New Models and Algorithms for Future Networks

Israel Cidon*, Inder Gopal, and Shay Kutten

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

Abstract

In future networks transmission and switching capacity will dominate processing capacity. In this paper we investigate the way in which distributed algorithms should be changed in order to operate efficiently in this new environment. We introduce a class of new models for distributed algorithms which make explicit the difference between switching and processing. Based on these new models we define new message and time complexity measures which, we believe, capture the costs in many high speed networks more accurately than traditional measures. In order to explore the consequences of the new models, we examine three problems in distributed computation. For the problem of maintaining network topology we devise a broadcast algorithm which takes $O(n)$ messages and $O(\log n)$ time in the new measure. For the problem of leader election we present a simple algorithm that uses $O(n)$ messages and $O(n)$ time. The third problem, distributed computation of a ″globally sensitive″ function, demonstrates some important features and trade-offs in the new models and emphasizes the differences with the traditional network model.

A preliminary version of this paper appeared in the 7th ACM Symposium on Principles of Distributed Computing, Toronto, Canada, August 1988. The last section of this paper appeared in the Proceedings of the 4th Workshop on Distributed Algorithms, Bari, Italy, September 1990.

* and department of Electrical Engineering, Technion, Haifa, Israel

1. Introduction

The advent of fiber optic media has pushed the transmission speed of communication links to over a Gigabit (10^9 bits) per second, representing a three to four order of magnitude increase over typical links in today's data networks [KMS87]. Today's packet switching nodes (e.g. ARPANET [MRR80]) implement most of the packet handling protocols in software executed in general purpose computers. Typically, thousands of instructions are needed for a computer to deal with a message [G87]. (We use the terms "packet" and "message" interchangeably.) The increase in the communication link speeds has not been matched by a corresponding increase in the processing capacity of the communication nodes. Thus, processing has become the bottleneck in today's communication systems.

Several previous works have noted this and have proposed new designs for networks to reduce the effect of this bottleneck [TW83, HA87, CG88, CGGG93]. New standards such as ATM (Asynchronous Transfer Mode) have evolved as a consequence of these new designs [B92]. A common aspect of all of the proposed new designs has been the introduction of dedicated and optimized fast packet switching hardware. This hardware is designed to perform the most common packet handling function, i.e. the function of switching or moving a packet from the input link to the appropriate output link as defined in the packet header. By attaching the switching hardware to a communication node it is possible to off-load most of the networking related processing burden from the software driven processor. Packets that only require to be switched from an input to an output link are handled by the switching hardware. Only packets that require more complex processing are delivered to the general purpose processor to be handled by software.

Note that we use the terms ″hardware″ and ″software″ because they represent the typical implementation techniques used in proposed designs. The real implication of the term ″hardware″ is that the functions embodied therein are performed in a specially designed machine in a performance optimized fashion. Conversely, ″software″ functions are not optimized in this fashion. A dedicated set of embedded processors, executing optimized microcode to perform switching operations count as ″hardware″ from this perspective. For the purpose of clarity of presentation, we will use the terms ″hardware″ and ″software″ throughout this paper.

We categorize the delay experienced by a packet forwarded from one node to another into three components: 1) *Transmission* delays experienced in traversing the communication link, 2) *Switching* delays experienced in the switching hardware and 3) *Processing* delays experienced in the processor in case that more complex functions must be invoked.

The bulk of traffic transported in such networks will consist of high capacity user-to-user (or application-to-application) transmissions of data, voice, image and video. Such traffic does not require complex processing in the intermediate nodes and consequently will travel only through the

switching hardware. Since future networks are expected to support large volumes of user-to-user traffic, it is likely that future node designers will continue to improve the capacity of switching hardware. Already, the hardware switching structures proposed in [HA87] have throughputs that approach a Terrabit (10^{12} bits) per second. In contrast with the heavy demand for switching resources, it will be mainly the distributed algorithms used to control and manage the network (the route computation, configuration management, etc.) that will use the processing resources. The generality of these tasks (compared with switching), their relative complexity, and the need to maintain flexibility and openness for changes will probably require these control and management functions to remain software procedures for a while. The traffic associated with these functions is typically lower in volume by several orders of magnitude than the user-to-user traffic. (A video connection sends about 10,000 packets per second [MASK88]. Distributed computing applications typically send no more than a few dozen packets per second [AKRP87].) Therefore, in contrast with switching capacity, communication node designers will not have the same incentive to increase processing capacity within the node. (The PARIS network prototype [CG88] uses a single nodal microprocessor to perform the distributed network control operations).

Thus, future networks are likely to be very different from current networks, with transmission and switching (hardware) capacity dominating processing (software) capacity and, consequently, software delays dominating hardware delays. The purpose of this paper is to explore how distributed algorithms should be changed in order to operate efficiently in this new environment. Traditional models used in the design of distributed algorithms do not differentiate between hardware functions and software functions. In other words, a message sent from one node to another is typically counted as contributing one unit towards message and time complexity whether it requires complex processing at that node or whether it requires simply to be relayed to another node without any further processing.

In this paper, we introduce a class of models wherein the difference between hardware and software functions is made explicit. We define the functions that can reasonably be performed in hardware, given our judgement of the the limitations of current technology in capabilities and cost. (For example, we assume that it will not be cost effective, efficient and flexible enough to implement complex network control functions directly in hardware.) By assigning the hardware functions different costs and delays from the software functions, we are able to construct a class of models that can better capture the realistic costs and delays in many new network environments. While it is possible to use a general parameterized cost model for certain cases, we find it useful to simplify the model in order to study a wider class of problems. In particular, we consider a special case where the hardware costs and delays are essentially negligible (subject to some constraints) compared to the corresponding software values.

In order to explore the consequences of the new model we examine three problems, two of which have been extensively studied in the literature. The first, which is of major practical impor-

tance in networks, is the problem of maintaining network topology. The best known solution to this problem was used in the ARPANET [MRR80], wherein each node periodically broadcasts local topology (its adjacent links′ states and loads). The broadcast takes $O(m)$ messages and $O(n)$ time, where m is the number of communication links and n is the number of nodes. (These complexities do not change under our new measures.) In this paper we present a new topology maintenance algorithm that, under the new measures, takes $O(n)$ "messages" and $O(\log n)$ time per broadcast. A tight lower bound for the $O(\log n)$ time is given, for a certain class of potential algorithms.

The second problem that we examine is the well-studied problem of distributed leader election [GHS83]. Here the nodes in the network are required to cooperate in order to elect one of them as the leader and every node must know whether or not it is the elected leader. Distributed leader election is important for organizing a network after faults have occurred, resolving deadlocks and conflicting decisions, managing distributed databases, etc.. We present a simple leader election algorithm that uses $O(n)$ "messages" and $O(n)$ time. (The message complexity of the traditional algorithms is $\Omega(n \log n)$ [B80, PKR84, KMZ84] under our new measure as well.)

The third problem is the distributed computation of a "globally-sensitive" function. Roughly speaking, such a function is one that depends on every one of the distributed inputs, though a more formal definition will be given later. For this problem, we use the general parameterized cost model and show how the structure of the *optimal* algorithm and its time complexity depends heavily on the relative delays of the hardware and software. Note that the traditional model is also an extreme case of the general parameterized cost model. An interesting observation from these results is that the new model does not degenerate to the traditional one even if the network is fully connected and each node can get to any other in a single hop.

We emphasize that our new model is not intended as a "definitive" model for future networks. Rather, it should be viewed as the first step and the main contribution of this paper is the demonstration and understanding of the deficiencies in the traditional model. Our paper also provides some further practical motivation for work done in somewhat different contexts. For example, in [DHSS84, PS87] importance is given to the number of distinct routes a message traverses rather than the number of actual message hops. Similarly, in [KO87] messages relayed through intermediate nodes are not counted in certain cases. Since the initial version of this paper (the first two problems in [CGK88] and the third problem in [CGK90]), several works have appeared which explore distributed algorithms for fast networks (see for example [BZ90, ACGK90, CS89]). In addition, the parameterized model of [CGK90] has been further extended and studied in [BK92, CKPS93].

The paper is structured as follows. In the next section we present and justify the new class of models and measures. We then present the new topology maintenance algorithm, together with the

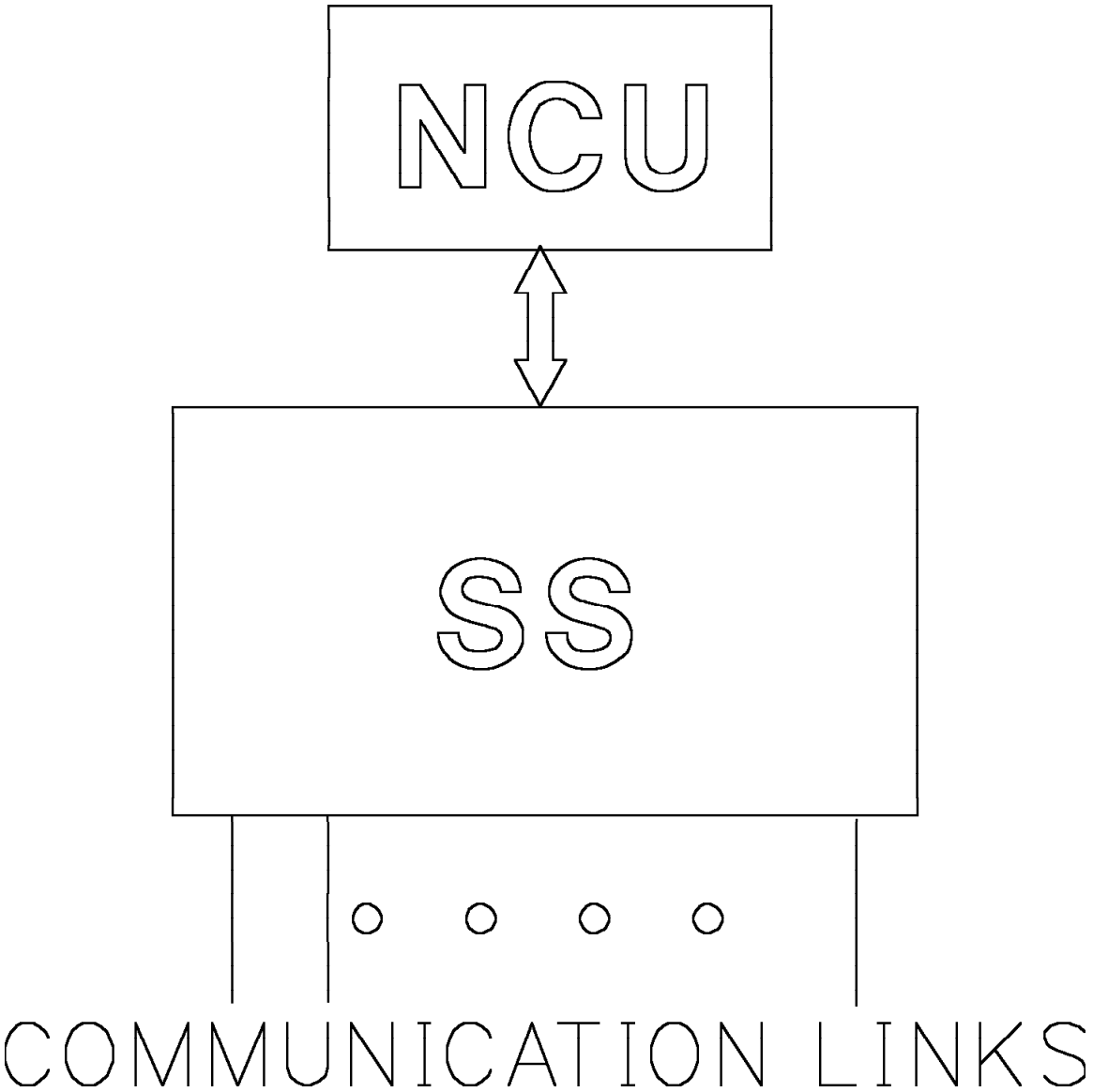

Figure 1. A Node Structure.

proof of the lower bound (Section 3). In Section 4 we present the new election algorithm. Finally, we present the results on the distributed computation of globally sensitive functions in Section 5.

2. The model

The communication network is represented by a graph (V, E) where V is the set of nodes and E is the set of bidirectional communication links. We denote $|V|$ by n and $|E|$ by m. Each node (see Figure 1) consists of a high speed switching hardware (Switching Subsystem (SS)) which is attached to the communication links and a single processor (Network Control Unit (NCU)).

Delay and cost measures

As mentioned in the previous section, a message suffers a hardware delay at every hop (namely, a transmission delay in traversing the communication link and a switching delay in the node's SS). In some cases, the message will travel to the NCU in which case it will suffer an additional software delay. Our model is similar to the standard model for asynchronous communication networks [eg. GHS83] in that all delays associated with a message are assumed to be finite but unbounded. No errors or lost messages are permitted unless link or node failures occur (as described later on in this section.) FIFO ordering of messages is assumed. (In fact, FIFO ordering is required only in Section 5). For purposes of time complexity analysis, we count hardware and software delays separately. In particular, we assume that the hardware delays are upper bounded by C time units and the software delays are upper bounded by P time units. These upper bounds are only used for the purpose of time complexity analysis and the algorithms must operate correctly even if delays are unbounded. Our model permits the transmission of the same message over multiple outgoing link of the transmitting node at no extra processing cost. This feature applies for the PARIS network [CG88] but might not apply in other architectures. We use this feature in the solution of the first problem.

We have the following definition:

- Time Complexity - The maximum time which may elapse from the algorithm starting point to its end under the assumption that the hardware and software delays are upper bounded as described above.

In terms of network resources cost we have two measures that are both meaningful. The first is the traditional communication complexity, i.e. the number of hops traversed by messages in the operation of the algorithm. This captures the hardware cost in our model. The other cost is the software cost which we term "system-call" complexity. It is defined as follows:

- System-Call Complexity - The sum over all nodes of the number of times that each NCU is involved in the algorithm process.

The limiting case on which we concentrate for most of the paper is where hardware costs and delays are negligible. For this case, the only contribution to the delay is when the NCU is visited and the only meaningful cost measure is system-call complexity. We also assume (as validated in [CJRS89]) that most of the system-call delay is due to the routine processing of messages, i.e. data movement, processor interrupts, task switching, message reassembly, integrity check (CRC), etc.. The algorithmic steps are considered to take negligible time and therefore the system-call delay does not depend on the message content.

The hardware model

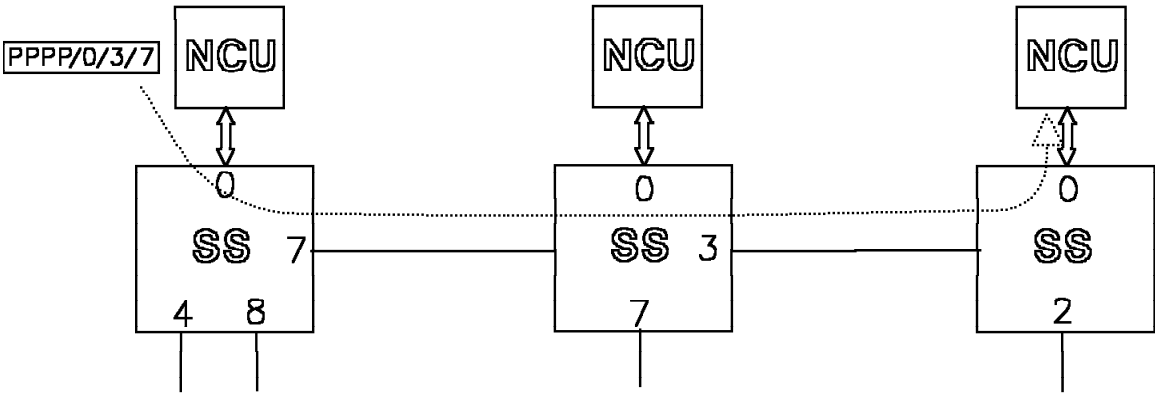

Figure 2. ANR Structure.

The switching functions that can be performed in the SS alone are considered fast and cheap. It is therefore important to define precisely these switching functions. Our definitions are based upon technology availability as embodied by specific hardware structures under development (e.g. [CG88]).

Consider an SS with m bi-directional incident links. (The NCU is a special case of an incident link.) We assume that link i has a finite, non-empty, set of ID′s, L_i, where all ID′s are of length k bits. We restrict k to be $O(\log m)$. A packet (a **message**) p, is a string of bits. We represent packet p by the concatenation of two substrings, $p = xy$, where x is of length k. The SS receives xy as input over one of its links and outputs the string y over every link i, such that $x \in L_i$.

In this paper we use the hardware model in a specific way. Each link is assigned an ID called the normal ID. The normal ID′s are unique within a single SS. (The incident link leading to the NCU is always assigned the same predefined normal ID, say the ID ′0′ in each SS.) A link may have different normal ID′s at each of its end points. In addition each link apart from the link to the NCU is assigned a second unique ID, the copy ID. (Typically, the copy ID and the normal ID can be identical except for the most significant bit.) The link to the NCU is assigned all the copy ID′s of the other links. Thus, every link has exactly two ID′s except the NCU which has m ID′s. Copy bits are used to copy a message that traverse a certain path to all or some of the NCUs along the path. An example how the copy functions is used for setup and take-down of calls appears in [CG88].

Using the hardware structure described above, we can route messages in the following fashion. Assume that a certain node (where no ambiguity exists, we shall say “node” instead of “the NCU in a node”) wishes to send a packet to a certain destination node and that it has knowledge of a suitable path to that destination node. To accomplish this, it prefixes the data with a string that is the concatenation of all the normal link ID′s along the computed path. The message travels

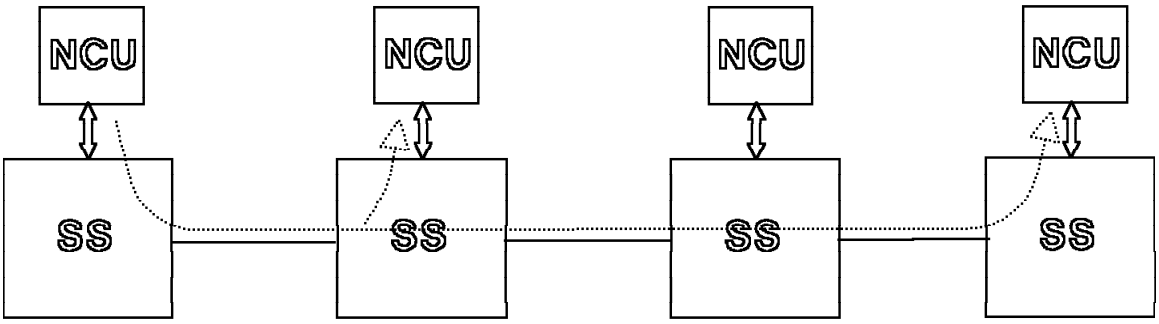

Figure 3. The Selective Copy.

through the SS′s along the predetermined route to the specified NCU. (See Figure 2.) It will experience no software delays along the path (except the endpoints). The routing technique is called Automatic Network Routing (ANR) or source routing [CG88]. We denote the sequence of ID′s that are used to route a packet along the path as the *ANR header*.

Sometimes, it is useful to drop a copy of the message at intermediate nodes along the path. This can be done by using the copy ID (or set the copy bit) rather than the normal ID in the appropriate position in the header string. As can be seen, a message with a copy ID will be sent in parallel to the NCU and to the appropriate output link. This routing mechanism (selective copy) is depicted in Figure 3.

In many scenarios it is useful for a receiver to be able to reply immediately to the sender. Since the sender can be physically non-adjacent from the receiver, this capability would, in general, require the receiver to possess a routing table or topology map. Several techniques exist which permit the receiver to respond without such information ([CG88, CGGG93]). These techniques include having the sender add reverse path information, enhancements to the hardware, etc.. We will assume in this paper that through the use of one of these techniques, a receiver will be able to send a packet back to the sender.

It is possible that different routing mechanisms can be defined for our hardware model. In addition, it is possible to define more powerful models with the capability of performing more functions at hardware speed. (For example, update of a stored variable, table lookup and compare function.). The hardware model chosen for this paper is based upon proposed designs that are likely to be deployed in real networks in the next decade [TW83, CG88, HA87]. In defining this model, we have tried to capture a most common subset of functions shared by the various hardware designs.

Path length restriction

As described above, the model permits unbounded length paths through the hardware. In order to make the model more realistic we restrict the length of the maximal path permitted through the hardware. We denote this upper bound by d_{max}. The rationale for such a restriction is twofold:

1) With source routing (ANR), the length of the message grows linearly with the path length. While hardware costs and delays are assumed to be negligible, there are typically restrictions on message buffer sizes and other hardware components that physically prevent the transmission of very large messages or make the cost and delays non-negligible any more.
2) While we have assumed no errors in transmission, there is actually a small probability that a message will be corrupted during a transmission. Though this number is usually very small and can be ignored (around 10^{-12}), it can be significant when the path length is very large. It is therefore necessary to bound the path length in order to preserve the assumption of error free transmission.

The choice of d_{max} actually defines a class of models. A reasonable value for d_{max} is the typical diameter of the network. The rationale behind this choice is that most communication networks are designed for the support of end-to-end communication. Therefore, parameters such as maximal packet length, buffer sizes and link error probabilities are tuned for end-to-end services. In this paper we will either use the diameter or the total number of nodes as our choice for d_{max}.

Changing topology

In the following, we describe the extensions to the model required for modeling dynamically changing topology. This model is used for the first problem (maintaining network topology). An NCU is initially familiar with the ID′s (both copy and normal) of its adjacent links and the identity of the neighboring nodes. A link is either *active*, in which case it delivers every message sent over it to the other side in finite but unbounded time, or *inactive*, in which case it does not deliver any messages. An inactive node is modeled by a node all of whose links are inactive. If an adjacent link remains in either the active or inactive state for sufficiently long period, the NCU will be aware of that state. This assumption is typically realized through a data link control protocol and is weaker than the assumption normally made for distributed algorithms in changing topology networks [BS84].

3. Topology Maintenance

A broadcast algorithm for topology maintenance is described in Section 3.1. The algorithm is analyzed in Section 3.2. A matching lower bound proof for a certain class of algorithms is given in Section 3.3. In this (and the next) section, we assume that hardware delays are negligible in comparison to software delays.

We briefly define the topology maintenance problem (a more complete definition is provided in [T77]). Each node keeps the properties of each adjacent link including the identity of the neighboring node, the various link ID's, the operational state and other parameters such as the load condition. The collection of these properties for all adjacent links is referred to as the node's local topology. The purpose of the topology maintenance algorithm is to maintain in every node a complete updated description of the topology of the network, i.e. the collection of all the local topologies. As some of the properties dynamically change, it is not possible to maintain an exact consistent global topology. The objective, therefore, is to maintain eventual consistency, i.e. when all topological changes stop, then within a finite time all nodes will have a consistent and correct view of the global topology [T77]. As in the ARPANET [MRR80], this eventual consistency is achieved by having every node broadcast its local topology periodically (with an incremented sequence number each time). We are interested in minimizing the time and system call complexity per broadcast. The broadcast in the ARPANET is based on a "flooding" algorithm that sends a message over every link and consequently takes $O(n)$ time and $O(m)$ system calls. In contrast, each of our broadcasts uses a tree, thereby saving system calls and also improving the time complexity of each broadcast. This is done by dividing the tree into paths, and broadcasting over each path in one unit of time. In the next section we give details of our tree decomposition algorithm. (Other methods of tree decomposition appear in different contexts, e.g. in [HT84, SV88, T83].)

3.1 The Topology Broadcast Algorithm

First, we give a brief description of the overall algorithm. Prior to the t^{th} execution of the broadcast, node i has obtained some information about the topology of the network. By assumption, the node is always aware of its local topology. Information on remote nodes is obtained from the topology maintenance messages received from other nodes. Note that the remote topology information may not reflect the actual state of the network at that instance. Let $G_i(t)$ be the network topology according to i's most recent information just before the t^{th} periodic execution. Essentially, i's algorithm is: (1) compute $T_i(t)$- a spanning tree (rooted at i) of minimum hop paths in $G_i(t)$ from i to all connected nodes in $G_i(t)$ (2) send the local topology ($G_i(t)$) over $T_i(t)$ with an incremented sequence number according to the broadcast algorithm and (3) update the remote topological information according to the most recent (determined by sequence number) topology messages received. The main issue of this section is to devise an appropriate and efficient broadcast algorithm for (2).

Under our model, there are various ways to perform a broadcast. For example, i may send a message directly to each node. The system call and time complexities are both $O(n)$. Another way is for i to generate a single message which will traverse (in Depth First Search manner, see, for example, [E79]) the tree, and be copied once by every node. The system call complexity in this case is still $O(n)$ (actually, exactly n). However, the time complexity is only 1!

Unfortunately there is a problem in employing this Depth First Search method in the presence of link failures as the traversing message is lost on encountering the first failure. Thus, a node j may not receive the broadcast message even if the route from i to j on $T_i(t)$ is active. This may lead to a problem of non-convergence where there is no further topological change but the nodes remain forever with inconsistent topological information. To see this problem, we present the following example.

Example: Consider a graph with six nodes u, v, w, u_1, v_1, w_1 and 6 edges (u,v), (v,w), (w,u), (u, u_1), (v, v_1), (w, w_1). Assume that the last three edges fail, and nodes u, v and w try to broadcast (using the Depth First Search method) topology updates notifying about these failures. Assume further that the path chosen by node u starts with links (u,v) and (v, v_1). Clearly, its message will never reach node w, unless node u receives node v's broadcast describing v's local topology. However, by symmetry, node u never receive v′s broadcast unless v receives w's broadcast, and v never receive w's broadcast unless w receives u's broadcast. In other words a deadlock is reached and the topology map known to the nodes never converges to the actual topology.

The scheme developed in this section overcomes this problem. Its system call complexity is still $O(n)$. The time complexity is bounded by $O(\log n)$. To overcome the problem demonstrated in the above example our broadcast is one-way: We define a one-way algorithm on a tree to be one in which a link is traversed (zero or more times) only in one direction (away from the root)[1].

The following sequential algorithm (performed by i on $T_i(t)$) is used to construct paths for the broadcast message (see Figure 4). First assign the label 0 to each leaf of $T_i(t)$. Let j be a node all of whose children in $T_i(t)$ have been labeled. Consider the child with the largest label l. If j has another child labeled l then we label j with $l + 1$. Otherwise, we label j with l. This procedure is repeated until all nodes are labeled. Also, assign the label of every node $j \in T_i(t) - \{i\}$ to the edge from j to its parent.

Let k be i's label. Let us now construct a path. We start this path from i and extend it to a child of i. Let l be the label of the edge between i and that child. Extend the path away from i as long as possible using only edges labeled l. As a consequence of Lemma 1 (below) this procedure is well defined. In order to construct the next path, remove all the edges of this path from $T_i(t)$.

---

[1]If direct messages whose path length is of $O(n^2)$ are permitted (i.e., there is no path length restriction), then a simple algorithm can be designed which guarantees convergence after $O(\log n)$ rounds and has a time complexity of one time unit. This algorithm is based on traversing the minimum hop tree (BFS tree) a layer at a time. First we traverse the sub-tree that spans all nodes of distance 1 hop from the root (but no node of distance greater than 1) and terminate the traversal at the origin. We then traverse the sub-tree that spans all nodes of distance 2 hops and go back to the origin, and so on. The overall traversal is composed of a concatenation of these subgraphs traversals. Messages are copied only during the first visit at each node.

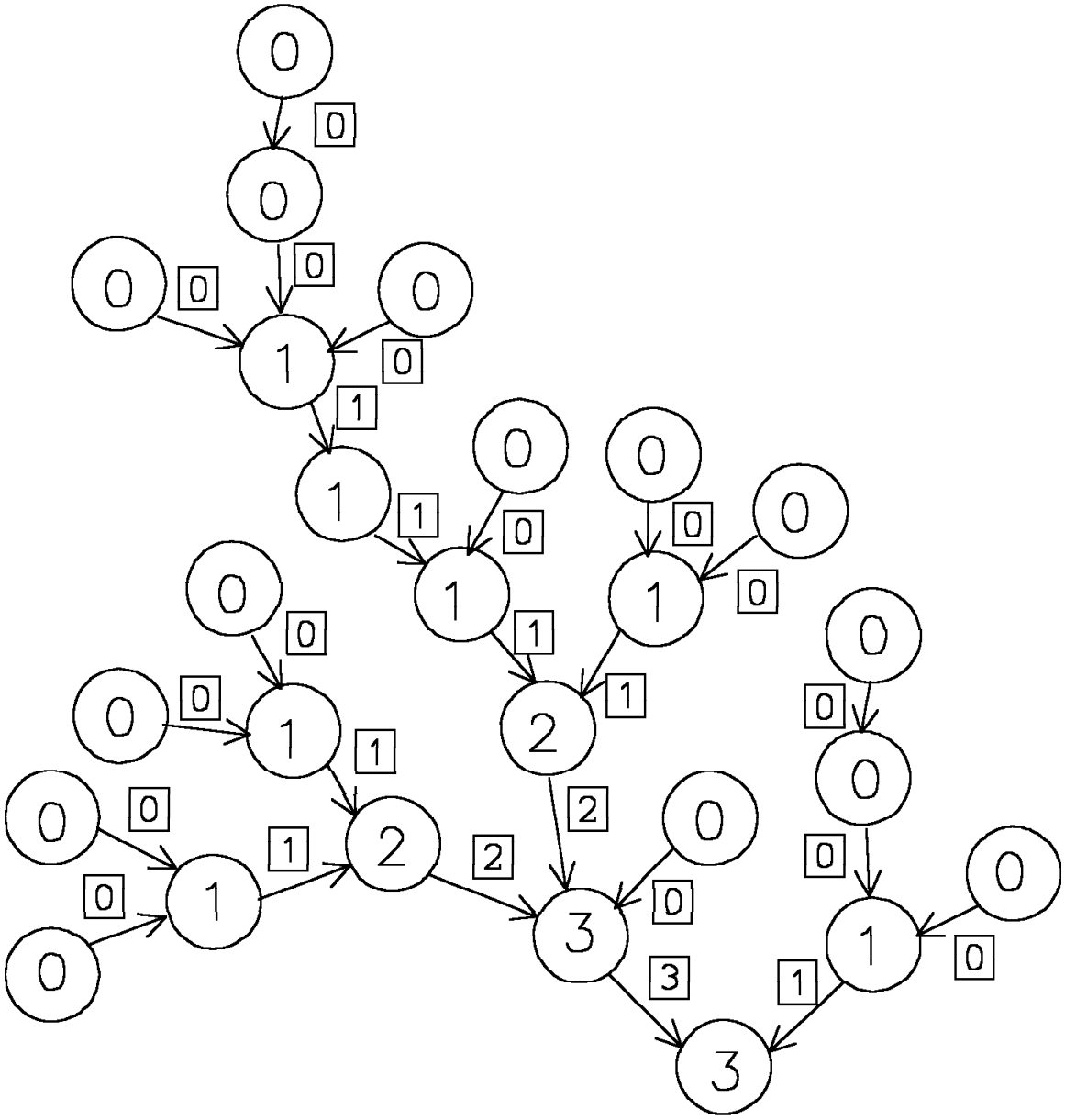

Figure 4. The Branching Path Broadcast Algorithm.

The next path starts with the largest numbered non-isolated node, and is constructed similarly to the above. This operation is repeated until all the nodes are isolated in $T_i(t)$.

Node *i*'s broadcast is performed over the set of paths defined above. First *i* sends its local topology over all the paths that start from it. Note that the number of such paths is at most the degree of *i* and consequently by the broadcast primitive this can be done in one unit of time. Using the copy mechanism, the message is received by every NCU along these paths. In addition to the local topology, the message contains a description of the tree, enabling every starting node *j* of a

new path to know that it is such a node. When j receives the broadcast message it sends it over all the paths that start in it. Again, this takes one time unit. We refer to the algorithm as **branching paths broadcast**. The system call complexity of each broadcast is, clearly, n. The time complexity is the maximum number of distinct paths that a message will have to traverse. In the next section we prove that the algorithm operates correctly and has time complexity bounded by $\log n$.

3.2 Analysis

The following two lemmas are easy to prove.

Lemma 1: In the labelling procedure described above, a node of label l can have at most one child of label l.

Lemma 2: Let P be a path on $T_i(t)$ that starts in i, and contains only active nodes and edges. Then, when i uses the branching paths broadcast protocol, every node on P receives the broadcasted message.

The next result demonstrates that the topology maintenance algorithm operates correctly according to the definition of correctness provided in [T77].

Theorem 1: Assume that the topological changes cease at some time. Then, eventually, every node in the network knows the correct topology of its connected component.

Proof: Consider the network after the last topological change. We prove the theorem by an induction on the distance in this final network (measured in hops) of a node from every other node. The base case follows from the assumption on the data link protocol: every node eventually knows the network topology within one hop of itself. Assume that every node eventually knows the topology within k hops of itself. Let u be a node at distance $k + 1$ from i (if no such node exists we are done). Let v be a neighbor of u whose distance from i is k. Clearly v knows about u after one unit of time. By the assumption a route of active edges and links from v to i appears in $T_v(t)$ at some time t. By Lemma 2 v's broadcast at that time will eventually reach i. Thus, i eventually knows the topology within $k + 1$ hops of itself.

•

Comment: Let d be the network diameter. Using a similar argument to the above proof it can be shown that a node's topology knowledge covers at least a distance k just before its k-th broadcast. Thus $O(d)$ broadcasts per node may be necessary until all nodes know the whole topology. This can be improved to $\log d$ by having each node broadcast all the topological information it knows.

Theorem 2: The time complexity of each broadcast is bounded by $\log n$.

Proof: Consider the labeling algorithm of Section 3.1. Let j be a node whose label is l. It is easily shown (by induction on l) that in the subtree (of $T_i(t)$) rooted in j there are at least 2^l nodes. Thus, the highest label in $T_i(t)$ is at most $\log n$.

Let x be the highest label. Clearly a path labeled x receives the broadcast message at time unit $1 = 1 + x - x$. Similarly, a path labeled $x - 1$ receives the broadcast at time unit 1 in case it directly starts at the root and at time unit $2 = 1 + x - (x - 1)$ otherwise. Generally, consider a path labeled by y. By an induction on the value of y it is easy to prove that the broadcast message is transmitted over this path after no more than $1 + x - y$ time units. The theorem follows.

•

3.4 Lower Bound

In this section we prove that $\Omega(\log n)$ time is necessary for any one-way broadcast.

Theorem 3: Any one-way broadcast algorithm uses $\Omega(\log n)$ messages to cover a (rooted) complete binary tree.

Proof: Consider an execution of any one way broadcast algorithm on a complete binary tree of depth D. To prove the lower bound it is enough to consider only an execution in which the delivery of each message takes exactly one time unit. The following claim shows that at time t after the broadcast starts, there are nodes at some level $l(t) = O(t)$ that have not yet received the broadcasted message. Given the claim, the theorem follows by considering $t = \Omega(\log n)$.

Claim: For every t $(1 \leq t < \frac{D-5}{5})$ there exists a set V_t such that

1. V_t is a subset of vertices at depth $5t$ of the tree,
2. the nodes of V_{t+1} are descendents (in the tree) of the nodes of V_t,
3. $|V_t| = 2^t$, and
4. At time t in the execution (i.e. just after the tth step in the execution) no node of V_t has yet received the message.

Proof of the Claim: Identify the sets V_t inductively. The source can send at most two messages in each time unit. After the first time unit at most two nodes at depth $5t = 5$ have received the message. However, there are 2^5 nodes at depth 5. Thus we can select two (other) nodes at depth 5 that have not yet received the message.

Now suppose we have V_t and consider $t + 1$ (if $t + 1$ is still no larger than $\frac{D - 5}{5}$). Let S denote the set of descendants of V_t at depth $5(t + 1)$ of the tree. By (4) of the induction hypothesis, no node of S has received the message before the $t + 1$th step. Messages reaching nodes of S at step $t + 1$ arrive along a path from some node that is

(a) a predecessor of a node in V_t (by the assumption that the algorithm is one way);

(b) is not in V_t (by part (4) of the induction hypothesis).

Each such predecessor produces at most two paths at time $t + 1$, and each such path passes in at most one node of S (by the one-way assumption). Hence, denoting the number of predecessors of nodes in V_t by P_t, at most $2P_t$ nodes in S receive the message in time $t + 1$, and at least $B \geq |S| - 2P_t$ do not. Now $|S| = 2^{5t + 5}$ while P_t obeys the recursive inequality $P_t \leq 5|V_t| + P_{t-1}$, implying $P_t \leq 6(2^{t+1})$, so $B \geq 2^{5t+5} - 6(2^{t+2}) \geq 2^{t+1}$. Thus it is possible to choose 2^{t+1} ″uninformed″ nodes of S into V_{t+1} and the theorem follows.

•

4. Leader Election

In the leader election problem all the nodes start in the same state (*not.leader*). When the algorithm terminates exactly one node is in a special state (*leader*), and the others are in another new state (*leader.elected*).

The leader election problem has been extensively investigated in the literature. Before we describe the new algorithm in detail, we first motivate the need for a new algorithm and describe other key properties of our algorithm. A straightforward application of the traditional techniques to the new model would result in system call complexity of $\Omega(n \log n)$, as explained later.

To motivate the need for a new technique which exploits the new model and reduces the system call complexity, we introduce an example (Figure 5). It shows a typical situation during the execution of a ″generic″ leader election algorithm. There is a set of nodes which are candidates for leadership. These candidates communicate with each other and the set of candidates is gradually reduced until, finally, there is a single candidate which becomes the leader. The reduction is accomplished through comparison of the candidates′ ″priority″; the lower priority node ceases to be a candidate and becomes a ″supporter″ of the higher priority node. (The previous supporters of the lower priority node will also become supporters of the higher priority node.) In the example, nodes A and B are candidates for leadership. The eight lower nodes are supporters of B, while the upper nodes are supporters of A. By communicating with node E, node A discovers that E supports another candidate (i.e. B). If the priority of A is smaller than that of B then A (and its supporters) become supporters of B. This process is repeated until exactly one candidate remains. Our algo-

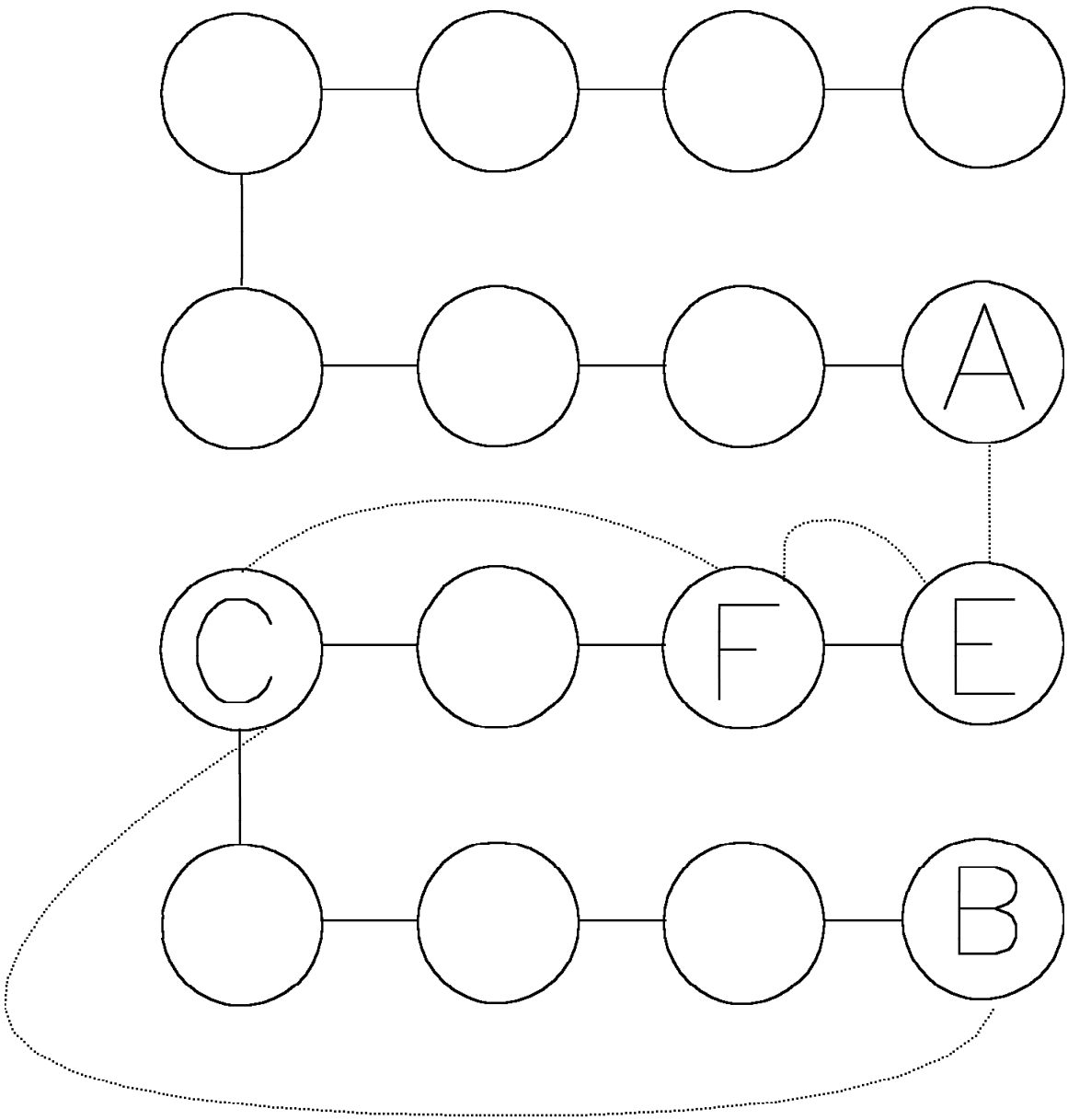

Figure 5. Leader Election Example

rithm follows this paradigm. The novel part is the method we use to reduce the system call complexity of the operation of comparing the priorities of such A and B.

Recall that in our model, if the route to node B is ″known″ to node E, then the priorities of A and B can be compared using only two system calls. That is, A′s message is sent to E, and then forwarded directly to B. However, we now face the question: ″do we notify all of A′s supporters that they are now supporters of B (and give them the route to B)?″ If we do, then the number of system calls paid for the elimination of A is linear in the number of A′s supporters, leading to a total

system call complexity of $O(n \log n)$. On the other hand (if we do not notify A's supporters), consider a candidate who contacts H, a supporter of A. Note that H is now, actually, a supporter of B. However, (unlike the previous case with E) the route to B is not available in H. Thus, B cannot be reached from its supporter H in one system call. The way to reach B will be to go to H, then to A, and only then to B. If supporters are not notified of a candidate's elimination, and if the process of candidate elimination is not controlled, then the number of system calls needed to eliminate a candidate will be linear with the number of its supporters. This again leads to a system call complexity of $O(n \log n)$.

The algorithm presented below does not notify A's supporters that they now support B. However, candidate elimination is done in a way that ensures that a message can be sent from any supporter of a candidate B to B using only a logarithmic (in the number of supporters) number of system calls. Note that this technique is very similar to the one used for the broadcast in Section 3.

4.1 The Leader Election Algorithm

We present the algorithm in terms of **tokens** (see [KKM85]). We view the token as a process that can migrate from a node over an edge to the other end-point. By repeating the operation of migration, a token traverses a graph or a subgraph. In order to avoid problems of mutual exclusion, no two tokens are processed concurrently in a node. Instead, once a token has control over a node, it retains control until it either halts or migrates to another node or executes a "wait" instruction. At that time the control may pass to another token (which is either released from a "wait" instruction, or has migrated from another node). The use of tokens simplifies the presentation and proofs. Clearly, the token algorithm can be simulated by a conventional algorithm [KKM85].

During the operation of the algorithm the nodes are partitioned into **domains**. Each domain is a collection of nodes. At the beginning of the algorithm each node belongs to its own domain. At the end of the algorithm all the network is included in a single domain.

Each node creates a candidate token that represents its domain. The token carries the identity of its creating node, and the number of nodes in its domain. The creating node of a candidate is called its **origin**. In origin node i, two sets are recorded: a set of all nodes in the domain (IN_i) and a set of all neighbors of these domain nodes which are outside the domain (OUT_i). For each node in both lists, an ANR field is attached which permits this node to be reached via a direct message. This data structure is kept as a tree (the $INOUT_i$ tree) which is a subgraph of the network's graph. Thus, all the ANR field lengths as well as the data sets sizes are linear in n.

A domain is organized as a (virtual) tree. The origin is the root. Every other node i in the domain has a pointer (ANR field) to the parent of i, denoted by F_i, in the virtual tree. Note, that F_i is not necessarily a neighbor of i. Thus, unlike the $INOUT_i$ tree, the domain tree is not necessarily a subgraph of the network's graph. However, it is possible to reach F_i from i via a direct message.

During the execution of the election algorithm candidates may enlarge their domains by capturing other domains. Let the **size** of i, (S_i), be the number of nodes in i's domain, the **level** of candidate i (L_i) be the pair (S_i, i), and the **phase** of candidate i (PH_i) be $\lfloor \log_2 S \rfloor$. A candidate which has noticed the existence of another candidate, with a higher level, becomes **inactive** and does not attempt to enlarge its domain any more.

The algorithm at each node i starts when it receives for the first time either a START message from outside, or a message of the algorithm. At that time it creates a candidate token, sets $IN_i = \{i\}$, and $OUT_i = \{j/(i,j) \in E\}$. candidate i has $S_i = 1$ and $PH_i = 0$.

An active candidate, i, repeatedly performs **tour**s for the purpose of capturing other domains. The tour is started at the candidate's origin by selecting an arbitrary node o from the set OUT_i. Next, the candidate travels to this node and attempts to reach an origin by going from that node to its parent via the virtual tree. However, candidate i will never travel more than $PH_i + 1$ "hops" in its search for an origin. (Note that a "hop" is comprised of a single direct message and may traverse several nodes.) If an origin is not found after such a limited length tour, candidate i directly returns to its origin and becomes inactive. In order to be able to directly return to its origin, the candidate always carries ANR(o,i). Let q be the node in which candidate i has decided to return to its origin. ANR(q,o) is at that time computed in q, using $INOUT_q$. This can be done since (as we shall see) $o \in IN_q$. This route is used, rather than the reverse of the concatenation of candidate i's ANRs from o to q, since the length of the latter may be more than n.

Let us now describe what candidate i does when it arrives at a new node v, in its tour. We have the following cases:

(1) v is not an origin -

If the length of the tour is already more than PH_i, then candidate i returns to its origin and becomes inactive. In order to return, the candidate first finds in node v a linear length ANR to o (since $o \in IN_v$). It uses the concatenation of this ANR and $ANR(o,i)$. If candidate i has not became inactive in v then it proceeds to node F_v.

(2) v is an origin-

(2.1) If $L_v > L_i$ (lexicographically)- candidate i returns to its origin and then becomes inactive.

(2.2) If $L_v < L_i$ and the local candidate is inactive. - candidate i captures domain v by assigning $F_v = i$ (and leaving a linear length ANR(v,i) in v). It then returns to its origin carrying with it both of v's sets. There, a merging of the sets is performed by assigning

$IN_i = IN_i \cup IN_v$, $OUT_i = OUT_i \cup OUT_v - IN_i$ (Actually the $INOUT_i$ and $INOUT_v$ trees are combined in node i by connecting node o of IN_v to its neighbor in IN_i. This keeps the ANRs linear in length.) Also $S_i := S_i + S_v$. The candidate is now ready for a new tour.

(2.3) If $L_v < L_i$ and the local candidate is on a tour and no other candidate is waiting in v - candidate i waits for the return of candidate v and for the completion of all computation that are related to this comeback. At that point a comparison between the candidates' levels is done and actions are taken in a manner similar to (2.1) and (2.2).

(2.4) Same as (2.3) but there is another candidate, j, waiting in v - The lower level candidate returns inactive to its origin.

If after a tour the OUT set becomes empty, then the candidate declares itself the leader. Otherwise, the candidate proceeds on another tour.

4.2 Correctness Proof

In this section we prove that a single candidate is elected in every execution of the algorithm. In order to prove the general result (Theorem 4) we use several lemmas. Lemma 3 is used to prove that rule (1) of the algorithm does not cause the algorithm to deadlock. Lemma 4 will be used to show that no deadlock results from rule (2). Lemma 5 shows that a candidate that remains alive (Lemmas 2 and 4) will, eventually, capture every other candidate. The proof of Lemma 3 also shows the similarity between the routing techniques used here, and in Section 3 (in the paths construction). This lemma is also used later in the analysis of time and system call complexities.

Lemma 3: The depth at any time t of a virtual tree of a candidate i that is alive at time t is never more than i's phase at time t.

Proof: Note (rules (2.2) and (2.3)) that the size of a capturing candidate is no less than that of the captured one. Thus, when candidate i approaches the origin of candidate j from a leaf, the number of nodes in the subtree below it increases at least by a factor of two in each step. The lemma follows.

•

Lemma 4: During any execution of the algorithm there is always at least one alive candidate.

Proof: By Lemma 3, if candidate i becomes inactive because of rule (1) then it is trying to arrive at an origin of a candidate in a higher level. If it becomes inactive because of rule (2) then it is

during a visit to the origin of a candidate in a higher level. Thus, at any given time, the candidate with the highest level cannot become inactive.

•

Lemma 5: A candidate that starts a tour eventually returns to its origin.

Proof: Note that a candidate waits only for a candidate with a smaller level (in the origin of the latter). Thus, at least the smallest in a chain of waiting candidates must return.

•

Theorem 4: If a non-empty set of nodes starts the leader election algorithm, then the algorithm eventually terminates (and exactly one node declares itself a leader).

Proof: The theorem follows from Lemmas 3 through 5, and from the observation that OUT_i can become empty only when all nodes have been visited by candidate i.

•

4.3 Complexity

Theorem 5 gives a bound on the system-call complexity, and thus also on the time complexity. In the theorem we use the following lemma

Lemma 6: There are at most $\frac{n}{2^p}$ domains at phase p.

Proof: By the definition of a phase there are at least 2^p nodes in phase p. The lemma now follows from rule (2.2) which implies that a node can belong to at most one domain in each phase.

•

Theorem 5: The number of system-calls in the algorithm is never more than $6n$.

Proof: All the system calls (direct messages) of the election algorithm are the result of candidates going on tours and returning from them. First, consider tours in which the touring candidate succeeds in capturing a domain. Let V_p be the number of candidates that enter phase p. The number of domains captured in phase p is then

$$\sum_p (V_p - V_{p+1})$$

By Lemma 3, the capturing of a domain in phase p requires at most $p + 2$ messages (including the one to a node in the other domain, and the direct message that is needed for the return of the candidate to its origin). Hence, the number of messages used in capturing domains is

$$\sum_{p}(V_p - V_{p+1})(p + 2) = \sum_{p>0} V_p + 2(V_0 - 1) \leq 3n$$

Consider now the messages used by candidates for tours in which they have not succeeded in capturing a domain. In such a tour, the touring candidate becomes inactive. This may happen only once per candidate, and by rule (1) the number of messages used when this happens to a candidate in phase p is not larger than $p + 2$. Thus, the same bound we have computed for successful tours also holds for the unsuccessful tours. The theorem follows.

•

5. Distributed Computation of Globally Sensitive Functions

In this section we clarify some of the basic differences between system call complexity and communication complexity for a broad class of distributed computation problems. Previously, we assumed that the nodes do not know in advance a direct path to remote nodes. By learning this information via the operation of the algorithm, the features of the new model could be exploited. This might lead to the (wrong) conclusion that once each node knows how to directly reach any other node, the difference between the old model and the new class of models disappears. In other words, one might incorrectly concludes that once a complete routing information is available, the new model is no different from a completely connected traditional network where every node can reach any other in a single hop.

We disprove this wrong conclusion by considering a network whose underlying topology is a complete graph where each node knows its neighbors′ identities. We demonstrate that the optimal structure of distributed computation is still sensitive to the trade-off between communication and system-call delays. In order to capture this trade-off, we do not restrict our attention to the case where communication delays are negligible. Rather, we assume a more general case in which the worst case communication delay of each message is C and the worst case system call delay is P, where C and P are arbitrary numbers. As mentioned in the introduction, problems under this model have been further investigated in [BK92,CKPS93].

5.1 Problem formulation

We assume an asynchronous completely connected network. Without loss of generality we assume that there are n nodes, numbered 1, 2,..., n. We assume that each node i maintains a local

input value I_i which is drawn from a finite alphabet. A distributed algorithm is triggered at time 0 at all nodes. The algorithm terminates eventually at node 1. Upon termination, node 1 should have computed the correct value of some function $f(I_1, I_2,..., I_n)$. Function f is defined for all vectors of length equal or smaller than n.

The function f is defined to be:

1. Associative - $f(X, Y, Z) = f(X,f(Y,Z)) = f(f(X,Y),Z)$
2. Commutative - $f(X,Y) = f(Y,X)$

In addition f is **globally sensitive**. To define this term, we introduce some additional notation. Consider a vector $\overline{K} = (K_1, K_2, ..., K_j, ..., K_n)$. Let $\overline{K_j}(m)$ denote the vector $\overline{K}$ with the value of K_j set to be m (all other values remaining as in $\overline{K}$). For function f, we now define a **globally sensitive input vector**, $\overline{I} = (I_1, I_2,..., I_n)$, to be one where, for every j, $(1 \leq j \leq n)$, there exists a value m such that $f(\overline{I_j}(m)) \neq f(\overline{I})$. (Note that m is drawn from the same original finite alphabet as the components of input vector $\overline{I}$). A **globally sensitive** function is one for which there exists at least one globally sensitive input vector. Somewhat stronger definitions of global functions are also provided in [KMZ84,ALSY90]. Clearly, any such function can be computed in one time unit in the case of a complete graph under the traditional model. This is not necessarily the case under the new model.

5.2 The Construction of Optimal Algorithms

Our goal is to construct an optimal (time and system call) algorithm to compute globally sensitive functions. The optimal algorithm may be different for different values of C and P. Therefore, we assume that P and C are given. We first show that there exists a spanning tree and a simple optimal algorithm that sends messages only over a that tree and at most one message over each tree edge. This is established in Theorem 6 below. Later, we show how to compute the tree, given P and C. We begin by defining the simple algorithm given a spanning tree.

Definition: A *tree based algorithm* is an algorithm where the function is computed over a predefined spanning rooted tree (the same tree for all possible input vectors). The computation is done in the following fashion: At initialization, all tree leaves send their input values to their parents. Each node (which is not a leaf) waits until it has received a message from all its children and then computes the partial function of its subtree and forwards the result to its parent.

Theorem 6: There exists a single tree-based algorithm which is worst-case optimal for all functions f.

The proof appears in the appendix. Intuitively, it is based on the observation that in every execution of every optimal algorithm one can identify a tree, consisting of edges over which certain

″important″ messages were sent. These messages, termed **causal**, are messages which have a causal relationship (viz. Lamport′s ″happened before″ relation [L86]) to the output of the computed function at node 1. An algorithm may send many causal messages, but if we consider for every node the edge over which it sent the last causal message, this defines a tree. This tree can then be used by a tree based algorithm. See the appendix for more details.

We now address the main question of identifying the best tree-based algorithm, i.e., identifying the particular spanning tree over which the best algorithm is defined. Instead of formulating the problem as "what is the tree over which *n* nodes can perform the algorithm in minimum time" we first address a somewhat different but related question of "given a worst case termination time *t* what is the tree with the maximum number of nodes over which the tree-based algorithm terminates before *t*". We later show how to use the answer for the latter question for solving the former question.

We define an optimal (t,P,C) tree, as the rooted tree with the maximum number of nodes over which a tree-based algorithm can compute any function *f* and terminates no later than t. Fixing P and C, we denote this optimal tree by $OT(t)$ and the number of nodes in $OT(t)$ by $S(t)$. It is clear that the size of the optimal tree is a non-decreasing function of t. C and P were defined as worst case times. It is easy to prove that increasing any message delay in the tree-based algorithm never decreases the overall time to complete an execution. Therefore, we may assume that in the worst case situation messages encounter exactly the maximum allowable delays (P and C).

Let us consider an optimal tree OT(t). The root of this tree receives messages from its children. It is able to process all of them within time t. Let us observe the last message it processes. This message must be received no later than (t-P) otherwise it cannot terminate at t. This in turn implies that the message must have been sent before (t-P-C). In addition, assuming that messages are processed in FIFO order, the root must have received and processed all other messages by that time (t-P). It is also clear that:

$$S(t) = \begin{cases} 0 & t < P \\ 1 & t < 2P + C \end{cases} \tag{1}$$

Let X and Y be rooted trees. We define the operation $\overleftarrow{\cup}$ as an operation that gets as input two rooted trees and outputs a new tree in the following way. The rooted tree Z which is the result of $Z = X \overleftarrow{\cup} Y$ is constructed by augmenting the root of Y (we add a directed edge) as a child of the root X.

Since $S(t)$ is a non decreasing function of t, this leads us to the following recursive structure of the tree ($t \geq 2P + C$).

$$OT(t) = OT(t - P) \overleftarrow{\cup} OT(t - C - P) \tag{2}$$

This also translates to the numerical equation:

$$S(t) = S(t - P) + S(t - C - P) \tag{3}$$

Both the tree structure and size can be solved using the recursions defined by (2) and (3) respectively and by the initial conditions (1). One has to compute $OT(t - i(C + P) - jP)$ for all i,j such that $t - i(C + P) - jP \geq 2P + C$. The simplest way to do it is by computing the values of t, to be considered, ordering them from low to high and computing the structures and the values according to this order. Basically we can restrict the computation to discrete values of time that are in the form $iP + jC$. Other times t can be truncated to the highest value that still satisfied $t \geq iP + jC$.

Examples

1) Model of sections 3 and 4 - C=0, P=1. We can assume only integer values of time. The equations take the following simple structure:

$$S(t) = \begin{cases} 0 & t < 1 \\ 1 & 1 \leq t < 2 \end{cases} \tag{4}$$

$$OT(k) = OT(k - 1) \overleftarrow{\cup} OT(k - 1) \tag{5}$$

This structure is a binomial tree [SCH81]. Solving for the tree size results in

$$S(k) = 2S(k - 1) = 2^{k - 1} \tag{6}$$

2) Traditional model - C=1, P=0. It is easy to see that the recursion blows up. This is because by using a star configuration we can add any number of nodes to the structure and thereby get any tree size for t=1.

3) C=1, P=1

$$S(k) = \begin{cases} 0 & k < 1 \\ 1 & 1 \leq k < 3 \end{cases} \tag{7}$$

$$OT(k) = OT(k - 1) \overleftarrow{\cup} OT(k - 2) \tag{8}$$

This structure grows faster than a binomial tree. Solving for the tree size results in

$$S(k) = S(k - 1) + S(k - 2) \tag{9}$$

These are the Fibonacci numbers and therefore,

$$S(k) = \frac{\left((1 + \sqrt{5})^k - (1 - \sqrt{5})^k\right)}{\sqrt{5}\,(2)^k} \tag{11}$$

Computing the optimal time for a given size

Since the recursion is applied in the order of increasing value of time, one should just compute it for the discrete values of time until $S(t) \geq n$. As was mentioned before we should use only values of time that are in the form of $iP + jC$ where i and j are integers. Since in the tree-based algorithm there are only $n - 1$ messages sent it is clear that: $i \leq n$, $j \leq n$. Therefore, there are at most n^2 possible points at which we need to compute the recursion.

6. Conclusion

We have presented a family of new models for distributed algorithms which reflect the costs of communication, switching and processing in high speed networks more accurately than previously studied models. We have shown that three distributed algorithms can be made to operate more efficiently under this model. However, issues remain open. For example, can other distributed algorithms be similarly improved? What is the relationship between the power of the switching subsystem and the efficiency of the distributed algorithm? Are the models that we have proposed the most suitable for high speed networks or can they be improved? These and other issues form a rich area for future study.

Acknowledgements

The authors would like to thank Jeffrey Jaffe for the BFS algorithm of section 3. We also thank the referees of this paper for their constructive comments that helped us to improve the readability of the paper and to simplify some of the proofs.

References

[ACGK90] B. Awerbuch, I. Cidon, I.S. Gopal, M. Kaplan, S. Kutten, ″Distributed Control for PARIS,″ Proceedings of the Ninth ACM Symposium on Principles of Distributed Computing, Quebec City, pp. 145-160, August 1990.

[AKRP80] P. Amer, R. Kumar, R. Rao, J. Phillips, L. Cassel, ″Local Area Broadcast Network Measurement: Traffic Charecterization,″ IEEE Computer Society International Conference, Piscataway, 1987.

[ALSY90] Y. Afek, G. Landau, B. Schieber and M. Yung, ″The Power of Multimedia: Combining Point-Point and Multiaccess Networks,″ Information and Computation, Vol. 84, No. 1, January 1990.

[B80] J.E. Burns, ″A Formal Model for Message Passing Systems″, TR-91, Computer Science Department, Indiana University, Bloomington, Indiana, 1980.

[B92] J.Y. Le Boudec, ″The Asynchronous Transfer Mode: a tutorial″, *Computer Networks and ISDN Systems*, Vol. 24, pp. 279-309, 1992.

[BK92] A Bar-Noy and S. Kipnis, ″Designing broadcasting algorithms in the postal model for message-passing systems″, to appear in Mathematical Systems Theory. Also in 4th Annual Symposium on Parallel Algorithms and Architectures, ACM, June 1992.

[BS84] A.E. Baratz and A. Segall, ″Reliable Link Initialization Procedures,″ *IEEE Transaction on Communications*, Vol. COM-36, pp. 144-152, 1988.

[BZ90] S. Bitan and S. Zaks, ″Optimal Linear Broadcast″, *Journal of Algorithms*, Vol. 14, No. 2, March 1993, pp. 288-315.

[CG88] I. Cidon and I. Gopal, ″PARIS: An approach to private integrated networks″, *Inernational Journal on Analog and Digital Cabled Systems*, Vol. 1, pp. 77-85, March 1988.

[CGGG93] I. Cidon, I. Gopal, P. M. Gopal, R. Guerin, J. Janniello and M. Kaplan, ″The plaNET/ORBIT High Speed Network″, *Journal of High Speed Networks*, Vol. 2, No.3, 1993, pp. 1-38

[CGK88] I. Cidon, I. Gopal and S. Kutten, ″New Models and Algorithms for Future Networks,″ 7th Annual ACM Symposium on Principles of Distributed Computing″, Toronto, Ontario, Canada, August 1988, pp. 75-89.

[CGK90] I. Cidon, I. Gopal and S. Kutten, ″Optimal Computation of Global Sensitive Functions in Fast Networks″, *Distributed Algorithms*, editors: J. Van Leeuwen and N. Santoro, proceedings of the 4th International Workshop on Distributed Algorithms, Bari, Itali, September 1990, Springer-Verlag, pp. 185-191. Also in IBM Research Report, No. RC 16352, July 1990.

[CJRS89] D. Clark, V. Jacobson, J. Romsky and H. Salwen, ″An Analysis of TCP processing overhead″, *IEEE Communications Mag.*, Vol. 27, No. 6, June 1989, pp. 23-29.

]CKPS93[D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, ″LogP: towards a realistic model of parallel computation″, 4th SIGPLAN Symposium on Principles and Practices of Parallel Programming, ACM, May 1993.

[CS89] R. Cohen and A. Segall, ″A Distributed Query Protocol for High Speed Networks″, Proceeding of the Ninth International Conference on Computer Communications, Tel Aviv, 1988.

[DHSS84] D. Dolev, J. Helpern, B. Simons and R. Strong, ″A New Look at Fault Tolerant Network Routing,″ *Proceedings of the 17th ACM Symp. on Theory of Comput.* 1984, pp.536-541.

[E79] S. Even, ″Graph Algorithms″, *Computer Science Press* 1979.

[G87] Gray, J., Invited talk, Sixth ACM Symposium on Principles of Distributed Computing, Vancouver, British Columbia, Canada, August, 1987.

[GGK90] A.S. Gopal, I.S. Gopal and S. Kutten, ″Broadcast in Fast Networks″, IEEE INFOCOM 1990, San Francisco, pp. 338-347, June 1990.

[GHS83] R.G. Gallager, P.M. Humblet and P.M. Spira, ″A Distributed Algorithm for Minimum- Weight Spanning Trees″, ACM Transactions on Programming Languages and Systems, January 1983, Vol. 5, No. 1, pp. 67-77.

[HA87] J.Y. Hui and E. Arthurs, ″A Broadband Packet Switch for Integrated Transport,″ *IEEE Journal on Selected Area in Commun.*, Vol. SAC-5, No. 8, pp. 1264-1273, Oct. 1987.

[HT84] D. Harel and R.E. Tarjan, ″Fast algorithms for finding nearest common ancestors″, SIAM J. on Computing, 13 (1984), pp. 314-325.

[KKM85] E. Korach, S. Kutten and S. Moran, ″A Modular Technique for the Design of Efficient Distributed Leader Finding Algorithms″, *ACM Transactions on Programming Languages and Systems*, Vol. 12, No. 1, pp. 84-101, 1990.

[KMS87] P. Kaiser, J. Midwinter and S. Shimada, ″Status and Future Trends in Terrestrial Opical Fiber Systems in North America, Europe, and Japan″, IEEE Communications Magazine, Vol.25, No.10, October 1987.

[KMZ84] E. Korach, S. Moran and S. Zaks, ″Tight Lower and Upper Bounds for some Distributed Algorithms for a Complete Network of Processors″, Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing, Vancouver, B.C., Canada, August 1984, pp. 199-207.

[KO86] E. Korach and S. Onn, ″A New Model for Distributed Networks to Improve Complexity with Linear Applications to Multi-Commodity Flow and Routing in Series Parallel Graphs,″ Technical report #439, Computer Science Department, Technion, Haifa, Israel, November 1986.

[L86] L. Lamport, ″On Interprocess Communication, Pat I, II″.″ *Distributed Computing*, Vol 1, pp. 77-101, 1986.

[MASK88] B. Maglaris, D. Anastassiou, P. Sen, G. Karlsson, and J. Robbins, ″Performance Models of Statistical Multiplexing in Packet Video Communications,″ *IEEE Transactions on Communications*, VOL. 36, NO. 7, July 1988, pp. 834-844.

[MRR80] J. M. McQuillan, I. Richer and E. C. Rosen, ″The New Routing Algorithm for the ARPANET,″ *IEEE Transactions on Communications*, Vol. COM-28, May 1980, pp. 711-719.

[PKR84] J. Pachl, E. Korach and D. Rotem, Lower Bounds for Distributed Maximum- Finding Algorithms JACM, vol. 31, No. 4, October 1984, pp. 905-918.

[PS87] D. Peleg, and B. Simons, Fault Tolerant Routings in General Network″, Information and Computation, Vol. 74, No. 1, July 1987, pp. 33-49.

[SCH81] P.J. Slater, E.J. Cockayne, and S.T. Hedetniemi, ″Information Dissemination in Trees,″ SIAM J. of Computing (1981), pp.692-701.

[SV88] B. Schieber and U. Vishkin, ″On finding lowest common ancestors: simplification and parallelization″, *SIAM Journal on Computing*, Vol. 17, No. 6, December 1988, pp. 1253-1262.

[T77]. W.D. Tajibnapis, ″A Correctness Proof of a Topology Maintenance Protocol for Distributed Computer Networks,″ CACM, 20, 7, July 1977.

[T83] R.E. Tarjan, ″Data structures and network algorithms″, CBMS-NSF Regional Conference Series in Applied Mathematics (SIAM 1983).

[TW83] J.S. Turner and L.F. Wyatt, ″A packet network architecture for integrated services,″ *Proc. of Globecom*′83, pp. 45-50, Nov. 1983.

APPENDIX

Proof of Theorem 6.

Given any optimal algorithm let us show how to construct an optimal tree based algorithm that has the same worst case time complexity (and a minimum system call cost). Let us distinguish between messages that can affect the final result and messages that cannot. A **causal message** is defined in the following recursive way. It is either received by node 1 before the algorithm terminates or it is received by some other node before that node sends a causal message. Any other message is defined to be a **non-causal message**. It is clear from the definition that a causal message sent over a link cannot be preceded by a non-causal message as we assume FIFO reception. Lemma A.1 shows that we can essentially ignore or suppress any non-causal message.

Lemma A.1: Consider an arbitrary execution of the algorithm. We can delay any non-causal message at a node or a link for any arbitrary amount of time without changing the algorithm′s results or its termination time.

Proof: The proof follows immediately from the fact that non-causal messages cannot be received before a causal one at any node. Since causal messages cannot be sent after the reception of a non-causal message and since node 1 makes its final decision only due to reception of causal messages the lemma holds.

•

Lemma A.2: Assume that a function *f* is computed. Then, there exists at least one execution of the distributed algorithm in which **each** node other than node 1 sends at least one causal message.

Proof: The proof follows directly from lemma A.1 and the definitions of global sensitive functions and global sensitive input vectors.

•

Following the results of lemma A.2, for every correct algorithm (in particular optimal ones) there are executions in which each node must send at least one causal message. From now on we will focus on these particular executions.

Since we assume that message delays do not depend on their contents, a particular tree based algorithm will have the same worst case time delay for all functions *f* and all possible input vectors. We are looking for optimal algorithms in the worst case (over all executions and possible input vectors). Therefore, in order to prove the optimality of a tree based algorithm, it is sufficient to

show that for an arbitrary algorithm that computes an arbitrary function *f* there exists an input vector and a tree based protocol which is at least as efficient as the chosen algorithm (only for this particular input vector).

Lemma A.3: For any function *f* and any algorithm that computes *f* there exists a tree-based algorithm with worst case time and message cost better or equal to that algorithm.

Proof: Consider an execution of the algorithm for a globally sensitive input vector. Consider the last causal message that is sent by each node. It is easy to show that since this is the last message, the causal path of these messages defines a spanning tree rooted at node 1. This implies that this last causal messages process is equivalent to the operation of a tree based algorithm.

Now let us execute the tree-based algorithm over the above defined tree (for all input vectors). This algorithm has the same messages exchange as the last causal messages sent by the chosen algorithm for a specific input vector. Therefore, the tree based algorithm just introduced, has worst case message and time costs which are bounded by a particular execution of the original algorithm (not necessarily the worst one). This implies that its time and message delays are bounded by the worst case delay of the original one.

•

Theorem 6 now follows as a corollary.