 4. The Partition Algorithm

### 4.1. The Outline

Intuitively, the idea of the following algorithm is to choose each cluster as a maximal subset of nodes whose diameter does not exceed the logarithm of its cardinality. This guarantees that the total number of the neighboring cluster pairs is linear and the maximum cluster diameter is logarithmic in the number of network nodes.

The algorithm proceeds, constructing the clusters one by one. Throughout the algorithm, the "remaining graph" denotes the sub-network induced by the nodes which were not yet joined to clusters. The basic stage of the algorithm is a follows: a node in the remaining graph is chosen as a new cluster leader and then a cluster is formed around this node. This stage is repeated until there are no more nodes in the remaining graph.

A number of procedures are used in the algorithm. The *Cluster-Creation* procedure creates a cluster in the remaining graph around a given leader node. The *Search-for-Leader* procedure searches the remaining graph and chooses a new cluster leader in case that the remaining graph is not empty. The *Preferred-Link-Election* procedure chooses the preferred links outgoing from a cluster. Now, we describe each of the the procedures above with more details and finally give the code of the whole partition algorithm.

## 4.2. Cluster-Creation Procedure

This procedure is the heart of the partition algorithm. Basically, it operates as follows. A node chosen as a new cluster leader triggers execution of the BFS algorithm w.r.t. itself in the remaining graph. Each new BFS layer joins the cluster until the number of nodes in a certain layer is less than $k-1$ times the total number of nodes contained in *all* the previous layers; at that time the procedure terminates, and the Search-for-Leader procedure is called.

The set of all the nodes in the above layer (the first one which was *not* joined to the cluster) is called the *rejected layer* of that cluster. The intra-cluster tree of the resulting cluster is the BFS tree w.r.t. the leader.

**Theorem 1:** Suppose that the clusters are constructed as described above. Then, the parameters $E_p$, $H_p$ of the resulting partition satisfy:

$H_p \leq \log_k |V| = \frac{\log_2 |V|}{\log_2 k}$ and $E_p \leq k\,|V|$.

**Proof** : Clearly, $H_p$ equals to the maximum number of layers joined to a cluster. The bound on $H_p$ follows immediately by observing that the total number of nodes contained in a cluster must be multiplied by $k-1$ at least with each additional layer. It remains to prove the second bound on $E_p$. Observe that whenever creation of a cluster containing $q$ nodes is completed, the number of nodes in its rejected layer cannot exceed $(k-1)q$ (otherwise, the rejected layer should have been joined to the cluster). Thus, the number of preferred links connecting that cluster to clusters which are created later is at most $(k-1)q$. For each preferred link, connecting two clusters, let us charge the cluster which was created earlier. Summing the charge over all the clusters, it follows that the total number of preferred links is at most $(k-1)|V|$. Clearly, the total number of tree links cannot exceed $|V|$. Thus $E_p \leq k\,|V|$. □

Now, we describe a distributed implementation of the algorithm above. Basically, it is just the distributed Breadth-First-Search (BFS) algorithm in the remaining graph. It constructs the BFS tree layer after layer. This algorithm is derived from a synchronous one by means of a synchronization process, which is very similar to the Synchronizer B. The only difference is that synchronization is performed on the part of the BFS tree, constructed by the algorithm until now. This is essentially the "Algorithm D1" of [G-82].

At the beginning of a pulse number $P$, $P-1$ layers of the BFS tree are already constructed. The purpose of pulse number $P$ is to join layer $P$ to the tree, or to reject it and to terminate the process of cluster creation. The final decision about joining of layer $P$ to the cluster depends on the total number of nodes at this layer, joining to the cluster at this pulse together with $j$. and it will be made whenever the above number will be known. The initial assumption is that the last layer *is* joined to the cluster.

In order to trigger the execution of the next pulse, the leader $l$ of the cluster broadcasts a *PULSE* message over the existing tree. Each internal node at layer $P' \leq P-1$ propagates the *PULSE* message received from its father to all its sons, until it reachs nodes of the last layer $P-1$. Upon receipt of this message, node $i$ at last layer $P-1$ propagates the message $LAYER\{P-1,l\}$ to all neighbors informing them that it belongs to layer number $P-1$ of the cluster, governed by $l$.

Upon receipt of such message, a neighbor $j$ which was not yet joined to any cluster joins the layer $P$ of the cluster of $l$, and chooses $i$ as its father in the intra-cluster tree. In any case, acknowledgement $ACK\{bit\}$ is sent by $j$ back to $i$, carrying $bit=1$ in case that $i$ was chosen as father of $j$ and $bit=0$ otherwise.

To compute the number of new nodes and to ensure that all the nodes at layer $P$ were counted, a convergecast process is performed. Each node waits until the number of its descendants at layer $P$ is known and then reports this number to its father, inserting it into *COUNT*{*} message. A node at layer $P-1$ does it whenever *ACK* messages have been collected from all neighbors, and an internal node at layer $P' < P-1$ does it whenever the above reports have been received from each of its sons. The process terminates when the leader node knows the total number of nodes at layer $P$. If this number is high enough, i.e. at least $k-1$ times greater than the present number of nodes in the cluster, then the next pulse $P+1$ is started, and by this, the nodes of the last layer $P$ are assured that they are finally joined to the cluster. Otherwise, the leader $l$ broadcasts along the existing tree a *REJECT* message which notifies nodes of layer $P$ that they are *rejected* from the cluster. This message also means that the "father-son" relation, tentatively established between nodes of layers $P-1,P$ is now canceled.

Here, the Cluster-Creation procedure terminates, and the Search-for-Leader procedure is called. Observe that at this stage, each node knows about itself and each of its neighbors, whether they were already joined to some cluster, and if so, what is its leader identity. (The neighbors which were not yet joined to clusters are those neighbors from which *LAYER* message was not yet received.) Nodes joined to clusters know their father and sons in the tree. Also, no control message of the procedure is in transient in the network.

## 4.3. Search-for-Leader Procedure

Basically, this procedure operates as follows. After a certain cluster $C$ is formed, its rejected layer is examined. If it is not empty, then a node in this layer is chosen as a new leader. In case that the rejected layer of $C$ is empty, the center of activity backtracks to the cluster from which $C$ itself was discovered and the above procedure is repeated there. An easy way to conceive the Search-for-Leader procedure is to consider an auxiliary directed graph, whose nodes are the clusters and a link ( $i \rightarrow j$) means that cluster $j$ was discovered from cluster $i$. It is easy to see that this graph is a Depth-

First-Search tree [E-79] and the search process corresponds to a number of backward steps on that tree followed by one forward step.

This procedure is initiated at some cluster leader $l$, which starts execution of a certain Cluster-Search Subroutine. It determines whether the rejected layer of the cluster is non-empty. In order to trigger the subroutine, the leader node $l$ broadcasts a *TEST* message along the intra-cluster tree. The election is performed by means of a convergecast process which is very similar to the process of counting of the nodes in the last layer, used in the Cluster-Creation Procedure. A node $i$ at the last layer examines the set of its neighbors belonging to the remaining graph. In case that this set is non-empty the node with minimum identity in this set is chosen to be the local candidate at that node. Otherwise, the local candidate is chosen to be *nil*. Then a *CANDIDATE*{*} message is sent to father, containing the value of local candidate. An internal node sets its local candidate to the minimum value, contained in *CANDIDATE* messages received from sons, considering *nil* to be higher than any node's identity. Whenever these messages have been received from all the sons, a node reports the value of its candidate to its father. Upon termination of this subroutine the local candidate at the leader is *nil* if the rejected layer is empty. Otherwise it equals the minimum-identity node in that layer.

After termination of the subroutine, the center of activity of the search moves to another cluster, depending on the result of the search in the present cluster. In case that the rejected layer of present cluster is not empty, the node $k$ with minimal identity number in this layer is notified that it becomes a new cluster leader. For that purpose, *NEW−LEADER*$\{k\}$ message is broadcasted along the tree, until it reachs the node $k$ itself. Upon receipt of this message, the new leader $k$ remembers the node, from which it has arrived as its *Cluster −Father* and then starts the creating its own cluster. Otherwise, if the rejected layer is empty, the center of activity backtracks to the cluster from which the present cluster was discovered, in case that such cluster exists. For that purpose, a *RETREAT* message is sent from $l$ to its *Cluster* $-$*Father*. This message is further propagated by each node to its father until it reaches the cluster leader and the search procedure is repeated from that cluster. In case that the present cluster has no cluster-father, i.e. it was the very first cluster to be created, the whole Search-for-Leader Procedure terminates since the remaining graph must be empty.

## 4.4. Preferred-Link-Election Procedure

Basically, this procedure operates as follows. First, distinct weights are assigned to all the links. The weight of a link $(i,j)$ is the pair $(\min(i,j),\max(i,j))$ and these pairs are ordered lexicographically. Then, the preferred link between two neighboring clusters is chosen as the minimum-weight link whose endpoints belong to these clusters. This election rule enables each cluster to choose separately the preferred links incident to it, since it guarantees that a link connecting two clusters is chosen either at both or at none of these clusters. The election inside a certain cluster is performed whenever the center of activity backtracks from that cluster in the above Search-for-Leader Procedure. Observe that at that time, all the nodes in the neighborhood have already been joined to clusters.

The procedure is triggered by an *ELECTION* message, which is broadcasted by the leader along the tree. Election of the preferred edges is performed by means of a standard convergecast process. Each node transfers to its father the 'election list' *LIST*{*} prepared by it together with all its descendants in the intra-cluster tree. This list specifies, for each cluster neighboring with one of the above nodes, the minimal-weight link outgoing to it. Note that this list has a variable length.

The leaves of the intra-cluster tree start the process by sending their local lists to their fathers. An internal node merges its own list with the lists received from sons, while deleting redundant entries, resulting from this merging (i.e. two links outgoing to the same cluster). Whenever the above lists were received from all the sons, an internal node sends its own list to its father. This process terminates whenever the list at the leader is merged with lists of all its sons. Now, the leader broadcasts the final list along the intra-cluster tree. For a node receiving the above final list, the initialization phase has terminated and it may start execution of the first pulse of the synchronous algorithm right away.

## 4.5. The complexity of the Partition Algorithm

In order to initialize the above partition algorithm, we must choose the leader of the first cluster. For that purpose, an arbitrary node must be elected as a *leader* of the network. The algorithm of [GHS-83] can perform this task and its complexities are $C_{MST}=O(|E|+|V|log_2|V|)=O(|V|^2)$ and $T_{MST}=O(|V|log_2|V|)$.

Let us denote by $C_{BFS}$ ($T_{BFS}$) $C_{DFS}$ ($T_{DFS}$) and $C_{ELEC}$ ($T_{ELEC}$) the overall communication (time) requirements of the Cluster-Creation, Search-for-Leader, and the Preferred-Link- Election procedures. Clearly, $C_{init}(\gamma)=C_{MST}+C_{BFS}+C_{DFS}+C_{ELEC}$ and $T_{init}(\gamma)=T_{MST}+T_{BFS}+T_{DFS}+T_{ELEC}$. We now show that:

(1) $C_{BFS}=O(|E|+|V|\log_k|V|)$ $T_{BFS}=O(|V|)$.

(2) $C_{DFS}=O(|V|^2)$ $T_{DFS}=O(|V|\log_k|V|)$.

(3) $C_{ELEC}=O(k|V|^2)$ $T_{ELEC}=O(|V|\log_k|V|)$.

These equations imply that $C_{init}(\gamma)=O(k|V|^2)$ and $T_{init}(\gamma)=O(|V|\frac{\log_2|V|}{\log_2 k})$.

### 4.5.1. *Cluster-Creation*

At each cluster, this procedure is applied exactly once and it consists of at most $\log_k|V|$ pulses. At each pulse, one *PULSE* and one *COUNT* message pass through each link of the intra-cluster tree. One *LAYER* and *ACK* message is sent over each link exactly once throughout the whole algorithm. It yields a total communication cost of $C_{BFS}=O(|E|+|V|\log_k|V|)$. Consider now a cluster with $n$ nodes whose intra-cluster tree has height $h \leq \log_k n$. Each pulse takes $h$ time units and the total number of pulses is $h$. Thus, the total time spent in forming this cluster and deleting $n$ nodes from the remaining graph is $O(\log_k^2 n)$. Since for all integer $n$, and for all $k \geq 2$, $\log_k^2 n \leq \frac{9}{8} n$, the total time investment is linear, i.e. $T_{BFS} = O(|V|)$.

### 4.5.2. *Search-for-Leader*

In this part, the center of activity moves along the Depth-First-Search Tree in the Cluster Graph. Whenever a center of activity arrives at a certain cluster, this cluster is 'examined' by the Cluster-Search Subroutine. This subroutine involves broadcast of *TEST* messages and convergecast of *CANDIDATE* messages. Afterwards, the center of activity moves by means of *NEW–LEADER* or *RETREAT* message to another cluster. The whole process described above will be referred to as *move*. Observe that move is performed in whole along intra-cluster trees. Its complexities are $C_{move} = O(|V|)$ and $T_{move} = O(\log_k |V|)$. In these moves, each edge of the DFS tree of the cluster graph is transversed exactly twice and the total number of edges is the total number of clusters minus 1, which can no exceed $|V|-1$. It yields a total complexity $C_{DFS} = O(|V|) \times C_{move} = O(|V|^2)$ and $T_{DFS} = O(|V|) \times T_{move} = O(|V| \log_k |V|)$.

### 4.5.3. *Preferred-Link-Election*

This procedure is called once at each cluster and is performed along the intra-cluster spanning tree. To simplify the computations we will assume that, at each cluster, the elections of the preferred links are performed *sequentially*. In this case, it is easier to evaluate the complexities of the process since only constant-length messages are used. Recall that in the above Preferred-Link-Election Procedure, all the preferred links incident to a certain cluster were elected *at the same time*; this required variable-length messages and made the computations more complex. Certainly, the complexities may only increase as a result of this modification.

By "sequential" elections we mean that the preferred links, connecting the cluster to neighboring clusters, are elected one-by-one, by means of separate "elementary election processes". Each such elementary process is started only after the previous one has been completed and is performed along the intra-cluster tree, similarly to the original Procedure. Note, however, that election processes are performed in different clusters in *parallel* and thus the maximum election time in a *single* cluster determines the time complexity of the procedure.

An elementary election process requires $C_{elem} = O(|V|)$ and $T_{elem} = O(\log_k |V|)$. This elementary process is applied in total at most $(k-1)|V|$ times, according to the maximum possible number of preferred links. Thus, the total communication complexity is bounded by $C_{ELEC} = O(k|V|^2)$. Since the number of times that the elementary election process is performed in a certain cluster cannot exceed the total number of nodes, $|V|$, then $T_{ELEC} = O(|V| \log_k |V|)$.

## 4.6. The formal presentation of the Partition Algorithm

**The variables and messages used in the algorithm**

*The input variables:*

| | |
|---|---|
| Neighbors(i) = | The set of neighbors at the node $i$ in the network. |

*The output variables:*

| | |
|---|---|
| *Father*(i) = | The father of $i$ in the intra-cluster tree. Initially, *Father*(i) = *nil*. |
| *Sons*(i) = | The sons of $i$ in the intra-cluster tree. Initially, *Sons*(i) = {Ø}. |
| *Preferred*(i) = | Set of pointers to preferred links incident to $i$. For each such link (i-j), node $j$ is included in *Preferred*(i). Initially, *Preferred*(i) = {Ø}. |
| *Leader*(i) = | The identity of the leader of the cluster, to which node $i$ belongs. Initially, *Leader*(i) = *nil*. |
| *Leader*(i,j) = | The estimate of $i$ about *Leader*(j), kept for each $j \in Neighbors(i)$. Initially, *Leader*(i,j) = *nil*. |

*The global variables:*

| | |
|---|---|
| Remaining(i) = | The subset of Neighbors(i) which were not joined to clusters. Initially, Remaining(i) = Neighbors(i). |

*Messages used in the Cluster-Creation Procedure (BFS):*

| | |
|---|---|
| *PULSE* = | Message starting a new pulse of BFS |
| *LAYER* $\{j,q\}$ = | Message sent by a node belonging to layer number $j$ in a cluster whose leader is $q$ |
| *ACK* {x} = | Message sent in response to *LAYER*. Here, x is a binary flag, which equals 1 is the sender has chosen the receiver as its father |
| *COUNT* {c} = | Message sent by a node which has c new descendants in the tree. |
| *REJECT* = | Message informing the nodes of the last layer that they are rejected from the cluster and that the cluster-formation procedure has terminated |
| *REJECT−ACK* = | Acknowledgement for the above *REJECT* message |

*Variables used in the Cluster-Creation Procedure (BFS):*

| | |
|---|---|
| *Layer* (i) = | The layer of the intra-cluster tree to which $i$ belongs. Initially, *Layer*($i$) =*nil*. (*Layer*(i) =0,1,... $\log_k \lvert V \rvert$). |
| *Pulse* (i) = | The number of the present pulse. Initially, *Pulse* (i) = 0. (*Pulse* (i) =0,1,... $\lvert V \rvert -1$). |
| *ack* (i,j) = | The binary flag, kept for each $j \in Neighbors(i)$. which equals 1 if the *ACK* message from $q$ was received at the present pulse. (*ack* (i,q)=0,1). |
| *Count* (i) = | The number of new leaves, joined in the last pulse, whose ancestor is $i$. Initially, *Count* (i) = 0. (*Count* (i) = 0,1,... $\lvert V \rvert -1$). |
| *Total* (i) = | The total number of nodes in the cluster, accumulated until now. Initially, *Total* (i) =0. (*Total* (i) =0,1,... $\lvert V \rvert -1$). |
| *count* (i,q) = | A binary flag, kept for all $q \in Sons$ which equals 1 if the *COUNT* message from $q$ was received in the present pulse. (*count* (i,q)=0,1). |
| *reject−ack* (i,q) = | A binary flag, kept for all $q \in Sons(i)$ which equals 1 if the *REJECT−ACK* message from $q$ was received at the present pulse. (*reject−ack* (i,q)=0,1). |

*Messages used in the Search-for-Leader Procedure (DFS):*

| | |
|---|---|
| *NEW−LEADER* {i} = | Message informing that $i$ is a new cluster leader |
| *TEST* = | Message requiring the nodes to start election of the next cluster leader in the neighborhood of the cluster |
| *CANDIDATE* {c} = | Message including an identity c, which is a candidate for a new cluster leader |
| *RETREAT* = | Message used in the search of the remaining graph for backtracking from a cluster to its father in the cluster graph |

*The variables used used in the Search-for-Leader Procedure (DFS):*

| | |
|---|---|
| *Cluster−Father* (i) = | The neighbor $j$ from which node $i$ was chosen as a new cluster leader. Initially, *Cluster−Father* (i) = *nil*. |
| *Candidate* (i) = | The neighbor $j$ which node $i$ has chosen as a possible candidate for being a new cluster leader. Initially, *Candidate* (i)= *nil*. |
| *candidate* (i,q) = | A binary flag, kept for all $q \in Sons$ which equals 1 if the *CANDIDATE* message from $q$ was received in the present pulse. (*candidate* (i,q)=0,1). |

*Messages used for in the Preferred-Links-Election Procedure*

| | |
|---|---|
| *ELECT* = | Message requiring the nodes to start the election of the preferred links in the cluster |
| *LIST*{list} = | Message where "list" is a list of links, which are candidates for being preferred links |
| *FINAL−LIST*{list} = | Message carrying the final list of the prefeered links |

*Variables used in Preferred-Links-Election Procedure*

| | |
|---|---|
| *List*(i) = | the list of links, chosen by node *i* together with its descendants as possible candidate for being preferred links incident to a cluster. It has a format {[c,(k-q)],[b,(r-p)],....} where c,b are identities of neighboring clusters and (k-q),(r-p) are the preferred links to the above clusters. Initially, *List*(i)= {Ø}. The *MERGE* operation which can be performed with two lists of the above format first just joins these lists and then deletes the redundant entries, resulting from the join. |
| *list*(i,q) = | A binary flag, kept for all *q* ∈*Sons* which equals 1 if the *LIST* message from *q* was received in the present pulse. (*list*(i,q)=0,1). |

**The algorithm for node i**

```
Whenever notified about being chosen as a start node, do
    send NEW-LEADER{i} to itself
end

For NEW-LEADER{k} from j do
/* i is chosen as a new cluster leader */
    send NEW-LEADER{k} to all q ∈ Sons(i)
    if k ∈ Remaining(i) then send NEW-LEADER{k} to k
    if k =i and Leader(i)=nil then do
        /* i is notified for the first time that it was chosen as a new cluster
        leader */
        Cluster-Father(i) ← j
        Father(i) ← i
        Leader(i) ← i
        Layer(i) ← 0
        Pulse(i) ← 0
        send PULSE to itself.
        /* trigger the cluster creation  process around yourself */
    end
end

For PULSE message do
/* next pulse of the cluster creation  process */
    Pulse (i) ← k
    if Layer(i)< k then /* i is an internal node in the tree */
        for all q ∈ Sons(i) do
            send PULSE to q ;
            count (i,q) ← 0
        end
        if Sons(i) ={Ø} then send COUNT{0} to Father(i)
    else
    /* node i is belongs to the last BFS layer which is finally joined to the clus-
    ter */
        for all p ∈ Neighbors(i) do
```

```
                    send LAYER{Layer(i),Leader(i)} to p
                    ack(i,p) ← 0
          end
end

For LAYER{k,j} from q do
      Leader(i,q) ← j
      Drop q from Remaining(i)
      MERGE {k,(i-j)} to List(i)
      /* consider link (i-j) as a candidate to be a preferred link */
      if Father (i) = nil then do
      /* tentatively join the cluster */
            Leader(i) ← j
            Layer (i) ← k+1
            Father (i) ← q
            send ACK{1} to q
            /* inform q that it is your father */
      end
      else send ACK{0} to q
      /* i was already joined to some cluster */
end

For ACK{x} from q do
      ack(i,q ) ← 1
      if x = 1 then do /* q is a new son */
            join q to Sons(i)
            Count(i) ← Count(i) + 1
            /* counter of sons increased by 1 */
      end
      if ack(i,j) = 1 for all j ∈ Remaining (i) then
            send COUNT{Count(i)} to Father(i)
end

For COUNT{c} from j do
/* node j has c descendants in the last layer */
      count (i,j) ← 1
      Count (i) ← Count(i) + c
      if count (i,q) = 1 for all q ∈ Sons(i) then do
            if Leader(i) ≠ i then send COUNT{Count(i)} to Father(i)
            else do /* i is a leader */
                  if Count(i) ≥ Total (i) then do
                        /*continue creation of the cluster */
                        Total(i) ← Total(i) + Count(i)
                        Pulse(i) ← Pulse(i) + 1
                        send PULSE to itself
                        /* trigger the new pulse of cluster creation process */
                  end
                  else send REJECT to itself
                  /* reject the last layer; creation of the cluster is completed */
            end
end

For REJECT from q do /* Last layer is rejected */
      for all q ∈ Sons(i) do
            reject - ack (i,q) ← 0
            send REJECT to q
      end
      if Layer(i) = Pulse(i)+1 then Father(i) ← nil.
      /* i belongs to the last layer, which is now rejected */
```

```
        if Layer(i) = Pulse(i) then Sons(i) ← {Ø}
        /* i is in the last layer which will finally remain in the cluster */
        if Sons(i) = {Ø} then send REJECT-ACK to Father(i)
end

For REJECT-ACK from q do
        reject-ack (i,q) ← 1
        if reject-ack(i,j)=1 for all j ∈ Sons(i)
        then do
                if Leader (i) ≠ i then send REJECT-ACK to Father(i)
                else send TEST to itself
                /* if i is a leader then start looking for a new cluster leader */
        end
end

For TEST from q do
        Candidate(i) ← nil
        for all q ∈ Sons(i) do
                candidate(i,q ) ← 0
                send TEST to q
        end
        if Layer(i) = Pulse(i) then do /* i is in the external layer */
                if Remaining(i) ≠{Ø} then do
                        Candidate (i) ← min { k | k ∈ Remaining (i)}
                        /* choose a local candidate for the new cluster leader */
                        send CANDIDATE { Candidate(i) } to Father(i)
                end
        end
        if Sons(i)= {Ø} then send CANDIDATE { nil } to Father(i).
end

For CANDIDATE { c } from q do
        Candidate(i) ←min { Candidate (i), c }
        candidate(i,q) ← 1
        if candidate(i,j) = 1 for all j ∈ Sons(i)
        then do
                if Leader(i) ≠ i then send CANDIDATE { Candidate(i) } to Father(i)
                else do /* i is a leader */
                        if Candidate (i) = c ≠ nil then send NEW-LEADER { c } to itself
                        else do
                        /* all the nodes neighboring to your cluster already belong to
                        some clusters */
                                send ELECT to itself
                                /* trigger the procedure for election of preferred links in
                                your cluster */
                                if Cluster-Father(i) ≠ i then
                                /* backtrack in the cluster graph and continue search */
                                        send RETREAT to Cluster-Father(i)
                                /* else the remaining graph is empty and after the election
                                of preferred links is completed, the algorithm terminates */
                        end
                end
end

For RETREAT do
/* backtrack to the father of the cluster which will coordinate the search */
        if Leader(i) ≠ i then send RETREAT to Father(i)
        else send TEST to itself
        /* if i is a leader, then trigger the search in its cluster */
```

```
end

For ELECT from j do
    for all q ∈ Sons(i) do
        list(i,q ) ← 0
        send ELECT to q
    end
    if Sons(i) = {∅} then send LIST { List(i) } to Father(i) /* i is a leaf */
end

For LIST { AList } from q do
    list(i,q) ← 1
    MERGE AList to List(i)
    /* Merge AList with List(i) and then discard duplicate links emanating to the
    same cluster */
    if list(i,j) = 1 for all q ∈ Sons(i) then do
        if Leader(i) ≠ i then send LIST { List(i) } to Father(i)
        else send FINAL-LIST { List(i) } to itself
        /* i is a leader and List(i) is the final list containing all the preferred
        links */
        end
    end
end

For FINAL-LIST { AList } from p do
    for all j ∈ Remaining(i) do
        if [*,(i − j)] appears in AList, then join j to Preferred(i)
    end
    for all q ∈ Sons(i) send FINAL-LIST { AList } to q
    /* Now, the initialization phase has terminated for node i. It may trigger
    the first pulse of the synchronous algorithm right now */
end
```

## 5. Lower bound on complexity of Synchronization

Notice that Synchronizer $\gamma$ exhibits a trade-off between its communication and time complexities. To be more precise, $C(\gamma)=O(|V|^{1+\frac{1}{T(\gamma)}})$, while $T(\gamma)=\log_k V$ for any $2 \leq k < V$. A natural question is whether this trade-off is an optimum one, i.e. whether there exists another Synchronizer $\delta$, which is better than the Synchronizer $\gamma$ *both* in communication and in time. We give only a partial answer to this question. For particular networks, this might be true. However, we are able to show that there exist networks, for which the best possible improvements are within small constant factors, i.e. the worst-case trade-off of any Synchronizer $\delta$ is $C(\delta)=\Omega(|V|^{1+\frac{1}{T(\delta)}})$. This fact is formally stated in the following Theorem.

**Theorem 2:** For any integer $i$, there exist (infinitely many) networks $(V,E)$, in which any Synchronizer $\delta$ with $T(\delta) < i-1$ requires $C(\delta) > \frac{1}{4}|V|^{1+\frac{1}{i}}$.

**Proof** : In order to satisfy the condition imposed on Synchronizer, each node should generate new pulse only after receipt of all the messages sent to it in the previous pulse. Thus, in between each two successive pulses, there must be some information flow, provided by the control messages of the Synchronizer, between each pair of neighbors in the network. Without such information flow, a node cannot find out in finite time whether some message, sent in the previous pulse, is still in transit on a certain link or not. This follows from the fact that the network is completely asynchronous and the node does not know a priori which of the incident links carry messages of a certain pulse. The information flow between neighbors may pass through the link, connecting them (e.g. Synchronizer $\alpha$), or may go along alternative paths (e.g. along links of a spanning tree, like in Synchronizer $\beta$). For any fixed pair of neighbors in the network, the length (in the number of edges) of the shortest information-flow path between these neighbors is an obvious lower bound on time complexity of a particular Synchronizer. Among these lower bounds, we choose the maximum one, i.e. the maximum, over all pairs of neighbors in the network, of the length of the shortest information-flow path between these neighbors.

Formally, define the *girth* of a graph to be the length of the shortest cycle in that graph. We use the following Lemma in our proof.

**Lemma** : For each integer $i$, there exist (infinitely many) networks $(V,E)$ with girth $g \geq i$ and $|E| > \frac{1}{4}|V|^{1+\frac{1}{i}}$.

**Proof** : [B-78], page 104, Theorem 1.1 . □

For a particular choice of $i$, let $(V,E)$ be a network with girth $g \geq i$ and $|E| > \frac{1}{4}|V|^{1+\frac{1}{i}}$. For an arbitrary Synchronizer $\delta$ for $(V,E)$ let $\Gamma(\delta) \subset E$ be the set of edges which carry the information flow and let $d(\delta)$ be the maximum, over all $(i,j) \in E$, of the length of a shortest path between $i$ and $j$ in the induced graph $(V,\Gamma(\delta))$. From the previous paragraph it follows that $T(\delta) \geq d(\delta)$ and $C(\delta) \geq |\Gamma(\delta)|$.

If $C(\delta) \geq |E|$ then the Theorem follows because $|E| > \frac{1}{4} |V|^{1+\frac{1}{i}}$. Otherwise, if $C(\delta) < |E|$ then $|\Gamma(\delta)| < |E|$ which implies that there exists an edge $e \in E - \Gamma(\delta)$. The length of a shortest path in $(V,\Gamma(\delta))$ between the two endpoints of $e$ is at least $g-1$, since this path together with the edge $e$ form a simple cycle in $(V,\Gamma(\delta))$. Thus, $T(\delta) \geq d(\delta) \geq g-1 \geq i-1$ . □

## 6. Summary and comparison with existing works

In this paper we have studied the problem of simulation of the synchronous network by the asynchronous one. We have proposed a new simulation technique, referred to as "Synchronizer" and have proved that its communication-time trade-off is optimum within a constant factor.

Essentially, our Synchronizer is a new, simple methodology for designing efficient distributed algorithms in asynchronous networks. For the model in question, i.e. point-to-point communication network no such methodology was explicitly proposed in the literature. However, let us mention briefly some of the related works. The current work was directly inspired by [G-82]. In this pioneering work, Gallager introduced the notion of communication-time trade-off in distributed algorithms and proposed a number of "synchronization" techniques, which were used in distributed of Breadth-First-Search Algorithms. These elegant techniques are not Synchronizers in the sense of this paper since they are not general and cannot be applied to other algorithms. However, Synchronizer $\alpha$ and Synchronizer $\beta$ of this paper are natural generalizations of these techniques. It is worth mentioning that we have been able to improve Gallager's BFS algorithms using Synchronizer $\gamma$, which can be viewed as a combination of the two Synchronizers above.

In our paper we consider a point-to-point communication network where communication is is performed exclusively by message passing. Other researchers ([AFL-83], [Sch-82]) studied some issues related to synchronization under different distributed computation model, where any processor can communicate with any other processor via shared memory. The results of [AFL-83, Sch-82] are of no use in our context, since the underlying model and the problems in question are substantially different from ours. Let us, however, give a brief review of these works.

[AFL-83] proves that synchronous network has greater computational power than asynchronous one, assuming that only a bounded number of processors can access the same variable. [Sch-82] deals with synchronization of distributed programs and other "state-machine" applications; it is not concerned at all with complexity of algorithms. However, it addresses fault-tolerant issues, not addressed in the current paper. It is worth mentioning that some of basic concepts as well as some of the basic difficulties in this paper are quite similar to those mentioned in [Sch-82]. For example, the notion of "safe" in the current paper corresponds to the technique described in [Sch-82] of only using "fully acknowledged" messages when checking a message queue. Similar technique appears also in [L-78] in the mutual exclusion example. The notion of a "pulse" in the current paper corresponds to a "phase" in [Sch-82] (timestamps generated by a logical clock are used in [Sch-82] instead of pulse numbers). The condition imposed on pulses in the current paper is analogous to the monotonicity requirement of [Shn-83].

**Acknowledgements**

The author wishes to thank Reuven Bar-Yehuda for pointing his attention to the Shiloach-Vishkin Maximum Flow Algorithm. Yossi Shiloach and Alon Itai read the manuscript and made a number of helpful comments. Noga Alon has provided an alternative proof of the lower bound, and Mike Luby has helped in simplifying that proof.

## References

[A-83] B.Awerbuch, "Applications of the network synchronization for distributed *BFS* and *Max–Flow* algorithms", preprint, October 1983.

[AFL-81] E. Arjomandi, M.J. Fisher and N.A. Lynch, "A Difference in Efficiency Between Synchronous and Asynchronous Systems", *JACM*; Vol. 30. No. 3, July 1983, pp. 449-456.

[B-78] B.Bollobas, "Extremal Graph Theory", *Academic Press,* 1978.

[E-79] S. Even, "Graph Algorithms", *Computer Science Press,* 1979.

[EC-79] D. Eckstein,. "Parallel Processing Using Depth- Search and Breadth-First Search", Ph.D. Thesis, Dept. of Comp. Science, *University of Iowa,* Iowa City, 1977.

[G-82] R.G. Gallager, "Distributed Minimum Hop Algorithms", *M.I.T. Technical Report,* LIDS-P-1175, January 1982.

[GHS-83] R.G. Gallager, P.A. Humblet and P.M. Spira, "A Distributed Algorithm for Minimum Weight Spanning Trees", *ACM Trans. on Program. Lang. & Systems,* Vol. 5, pp. 66-77, January 1983.

[L-78] L.Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System" *CASM,* Vol. 21, No. 7, 1978.

[S-82] A. Segall, "Decentralized Maximum Flow Algorithms", *Networks,* Vol. 12, pp. 213-230, 1982.

[S-83] A. Segall, "Distributed Network Protocol", *IEEE Trans. on Information Theory,* Vol. IT-29, No. 1, January 1983.

[SV-82] Y. Shiloach and U. Vishkin, "An $O(n^2 \log n)$ Parallel MAX-FLOW Algorithm", *Journal of Algorithms,* Vol. 3, pp. 128-146, 1982.

[Sch-82] F.Schneider, "Synchronization in Distributed Programs" *TOPLAS,* Vol. 4, Number 4, pp. 125-148, April 1982.