Design and Analysis of Distributed Algorithms

Chapter 2:

BASIC PROBLEMS AND PROTOCOLS

©2005, Nicola Santoro.

Contents

1 SINGLE-INITIATOR COMPUTATIONS 1

1.1 Broadcast . . . 2

1.1.1 The Problem . . . 2

1.1.2 Cost of Broadcasting . . . 2

1.1.3 Broadcasting in Special Networks . . . 4

1.2 Traversal . . . 8

1.2.1 Depth-First Traversal . . . 8

1.2.2 Hacking $\star$ . . . 10

1.2.3 Traversal in Special Networks . . . 14

1.2.4 Considerations on Traversal . . . 17

1.3 Practical Implications: Use a Subnet . . . 18

1.4 Constructing a Spanning Tree with a Single Initiator . . . 19

1.4.1 Spanning Tree Construction . . . 19

1.4.2 Protocol Shout . . . 20

1.4.3 SPT Construction via Global Protocols . . . 25

1.4.4 Considerations on the Constructed Tree . . . 27

1.4.5 Application: Better Traversal . . . 28

2 MULTIPLE-INITIATORS COMPUTATIONS 29

2.1 Wake-Up . . . 29

2.1.1 Generic Wake-Up . . . 29

2.1.2 WakeUp in Special Networks . . . 31

2.2 Spanning-Tree Construction . . . 35

2.2.1 Impossibility Result . . . 36

2.2.2 SPT with Initial Distinct Values . . . 38

3 SATURATION AND COMPUTATIONS IN TREES 43

3.1 Saturation: A Basic Technique . . . 44

3.2 Minimum Finding . . . 47

3.3 Distributed Function Evaluation . . . 48

3.3.1 Semigroup Operations . . . 48

3.3.2 Cardinal Statistics . . . 51

3.4 Finding Eccentricities 51
3.5 Center Finding 54
3.5.1 A Simple Protocol 54
3.5.2 A Refined Protocol 54
3.5.3 An Efficient Plug-In 56
3.6 Other Computations 58
3.6.1 Finding a Median 58
3.6.2 Finding Diametral Paths 59
3.7 Computing in Rooted Trees 60
3.7.1 Rooted Trees 60
3.7.2 Convergecast 61
3.7.3 Totally Ordered Trees 62
3.7.4 Application: Termination Detection 63

4 SUMMARY **64**
4.1 Summary of Problems 64
4.2 Summary of Techniques 65

5 Bibliographical Notes **65**

6 Exercises, Problems, and Answers **66**
6.1 Exercises 66
6.2 Problems 69
6.3 Answers to Exercises 70

The aim of this chapter is to introduce same of the more basic primitive computational problems and solution techniques. These problems are basic in the sense that their solution is commonly (sometimes, frequently) required for the functioning of the system (e.g., *broadcast* and *wakeup*); they are primitive in the sense that their computation is often a preliminary step or a module of complex computations and protocols (e.g., *traversal* and *spanning-tree construction*).

Some of these problems, by their nature, are started by a single entity, while others have no such a restriction. The computational differences created by the additional assumption of a single initiator can be dramatic; we will examine the two settings separately.

Included also in this chapter are the (multiple-initiators) computations in tree networks. Their fundamental importance derives from the fact that most *global* problems (i.e., problems that, to be solved, require the involvement of all entities), oftentimes can be correctly, easily, and efficiently solved by designing a protocol for trees, and executing it on a spanning-tree of the network.
All the problems considered here require, for their solution, the *Connectivity* (CN) restriction (i.e., every entity must be reachable from every other entity). In general, and unless otherwise stated, we will also assume *Total Reliability* (TR), and *Bidirectional Links* (BL). These three restrictions are commonly used together, and the set $\mathbf{R}$ = {BL, CN, TR} will be called set of *standard* restrictions.

The techniques we introduce in this chapter to solve these problems are basic ones; once properly understood, they form a powerful and essential *toolset* that can be effectively employed by every designer of distributed algorithms.

1 SINGLE-INITIATOR COMPUTATIONS

In this section we will discuss some of the most basic primitive distributed computations *broadcast*, *traversal*, and *spanning-tree construction*. The resulting protocols are important computational tools, usually employed as a preliminary step for more complex computations, or as modules within advanced protocols.

What *broadcast* and *traversal* also have in common is that, by their nature, are always started by a single entity. In other words, these two computational problems have, in their definition, the restriction *unique initiator* (UI); depending on the problem, this additional assumption can be further restricted. Unlike the other two problems, the *spanning-tree construction* problem does not have such a restriction in its definition. In this section, we will consider its solution under restriction UI; the more general case, without this restriction, will be discussed later, in Section 2 as well as in the next Chapter.

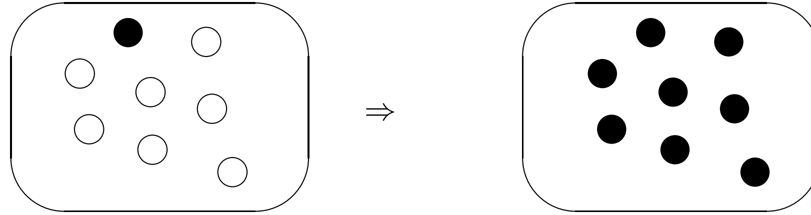

Figure 1: Broadcasting Process.

1.1 Broadcast

1.1.1 The Problem

Consider a distributed computing system where only one entity, x, knows some important information; this entity would like to share this information with *all* the other entities in the system. This problem is called *broadcasting* (**Bcast**), and we have started its examination already in the previous chapter. To solve this problem means to design a set of rules that, when executed by the entities, will lead (within finite time) to a configuration where all entities know the information; the solution must work regardless of which entity has the information at the beginning.

Built-in the definition of the problem, there is the assumption, *Unique Initiator* (UI), that only one entity will start the task. Actually, this assumption is further restricted, since the unique initiator must be the one with the initial information; we shall denote this restriction by UI+.

Clearly, every entity must be involved in the computation; however, x can send messages only to those entities to which it is connected directly (its out-neighbours). For its solution, broadcasting requires the *Connectivity* (CN) restriction (i.e., every entity must be reachable from every other entity) since in its absence, the problem is clearly unsolvable: some entities will never receive the information. We have seen a simple solution to this problem, *Flooding*, under two additional restrictions: Total Reliability (TR), and Bidirectional Links (BL). Recall that the set **R** ={BL, CN, TR} is the set of *standard* restrictions.

1.1.2 Cost of Broadcasting

As we have seen, the solution protocol *Flooding* uses $O(m)$ messages and, in the worst case, $O(d)$ ideal time units, where d is the diameter of the network.

The first and natural question is whether these costs could be reduced significantly (i.e., in order of magnitude) using a different approach or technique, and if so by how much. This question is equivalent to ask what is the *complexity* of the broadcasting problem. To answer

this type of questions we need to establish a *lower bound*: to find a bound f (typically, a function of the size of the network) and to prove that the cost of *every* solution algorithm is *at least* f. In other words, a lower bound is irrespective of the protocol and depends solely on the problem; hence, it is an indication of how complex the problem really is.

We will denote by $\mathcal{M}(\mathbf{Bcast/RI+})$ and $\mathcal{T}(\mathbf{Bcast/RI+})$ the message and the time complexity of broadcasting under $\mathbf{RI+} = \mathbf{R} \cup$ UI+, respectively.

A lower bound on the amount of ideal time units required to perform a broadcast is simple to derive: every entity must receive the information regardless of how distant they are from the initiator, and any entity could be the initiator. Hence, in the worst case,

$$\mathcal{T}(\mathbf{Bcast/RI+}) \geq Max\{d(x,y) : x, y \in V\} = d. \tag{1}$$

Since *Flooding* performs the broadcast in d ideal time units, the lower bound is *tight* (i.e., it can be achieved). In other words, we know exactly the ideal time complexity of broadcasting:

Property 1.1 *The ideal time complexity of broadcasting under* $\mathbf{RI+}$ *is* $\Theta(d)$

Let us now consider the message complexity. An obvious lower bound on the number of messages is also easy to derive: in the end, every entity must know the information; thus a message must be received by each of the $n-1$ entities who initially do not have the information. Hence,

$$\mathcal{M}(\mathbf{Bcast/RI+}) \geq n - 1.$$

With little extra effort, we can derive a more accurate lower bound:

Theorem 1.1 $\mathcal{M}(\mathbf{Bcast/RI+}) \geq m$

Proof. Assume that there exists a correct broadcasting protocol A which, in each execution under $\mathbf{RI+}$ on every G, uses fewer than $m(G)$ messages. This means that there is at least one link in G where no message is transmitted in any direction during an execution of the algorithm. Consider an execution of the algorithm on G, and let $e = (x, y) \in E$ be the link where no message is transmitted by A. Now construct a new graph G' from G by removing the edge e, and adding a new node z and two new edges $e_1 = (x, z)$ and $e_2 = (y, z)$ (see Fig. 2). Set z in a non initiator status. Run exactly the same execution of A on the new graph G': since no message was sent along (x, y), this is possible. But since no message was sent along (x, y) in the original execution, x and y never send a message to z in the current execution. As a result, z will never receive the information (i.e., change status). This contradicts the fact that A is a correct broadcasting protocol. ■

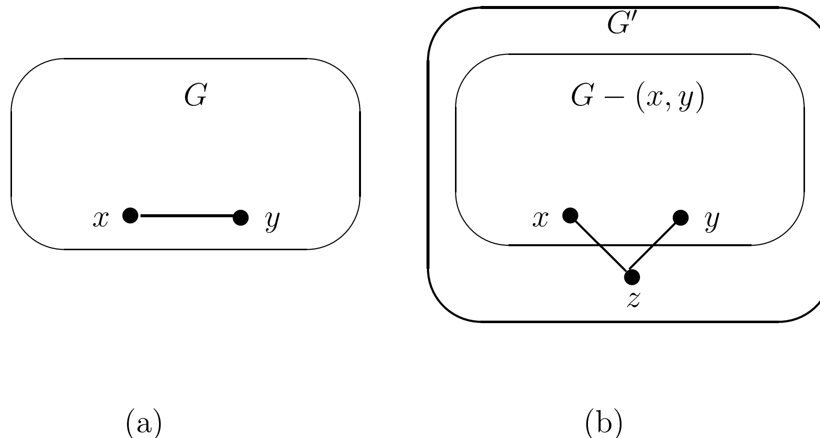

Figure 2: A message must be sent on each link.

This means that any broadcasting algorithm requires $\Omega(m)$ messages.

Since *Flooding* solves broadcasting with $2m - n + 1$ messages (see Exercise 6.1), this implies $\mathcal{M}(\textbf{Bcast}/\textbf{RI}+) \leq 2m - n + 1$. Since the upper-bound and the lower-bound are of the same order of magnitude, we can summarize

Property 1.2 *The message complexity of broadcasting under* **RI**+ *is* $\Theta(m)$

The immediate consequence is that, in order of magnitude, *Flooding* is a *message-optimal* solution. Thus, if we want to design a new protocol to improve the $2m - n + 1$ cost of *Flooding*, the best we can hope to achieve is to reduce the constant 2; in any case, because of Theorem 1.1, the reduction cannot bring the constant below 1.

1.1.3 Broadcasting in Special Networks

The results we have obtained so far apply to *generic* solutions; that is solutions that do not depend on G, and can thus be applied regardless of the communication topology (provided it is undirected and connected).

We will consider next performing the broadcast in special networks. Throughout we will assume the standard restrictions plus UI+.

Broadcasting in Trees

Consider the case when G is a tree; that is, G is connected and contains no cycles. In a tree, $m = n - 1$; hence, the use of protocol *Flooding* for broadcasting in a tree will cost

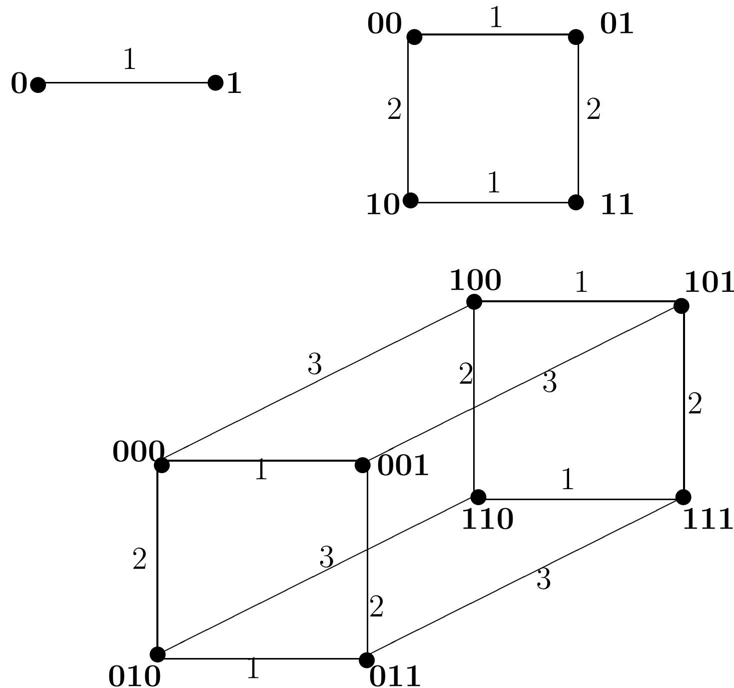

Figure 3: Labelled Hypercube Networks

$2m - (n-1) = 2(n-1) - (n-1) = n-1$ messages.

IMPORTANT. This cost is achieved even if the entities do *not know* that the network is a tree.

Broadcasting in Labeled Hypercubes A communication topology which is commonly used as an interconnection network is the (k-dimensional) labeled *hypercube*, denoted by H_k.

A labeled hypercube H_1 of dimension $k = 1$ is just a pair of nodes called (in binary) "0" and "1", connected by a link labelled "1" at both nodes.
A hypercube H_k of dimension $k > 1$ is obtained by taking two hypercubes of dimension $k-1$, H'_{k-1} and H''_{k-1}, and connecting the nodes with the same name with a link labelled k at both nodes; the name of each node in H'_{k-1} (resp. H''_{k-1}) is then modified by prefixing it with the bit 0 (resp., 1); see Fig. 3.
So, for example, node "0010" in H'_4 will be connected to node "0010" in H''_4 by a link labeled $l = 5$, and their names will become "00010" and "10010", respectively.

This labelling λ of the links is symmetric (i.e., $\lambda_x(x,y) = \lambda_y(x,y)$), and is called the *dimensional* labelling of a hypercube.

IMPORTANT. These names are used only for descriptive purposes; they are *not* known to the entities. On the other hand, the labels of the links (i.e., the port numbers) are known to the entities by the Local Orientation axiom.

A hypercube of dimension k has $n = 2^k$ nodes; each node has k links, labelled $1, 2, \ldots, k$. Hence the total number of links is $m = nk/2 = O(n \log n)$.

A straightforward application of *Flooding* in a hypercube will cost $2m-(n-1) = nk-(n-1) = O(n \log n)$ messages. However, hypercubes are highly structured networks with many very interesting properties. We can exploit these special properties to construct a more efficient broadcast. Obviously, if we do so, the protocol cannot be used in other networks.

Consider the following simple strategy.

STRATEGY *HyperFlood*:
(1) The initiator sends the message to all its neighbours.
(2) A node receiving a message from the link labelled l, will send the messages only to those neighbours with label $l' < l$.

NOTE. The only difference with the normal *flooding* is in step 2: instead of sending the message to *all* neighbours except the sender, the entity will forward it only to some of them; which ones will depend on the label of the port from where the message is received.

As we will see. this strategy correctly performs the broadcast using only $n-1$ messages (instead of $O(n \log n)$). Let us first examine termination and correctness.

Let $H_k(x)$ denote the subgraph of H_k induced by the links where messages are sent by *HyperFlood* when x is the initiator. Clearly every node in $H_k(x)$ will receive the information.

Lemma 1.1 *HyperFlood correctly terminates.*

Proof. Let x be the initiator; starting from x, the messages are sent only on links with *decreasing* labels: if y receives the message from link 4 it will forward it only to the ports $1, 2,$ and 3. To prove that every entity will receive the information sent by x, we need to show that, for every node y, there is a path from x to y such that the sequence of the labels on the path from x to y is decreasing. (Note that the labels on the path do not need to be consecutive integers.) To do so we will use the following property of hypercubes.

Property 1.3 *In a k dimensional hypercube H_k, any node x is connected to any other node y by a path $\pi \in \Pi[x, y]$ such that $\Lambda(\pi)$ is a decreasing sequence.*

Proof. Consider the k-bit names of x and of y in H_k: $< x_k, x_{k-1}, \ldots, x_1, x_0 >$ and $< y_k, y_{k-1}, \ldots, y_1, y_0 >$. If $x \neq y$, these two strings will differ in $t \geq 1$ positions. Let $j_1, j_2, \ldots, j_t$ be those positions in decreasing order; i.e., $j_i > j_{i+1}$. Consider now the nodes $v_0, v_1, v_2, \ldots, v_t$, where $v_0 = x$, and the name of v_i differs from the name of v_{i+1} only in the j_{i+1}-th position. Thus, there is a link labeled j_{i+1} connecting v_i to v_{i+1}, and clearly $v_t = y$. But this means that $< v_0, v_1, v_2, \ldots, v_t >$, is a path from x to y and the sequence of labels on this path is $< j_1, j_2, \ldots, j_t >$ which is decreasing. ■

Thus, $H_k(x)$ is connected and spans (i.e., it contains all the nodes of) H_k, regardless of x. In other words, within finite time, every entity will have the information. ■

Let us now concentrate on the cost of *HyperFlood*.

Lemma 1.2
M [HyperFlood/H_k] = n - 1.
T [HyperFlood/H_k] = k.

Proof. To prove that only $n - 1$ messages will be sent during the broadcast, we just need to show that every entity will receive the information only once. This is true because, for every x, $H_k(x)$ contains no cycles (see Exercise 6.9). Also as an exercise it is left the proof that for every x, the eccentricity of x in $H_k(x)$ is k (see Exercise 6.10); this implies that the ideal time delay of *HyperFlood* in H_k is always k. ■

These costs are the best any broadcast algorithm can perform in a hypercube regardless of how much more knowledge they have. However, they are obtained here under the additional restriction that *the network is a k-dimensional hypercube with a dimensional labeling*; that is, under $H = \{(G, \lambda) = H_k\}$. Summarizing, we have

Property 1.4 *The message complexity of broadcasting a k-dimensional hypercube with a dimensional labeling under* **RI+** *is* $\Theta(n)$

IMPORTANT. The reason why we are able to "bypass" the $\Omega(m)$ lower bound expressed by Theorem 1.1 is because we are restricting the applicability of the protocol.

Property 1.5 *The ideal time complexity of broadcasting a k-dimensional hypercube with a dimensional labeling under* **RI+** *is* $\Theta(k)$

Broadcasting in Complete Graphs

Among all network topologies, the *complete graph* is the one with the most links: every entity is connected to all others; thus $m = n(n-1)/2 = O(n^2)$ (recall we are considering bidirectional links), and $d = 1$.

The use of a generic protocol will require $O(n^2)$ messages. But this is really unnecessary.

Broadcasting in a complete graph is easily accomplished: since everybody is connected to everybody else, the initiator just needs to send the information to its neighbours (i.e., execute the command "**send**(I) **to** $N(x)$") and the broadcast is completed. This uses only $n - 1$ messages and $d = 1$ ideal time !

Clearly this protocol, *KBcast*, works only in a complete graph, that is under the additional restriction $K \equiv$ "*G is a complete graph*". Summarizing:

Property 1.6
$\mathcal{M}$(**Bcast**/**RI+** ; $K) = n - 1$
$\mathcal{T}$(**Bcast**/**RI+** ; $K) = 1$

1.2 Traversal

Traversal of the network allows every entity in the network to be "visited" sequentially (one after the other). Its main use is in the control and management of a shared resource and in sequential search processes. In abstract terms, the *traversal problem* starts with an initial configuration where all entities are in the same state (say *unvisited*) except one that is *visited* and is the sole initiator; the goal is to render all the entities visited but sequentially (i.e., one at the time).

A *traversal protocol* is a distributed algorithm that, starting from the single initiator, allows a special message called "traversal token" (or simply, *token*), to reach every entity *sequentially* (i.e., one at the time). Once a node is reached by the token, it is marked as "visited". Depending on the traversal strategy employed, we will have different traversal protocols.

1.2.1 Depth-First Traversal

A well known strategy is the *depth-first traversal* of a graph. According to this strategy, the graph is visited (i.e., the token is forwarded) trying to go forward as long as possible; if it is forwarded to an already visited node, it is sent back to the sender, and that link is marked as a *back-edge*; if the token can no longer be forwarded (it is at a node where all its neighbours have been visited), the algorithm will "backtrack" until it finds an unvisited node where the token can be forwarded to.

The distributed implementation of depth-first traversal is straightforward.
(1) When first visited, an entity remembers who sent the token, creates a list of all its still unvisited neighbours, forwards the token to one of them (removing it from the list), and waits for its reply returning the token.
(2) When the neighbour receives the token, it will return the token immediately if it had been visited already by somebody else, notifying that the link is a back-edge; otherwise, it will first forward the token to each of its unvisited neighbours sequentially, and then reply returning the token.
(3) Upon reception of the reply, the entity forwards the token to another unvisited neighbour.
(4) Should there be no more unvisited neighbours, the entity can no longer forward the token; it will then send the reply to the node from which it first received the token.

NOTE. When the neighbour in step (2) determines that a link is a back-edge , it knows that the sender of the token is already visited; thus, it will remove it from the list of unvisited neighbours.

We will use three types of messages: "T" to forward the token in the traversal, "Backedge" to notify the detection of a back-edge, and "Return" to return the token upon local termination.

Protocol *DF_Traversal* is shown in Fig. 4, where the operation of extracting an element from a set B and assigning it to variable a is denoted by $a \Leftarrow B$. Let us examine its costs.

Theorem 1.2 **T**[DF_Traversal] = **M**[DF_Traversal]= $2m$

Proof. Focus on a link $(x, y) \in E$. What messages can be sent on it? Suppose x sends

PROTOCOL DF_Traversal.

- Status: $\mathcal{S} = \{$INITIATOR,IDLE,VISITED,DONE$\}$;
  $\mathcal{S}_{INIT} = \{$INITIATOR,IDLE$\}$; $\mathcal{S}_{TERM} = \{$DONE$\}$.
- Restrictions: **R** ;UI.

```
INITIATOR
        Spontaneously
        begin
             Unvisited:= N(x);
             initiator:= true;
             VISIT;
        end

IDLE
        Receiving(T)
        begin
             entry:=sender;
             Unvisited:= N(x) − {sender};
             initiator:= false;
             VISIT;
        end

VISITED
        Receiving(T)
        begin
             Unvisited:= Unvisited −{sender};
             send(Backedge) to {sender};
        end

        Receiving(Return)
        begin
             VISIT;
        end

        Receiving(Backedge)
        begin
             VISIT;
        end

Procedure VISIT
begin
     if Unvisited ≠ ∅ then
        next ⇐ Unvisited;
        send(T) to next;
        become VISITED
     else
        if not(initiator) then send(Return) to entry; endif
        become DONE;
     endif
end
```

Figure 4: DF_Traversal

T to y; then y will only send to x either Return (if it was *idle* when the T arrived) or Backedge (otherwise). In other words, on each link there will be exactly two messages transmitted. Thus, $\mathbf{M}$[*DF_Traversal*]$= 2m$. Since the traversal is sequential, $\mathbf{T}$[*DF_Traversal*] = $\mathbf{M}$[*DF_Traversal*]. ■

To determine how efficient is the protocol, we are going to determine what is the complexity of the problem.

Using exactly the same technique we employed in the proof of Theorem 1.1, we have (Exercise 6.11):

Theorem 1.3 $\mathcal{M}(\mathbf{DFT}/\,\mathbf{R}) \geq m$

Therefore, the $2m$ message cost of protocol *DF_Traversal* is indeed excellent, and the protocol is *message optimal*.

Property 1.7 *The message complexity of depth-first traversal under* $\mathbf{R}$ *is* $\Theta(m)$

The time requirements of a depth-first traversal are quite different from those of a broadcast. In fact, since each node must be visited sequentially, starting from the sole initiator, the time complexity is at least the number of nodes:

Theorem 1.4 $\mathcal{T}(\mathbf{DFT}/\,\mathbf{R}) \geq n-1$

The time complexity of protocol *DF_Traversal* is dreadful. In fact, the upperbound $2m$ could be several order of magnitude larger than the lowerbound $n-1$. For example, in a complete graph, $2m = n^2 - n$. Some significant improvements in the time complexity can however be made by going into a finer granularity. We will discuss this topic in greater details later.

1.2.2 Hacking $\star$

Let us examine protocol *Protocol DF_Traversal* to see if it can be improved, especially its time cost.

IMPORTANT. When measuring ideal time, we consider only *synchronous* executions; however, when measuring messages and establishing correctness we must consider *every* possible schedule of events, especially the non-synchronous executions.

Basic Hacking

The protocol we have constructed is totally sequential: in a synchronous execution, at each time unit only one message will be sent, and every message requires one unit of time. So, to improve the time complexity, we need to (1) reduce the number of messages, and/or (2) introduce some concurrency.

By definition of traversal, each entity must receive the token (message T) at least once. In the execution of our protocol, however, some entities receive it more than once; those links from which these other T messages arrive are precisely the back-edges.

Question. Can we avoid sending T messages on back-edges?

To answer this question we must understand why T messages are sent on back-edges. When an entity x sends a T message to y, it does not know whether the link is a back-edge or not; that is, whether y has already been visited by somebody else or not. If x knew which of its neighbours are already visited, it would not send a T message to them, there would be no need for Backedge messages from them, and we would be saving messages and time. Let us examine how to achieve such a condition.

Suppose that, whenever a node is visited (i.e., it receives T) for the first time, it notifies all its (other) neighbours of this event (e.g., sending a "Visited" message), and waits for an acknowledgment (e.g., receiving a "Ack" message) from them before forwarding the token.

The consequence of such a simple act is that now an entity ready to forward the token (i.e., to send a T message) really knows which of its neighbours have already been visited.

This is exactly what we wanted. The price we have to pay is the transmission of the Visited and Ack messages.

Notice that now an *idle* entity (that is an entity that has not yet been involved in the traversal) might receive a Visited message as its first message. In the revised protocol, we will make such an entity enter a new status, *available*.

Let us examine the effects of this change on the overall *time cost* of the protocol; call *DF+* the resulting protocol. The time is really determined by the number of sequential messages. There are four types of messages which are sent: T, Return, Visited, and Ack.

Each entity (except the initiator) will receive only one T message and send only one Return message; the initiator does not receive any T message and does not send any Return; thus, in total there will be $2(n-1)$ such messages. Since all these communications occur sequentially (i.e., without any overlap), the time taken by sending the T and Return messages will be $2(n-1)$.

To determine how many ideal time units are added by the transmission of Visited and Ack messages, consider an entity: its transmission of all the Visited messages takes only a single time unit, since they are sent concurrently; the corresponding Ack messages will also be sent concurrently, adding an additional time unit. Since every node will do it, the sending of the Visited messages and receiving the Ack messages will increase the ideal time of the original algorithm by exactly $2n$.

This will give us a time cost of

$$\mathbf{T}[DF+] = 4n - 2. \tag{2}$$

How many *messages* this will cost is also easy to compute. As mentioned above, there is a total of $2(n-1)$ T and Return messages. In addition, each entity (except the initiator) sends a Visited message to all its neighbours except the one from which it received the token; the initiator will send it to all its neighbours. Thus, denoting by s the initiator, the total

number of Visited messages is $|N(s)| + \sum_{x \neq s}(|N(x)| - 1) = 2m - (n-1)$. Since for each Visited message there will be an Ack, the total message cost will be

$$\mathbf{M}[DF+] = 4m - 2(n-1) + 2(n-1) = 4m \quad (3)$$

Summarizing, we have been able to reduce the time costs from $O(m)$ to $O(n)$ which, because of Theorem 1.4, is *optimal.* The price has been the doubling of the number of messages.

Property 1.8 *The ideal time complexity of depth-first traversal under* $\mathbf{R}$ *is* $\Theta(n)$

Advanced Hacking

Let us see if the number of messages can be decreased without significantly increasing the time costs.

Question. Can we avoid sending the Ack messages?

To answer this question we must understand what would happen if we do not send Ack messages. Consider an entity x that sends Visited to its neighbours; (if we no longer use Ack) x will proceed immediately with forwarding the token. Assume that, after some time, the token arrives, for the first time, to a neighbour z of x (see Figure **??**); it is possible that the Visited message sent by x to z has not arrived yet (due to communication delays). In this case, z would not know that x has already been visited, and would send the T message to it. That is, we will again send a T message on a back-edge undoing what we had accomplished with the previous change to the protocol !

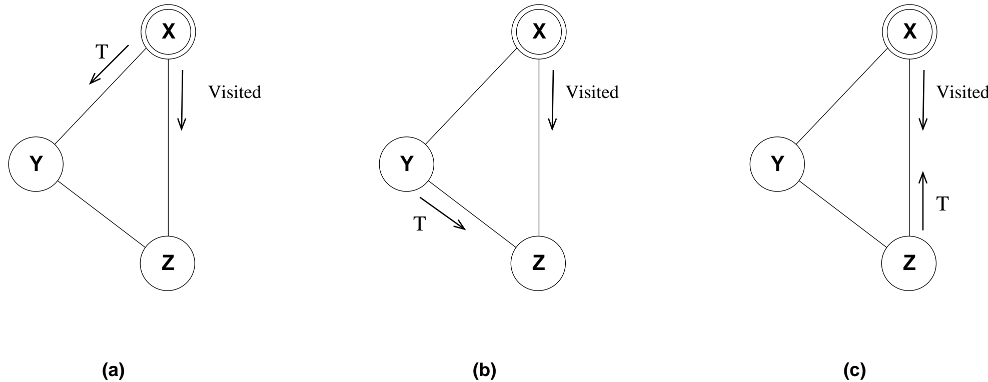

Figure 5: Slow *Visited* message : z does not know that x has been visited.

But the algorithm now is rather different (we are using Visited messages, no longer Backedge messages) and this situation might not happen all the time.

Still, if it happens, z will eventually receive the Visited message from x (recall we are operating under total reliability); z can then understand its mistake, pretend nothing happened

(just the waste of a T message), and continue like that T message was never really sent. On its side, x upon receiving the token will also understand that z made a mistake, and ignore the message; x also realizes (if it did not know already) that z is visited and will remove it from its list of unvisited neighbours.

Although the correctness will not be affected (Exercise 6.15), mistakes cost additional messages. Let us examine what is really the cost of this modified protocol, which we shall call *DF++*.

As before, the "correct" T and Return yield a total of $2n-2$ messages, and the Visited messages are $2m-n+1$ in total.

Then there are the "mistakes"; each mistake costs one message. The number of mistakes can be very large. In fact, unfriendly time delays can force mistakes to occur on *every* back-edge; on some backedges, there can be two mistakes, one in each direction ! (Exercise 6.16). In other words, there will be at most $2(m-n+1)$ incorrect T messages. Summing all up, this yields:

$$\mathbf{M}[DF++] \leq 4m-n+1. \tag{4}$$

Let us consider now the time. We have an improvement in that the Ack messages are no longer sent, saving n time units.

Since there are no more Ack to wait for, an entity can forward the token at the same time as the transmission of the Visited messages; if it does not have any unvisited neighbour to send the T to, the entity will send the Return at the same time as the Visited. Hence, the sending of the Visited is done in overlap with the sending of either a T or a Return message, saving another n time units.

In other words, without considering the mistakes, the total time will be $2n-2$. Let us now consider also the mistakes and evaluate the ideal time of the protocol.

Strange as it might sound, when we attempt to measure the ideal execution time of this protocol, in the execution *no mistakes will ever occur* ! This is because mistakes can only occur due to *arbitrarily long* communication delays; on the other hand, ideal time is only measured under *unitary* delays. But under unitary delays there are no mistakes. Therefore,

$$\mathbf{T}[DF++] = 2n-2 \tag{5}$$

IMPORTANT. It is crucial to understand this inherent limit of the cost measure we call *ideal time*. Unlike the number of messages, ideal time is not a "neutral" measure; it influences (thus limiting) the nature of what we want to measure. In other words, it should be treated and handled with caution. Even greater caution should be employed in interpreting the results it gives.

Extreme Hacking

Since we are on a roll, let us observe that we could actually use the T message as an implicit

Visited, saving some additional messages.

This saving will happen at every entity except those that, when they are reached for the first time by a T message, do not have any unvisited neighbour. Let $f_\star$ denote the number of these nodes; thus the number of Visited messages we save is $n - f_\star$. Hence, the total number of messages is $4m - n + 1 - n + f_\star$.

Summarizing, the cost of the optimized protocol, called *DF** and described in Figures 6 and 7, is as follows.

Theorem 1.5
$\mathbf{M}[DF*] = 4m - 2n + f_* + 1$
$\mathbf{T}[DF*] = 2n - 2$

IMPORTANT. The value of $f_\star$, unlike n and m, is *not* a system parameter. In fact, it is *execution-dependent*: it may change at each execution ! We shall indicate this fact (for f as well as for any other execution dependent value) by the use of the subscript $\star$.

1.2.3 Traversal in Special Networks

Trees
In a tree network, depth-first traversal is particularly efficient in terms of messages, and there is no need of any optimization effort (hacking). In fact, in any execution of *DF_Traversal* in a tree, no Backedge messages will be sent (Exercise 6.12). Hence, the total number of messages will be exactly $2(n-1)$. The time complexity is the same as the optimized version of the protocol: $2(n-1)$.

$$\mathbf{M}[DF_Traversal/Tree] = \mathbf{T}[DF_Traversal/Tree] = 2n - 2 \quad (6)$$

An interesting side effect of a depth-first traversal of a tree, is that it constructs a *virtual ring* on the tree. (Figure 8). In this ring some nodes appear more than once; in fact the ring has size $2n - 2$ (Exercise 6.13). This fact will have useful consequences.

Rings
In a ring network, every node has exactly two neighbours. Depth-first traversal in a ring can be achieved in a simple way: the initiator chooses one direction and the token is just forwarded along that direction; once the token reaches the initiator, the traversal has been completed. In other words, each entity will send and receive a single T message. Hence both the time and the message costs are exactly n. Clearly this protocol can be used only in rings.

Complete Graph
In a complete graph, execution of *DF** will require $O(n^2)$ messages. Exploiting the knowledge of being in a complete network, a better protocol can be derived: the initiator sequentially will sends the token to all its neighbours (which are all the other entities in the network);

PROTOCOL DF*

- Status: $\mathcal{S}$ = {INITIATOR,IDLE,AVAILABLE,VISITED,DONE};
  $\mathcal{S}_{INIT}$ = {INITIATOR,IDLE}; $\mathcal{S}_{TERM}$ = {DONE}.
- Restrictions: **R** ;UI.

```
INITIATOR
        Spontaneously
        begin
             initiator:= true;
             Unvisited:= N(x);
             next ⇐ Unvisited;
             send(T) to next;
             send(Visited) to N(x)-{next};
             become VISITED
        end

IDLE
        Receiving(T)
        begin
             Unvisited:= N(x);
             FIRST-VISIT;
        end

        Receiving(Visited)
        begin
             Unvisited:= N(x) − {sender};
             become AVAILABLE
        end

AVAILABLE
        Receiving(T)
        FIRST-VISIT;

        Receiving(Visited)
        begin
             Unvisited:= Unvisited − {sender};
        end

VISITED
        Receiving(Visited)
        begin
             Unvisited:= Unvisited −{sender};
             if next = sender then VISIT; endif
        end

        Receiving(T)
        begin
             Unvisited:= Unvisited −{sender};
             if next = sender then VISIT; endif
        end

        Receiving(Return)
        begin
             VISIT;
        end
```

Figure 6: Protocol DF*

```
Procedure FIRST-VISIT
begin
     initiator:= false;
     entry:=sender;
     Unvisited:= Unvisited-{sender};

     if Unvisited ≠ ∅ then
       next ⇐ Unvisited;
       send(T) to next;
       send(Visited) to N(x)−{entry,next};
       become VISITED;
     else
       send(Return) to {entry};
       send(Visited) to N(x)−{entry};
       become DONE;
     endif
end

Procedure VISIT
begin
     if Unvisited ≠ ∅ then
       next ⇐ Unvisited;
       send(T) to next;
     else
       if not(initiator) then send(Return) to entry; endif
       become DONE;
     endif
end
```

Figure 7: Routines used by Protocol DF*

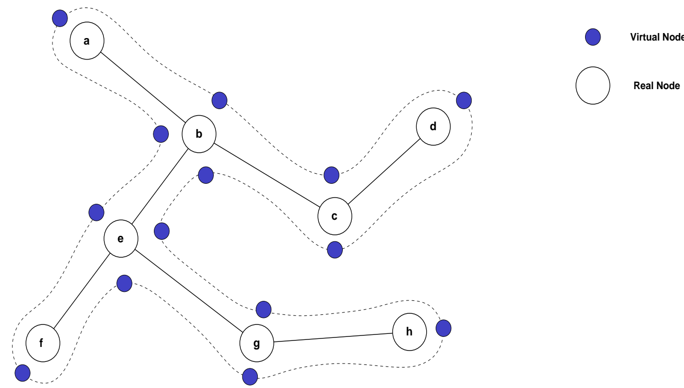

Figure 8: Virtual ring created by df-traversal.

each of this entities will return the token to the initiator without forwarding it to anybody else. The total number of messages is $2(n-1)$, and so is the time.

1.2.4 Considerations on Traversal

Traversal as Access Permission
The main use of a traversal protocol is in the control and management of shared resources.

For example, access to a shared transmission medium (e.g., bus) must be controlled to avoid *collisions* (simultaneous frame transmission by two or more entities). A typical mechanism to achieve this is by the use of a *control* (or *permission*) *token*. This token is passed from one entity to another according to the same set of rules. An entity can only transmit a frame when it is in possession of the token; once the frame has been transmitted, the token is passed to another entity. A traversal protocol by definition "passes" the token sequentially though all the entities, and thus solves the access control problem. The only proviso is that, for the access permission problem, it must be made *perennial*: once a traversal is terminated, another must be started by the initiator.

The access permission problem is part of a family of problems commonly called *Mutual Exclusion*, which will be discussed in details later in the book.

Traversal as Broadcast
It is not difficult to see that *any traversal protocol solves the broadcast problem*: the initiator puts the information in the token message; every entity will be visited by the token and thus will receive the information. The converse is not necessarily true; for example, *Flooding* violates the *sequentiality* requirement since the message is sent to all (other) neighbours simultaneously.

The use of traversal to broadcast does not lead to a more efficient broadcasting protocol. In fact, a comparison of the costs of *Flooding* and *DF** (Theorems **??** and 1.5) shows that *Flood-*

ing is more efficient in terms of both messages and ideal time. This is not surprising since a traversal is constrained to be sequential; flooding, on the other hand, exploits concurrency at its outmost.

1.3 Practical Implications: Use a Subnet

We have considered two basic problems (*broadcast* and *depth-first traversal*) and studied their complexity, devised solution protocols and analyzed their efficiency. Let us see what the *theoretical* results we have obtained tell us about the situation from a *practical* point of view.

We have seen that generic protocols for broadcasting require $\Omega(m)$ messages (Theorem 1.1). Indeed, in some special networks, we can sometimes develop topology-dependent solutions and obtain some improvements.

A similar situation exists for generic traversal protocols: they all require $\Omega(m)$ messages (Theorem 1.3); this cost cannot be reduced (in order of magnitude) unless we make additional restrictions, for example exploiting some special properties of G of which we have a priori (i.e., at design time) knowledge.

In any connected undirected graph G, we have

$$(n^2 - n)/2 \;\geq\; m \;\geq\; n - 1,$$

and, for every value in that range, there are networks with those many links; in particular, $m = (n^2 - n)/2$ occurs when G is the *complete* graph, and $m = n - 1$ when G is a *tree*.

Summarizing, the cost of broadcasting and traversal depends on the number of links: the more links the greater the cost; and it can be as bad as $O(n^2)$ messages per execution of any of the solution protocols.

This result is punitive for networks where a large investment has been made in the construction of communication links. Since broadcast is a basic communication tool (in some systems, it is a primitive) dense networks are penalized continuously. Similarly, larger operating costs will be incurred by dense networks every time a traversal (fortunately, not such a common operation) is performed.

The theoretical results, in other words, indicate that *investments in communication hardware* will result in *higher operating communication costs*.

Obviously, this is not an acceptable situation, and it is necessary to employ some "lateral thinking".

The strategy to circumvent the obstacle posed by these lower-bounds (Theorems 1.1 and 1.3) without restricting the applicability of the protocol is fortunately simple:

(1) construct a subnet G' of G;
(2) perform the operations only on the subnet.

If the subnet G' we construct is *connected* and *spans* G (that is, contains all nodes of G), then doing broadcast on G' will solve the broadcasting problem on G: every node (entity) will receive the information. Similarly, performing a traversal on G' will solve that problem on G.

The important consequence is that, if G' is a proper subnet, it has fewer links than G; thus, the cost of performing those operations on G' will be lower than doing it in G.

Which connected spanning subnet of G should we construct ?

If we want to minimize the message costs, we should choose the one with the fewest number of links; thus, the answer is: a *spanning tree* of G. So, the strategy for a general graph G will be

STRATEGY Use-a-Tree
(1) construct a spanning tree of G;
(2) perform the operations only on this spanning tree.

This strategy has two costs.

First, there is the cost of constructing the spanning tree; this task will have to be carried out only once (if no failures occur).

Then there are the operating costs, that is the costs of performing broadcast and traversal on the tree. Broadcast will cost exactly $n-1$ messages, and the cost of traversal will be twice that amount. These costs are *independent* of m and thus do not inhibit investments in communication links (which might be useful for other reasons).

1.4 Constructing a Spanning Tree with a Single Initiator

1.4.1 Spanning Tree Construction

Spanning tree construction (**SPT**) is a classical problem in computer science. In a distributed computing environment, the solution of this problem has, as we have seen, strong practical motivations. It also has distinct formulation and requirements.

In a distributed computing environment, to construct a spanning tree of G means to move the system from an initial system configuration where each entity is just aware of its own neigbours, to a system configuration where
(1) each entity x has selected a subset Tree-neighbours(x) $\subseteq N(x)$, and
(2) the collection of all the corresponding links form a spanning tree of G.

What is wanted is a distributed algorithm (specifying what each node has to do when receiving a message in a given status) such that, once executed, guarantees that a spanning tree $T(G)$ of G has been constructed; in the following we will indicate $T(G)$ simply by T, if no ambiguity arises.

Note that T is not known a priori to the entities, and might not be known after it has been constructed: an entity needs to know only which of its neighbours are also its neighbours in the spanning tree T.

As before, we will restrict ourselves to connected networks with bidirectional links and further assume that no failure will occur; we will also assume that the construction will be started by only one entity (i.e., Unique Initiator (UI) restriction); that is we will consider spanning-tree construction under restrictions **RI**.

1.4.2 Protocol Shout

Consider the entities; they do not know G, not even its size. The only things an entity is aware of are the labels on the ports leading to its neighbours (because of the Local Orientation axiom), and the fact that, if it sends a message to a neighbour, the message will eventually be received (because of the Finite Communication Delays axiom and the Total Reliability restriction).

How, using just this information, can a spanning tree be constructed?

The answer is surprisingly simple. Each entity needs to know which of its neighbours are also neighbours in the spanning tree. The solution strategy is: *just "ask"*:

STRATEGY Ask-your-Neighbours

(1) The initiator s will "ask" its neighbours; that is, it will sends a message $Q =$ (*"Are you my neighbour in the spanning tree?"*) to all its neighbours.

(2) An entity $x \neq s$ will reply *"Yes"* only the first time it is asked and, in this occasion, it will ask all its other neighbours; otherwise, it will reply *"No"*. The initiator s will always reply *"No"*.

(3) Each entity terminates when it has received a reply from all neighbours to which it asked the question.

For an entity x, its neighbours in the spanning tree T are the neighbours that have replied *"Yes"* and, if $x \neq s$, also the neighbour from which the question was first asked.

The corresponding set of rules is depicted in Fig. 9 where in bold are shown the tree links, in dotted lines the non-tree links. The protocol *Shout* implementing this strategy is shown in Figure 10. Initially, all nodes are in status *available* except the sole *initiator*.

Before we discuss the correctness and the efficiency of the protocol, consider how it is structured and operates. First of all observe that, in *Shout* the question Q is broadcasted through the network (using flooding). Further observe that, when an entity receives Q, it always sends a reply (either *Yes* or *No*). Summarizing, the structure of this protocol is a flood where every information message is acknowledged. This type of structure will be called *Flood+Reply*.

Correctness

Let us show now that *Flood+Reply*, as used above, always constructs a spanning tree; that is, the graph defined by all the *Tree-neighbours* computed by the entities form a spanning tree of G; furthermore, this tree is *rooted* in the initiator s.

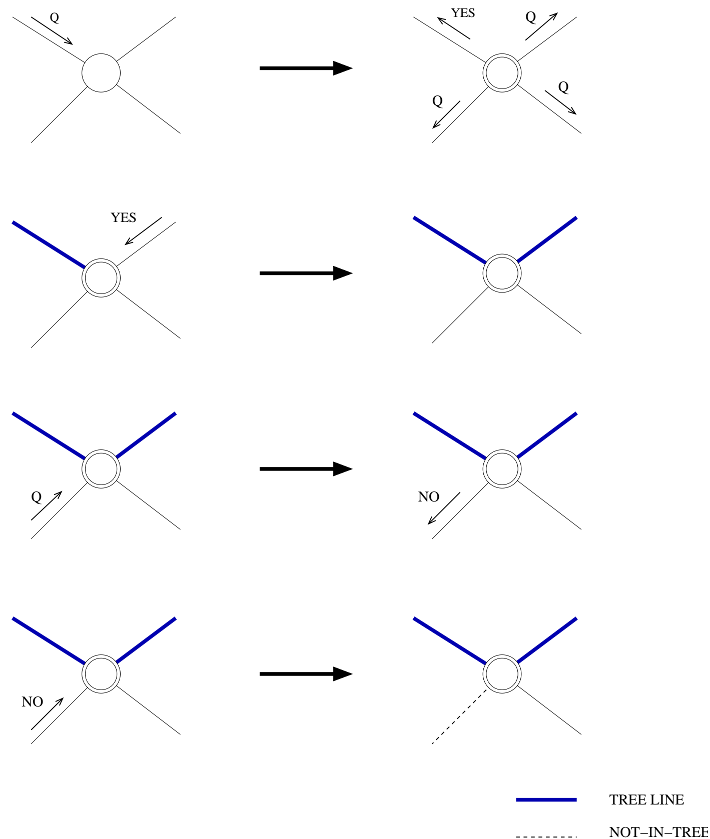

Figure 9: Set of Rules of Shout.

Theorem 1.6 *Protocol Shout correctly terminates.*

Proof. This protocol consists of the flooding of Q, where every Q message is acknowledged. Because of the correctness of flooding, we are guaranteed that every entity will receive Q, and by construction will reply (either *Yes* or *No*) to each Q it receives. Termination then follows.

To prove correctness we must show that the subnet G' defined by all the *Tree-neighbours* is a spanning tree of G. First observe that, if x is in *Tree-neighbours* of y, then y is in *Tree-neighbours* of x (see Exercise 6.18). If an entity x sends a *Yes* to y, then it is in *Tree-neighbours* of y; furthermore, it is connected to s by a path where a *Yes* is sent on each link (see Exercise 6.19). Since every $x \neq s$ sends exactly one *Yes*, the subnet G' defined by all the *Tree-neighbours* contains all the entities (i.e., it spans G), it is connected, and contains no cycles (see Exercise 6.20). Therefore, it is a spanning tree of G. ■

Note that G' is actually a tree *rooted* in the initiator. Recall that, in a rooted tree, every node (except the root) has one *parent*: the neighbour closest to the root; all its other neigh-

PROTOCOL Shout

- Status: $\mathcal{S} = \{$INITIATOR,IDLE,ACTIVE,DONE$\}$;
  $\mathcal{S}_{INIT} = \{$INITIATOR,IDLE$\}$;
  $\mathcal{S}_{TERM} = \{$DONE$\}$.
- Restrictions: **R** ;UI.

```
INITIATOR
          Spontaneously
          begin
               root:= true;
               Tree-neighbours:=∅;
               send(Q) to N(x);
               counter:=0;
               become ACTIVE;
          end

IDLE
          Receiving(Q)
          begin
               root:= false;
               parent:= sender;
               Tree-neighbours:={sender};
               send(Yes) to {sender};
               counter:=1;
               if counter=|N(x)| then
                  become DONE
               else
                  send(Q) to N(x) − {sender};
                  become ACTIVE;
               endif
          end

ACTIVE
          Receiving(Q)
          begin
               send(No) to {sender};
          end

          Receiving(Yes)
          begin
               Tree-neighbours:=Tree-neighbours ∪{sender};
               counter:=counter+1;
               if counter=|N(x)| then become DONE; endif
          end

          Receiving(No)
          begin
               counter:=counter+1;
               if counter=|N(x)| then become DONE; endif
          end
```

Figure 10: Protocol Shout

bours are called *children*. The neighbour to which x sends a *Yes* is its *parent*; all neighbours from which it receives a *Yes* are its *children*. This fact can be useful in subsequent operations.

IMPORTANT The execution of protocol *Shout* ends with *local termination*: each entity knows when its own execution is over; this occurs when it enters status *done*. Notice however that no entity, including the initiator, is aware of *global termination* (i.e., that every entity has locally terminated). This situation is fairly common in distributed computations. Should we need the initiator to know that the execution has terminated (e.g., to start another task), *Flood+Reply* can be easily modified to achieve this goal (Exercise 6.24).

Costs

The message costs of *Flood+Reply*, and thus of *Shout*, are simple to analyze. As mentioned before, *Flood+Reply* consists of an execution of $Flooding(Q)$ with the addition of a reply (either *Yes* or *No*) for every Q. In other words,

$$\mathbf{M}[\textit{Flood+Reply}] = 2\ \mathbf{M}[\textit{Flooding}].$$

The time costs of *Flood+Reply*, and thus of *Shout*, are also simple to determine; in fact (Exercise 6.21):

$$\mathbf{T}[\textit{Flood+Reply}] = \mathbf{T}[\textit{Flooding}]+1.$$

Thus

Theorem 1.7
$\mathbf{M}[\textit{Shout}] = 4m - 2n + 2$
$\mathbf{T}[\textit{Shout}] = r(s_\star) + 1 \leq d + 1$

The efficiency of protocol Shout can be evaluated better taking into account the complexity of the problem it is solving.

Since every node must be involved, using an argument similar to the proof of Theorem 1.1, we have:

Theorem 1.8 $\mathcal{M}(\mathbf{SPT/RI}) \geq m$

Proof. Assume that there exists a correct SPT protocol A which, in each execution under **RI** on every G, uses fewer than $m(G)$ messages. This means that there is at least one link in G where no message is transmitted in any direction during an execution of the algorithm. Consider an execution of the algorithm on G, and let $e = (x, y) \in E$ be the link where no message is transmitted by A. Now construct a new graph G' from G by removing the edge e, and adding a new node z and two new edges $e_1 = (x, z)$ and $e_2 = (y, z)$ (see Fig. 2). Set z in a non initiator status. Run exactly the same execution of A on the new graph G': since no message was sent along (x, y), this is possible. But since no message was sent along (x, y) in the original execution in G, x and y never send a message to z in the current execution in G'; and since z is not the initiator and does not receive any message, it will not send any message. Within finite time, protocol A terminates claiming that a spanning-tree T of G' has been constructed; however, z is not part of T, and hence T does not span G'. ∎

and similarly to the broadcast problem we have

Theorem 1.9 $\mathcal{T}(\mathbf{SPT}/\mathbf{RI}) \geq d$

This implies that protocol *Shout* is both *time optimal* and *message optimal* with respect to order of magnitude. In other words,

Property 1.9 *The message complexity of spanning-tree construction under* **RI** *is* $\Theta(m)$

Property 1.10 *The ideal time complexity of spanning-tree construction under* **RI** *is* $\Theta(d)$

In the case of the number of messages some improvement might be possible in terms of the constant.

Hacking

Let us examine protocol *Shout* to see if it can be improved and we can save some messages.

Question: Do we have to send *No* messages?

When constructing the spanning tree, an entity needs to know who its tree- neighbours are; by construction, they are the ones that reply *Yes* and, except for the initiator, also the one that first asked the question. Thus, for this determination, the *No* messages are not needed.

On the other hand, the *No* messages are used by the protocol to terminate in finite time. Consider an entity x that just sent Q to neighbour y; it is now waiting for a reply. If the reply is *Yes*, it knows y is in the tree; if the reply is *No*, it knows y is not. Should we remove the sending of *No*, how can x determine that y would have sent *No* ?

More clearly: suppose x has been waiting for a reply from y for a (very) long time; it does not know if y has sent *Yes* and the delays are very long, or y would have sent *No* and thus will send nothing. Because the algorithm must terminate, x cannot wait forever, and has to make a decision. How can x decide?

The question is relevant because communication delays are finite but unpredictable.

Fortunately, there is a simple answer to the question, that can be derived by examining how protocol *Shout* operates.

Focus on a node x that just sent Q to its neighbour y. Why would y reply *No* ? It would do so only if it had already said *Yes* to somebody else; if that happened, y sent at the same time Q to all its other neighbours, including x. Summarizing, if y replies *No* to x, it must have already sent Q to x. We can clearly use this fact to our advantage: after x sent Q to y, if it receives *Yes* it knows that y is its neighbour in the tree; if it receives Q, it can deduce that y will definitely reply *No* to x's question. All of this x can deduce without having received the *No*.

In other words: a message Q that arrives at a node waiting for a reply can act as an *implicit negative acknowledgment*; therefore, we can avoid sending *No* messages.

Let us now analyze the message complexity of the resulting protocol *Shout+*.

Theorem 1.10
$\mathbf{M}[\textit{Shout+}] = 2m$
$\mathbf{T}[\textit{Shout+}] = r(s_\star) + 1 \leq d + 1$

Proof. The time complexity is clearly unchanged. On each link $(x, y) \in E$ there will be exactly a pair of messages: either Q in one direction and *Yes* in the other, or two Q messages, one in each direction. Thus the total number of messages sent is $2m$. ■

1.4.3 SPT Construction via Global Protocols

SPT Construction by Traversal

It is well known that a depth-first traversal of a graph G actually constructs a spanning tree (*df-tree*) of that graph. The df-tree is obtained by removing from G the back-edges (i.e., the edges where a Backedge message was sent in *DF_Traversal*). In other words, the Tree-neighbours of an entity x will be those from which it receives a Return message and, if x is not the initiator, the one from which x received the first T.

Simple modifications to protocol *DF** will ensure that each entity will correctly compute their neighbours in the df-tree and locally terminate in finite time (Exercise 6.25). Notice that these modifications involve just local bookkeeping and no additional communication. Hence the time and message costs are unchanged. Denote by *df-SPT* the resulting protocol.

Theorem 1.11
$\mathbf{M}\ [\textit{df-SPT}] = 4m - 2n + f_* + 1$
$\mathbf{T}[\textit{df-SPT}] = 2n - 2$

We can now better characterize the variable f_* which appears in the cost above. In fact, f_* is exactly the number of *leaves* of the df-tree constructed by *df-SPT* (Exercise 6.26).

The results of Theorem 1.11, when compared with the costs of protocol *Shout*, indicate that depth-first traversal is *not* an efficient tool for constructing a spanning tree; this is particularly true for its very high time costs.

Notice that, like in protocol *Shout*, all entities will become aware of their local termination, but only the initiator will also be aware of *global termination*, i.e., that the construction of the spanning tree has been completed (Exercise 6.27).

SPT Construction by Broadcasting

We have just seen how, with simple modifications, the techniques of flooding and of df-traversal can be used to construct a spanning tree, if there is a unique initiator. This fact is part of a very interesting and more general phenomenon: under **RI**,

the execution of any *broadcast protocol constructs a spanning-tree.*

Let us examine this statement in more details. Take any broadcast protocol B; by definition of broadcast, its execution will result in all entities receiving the information initially held by the initiator. For each entity x different from the initiator, call *parent* the neighbour from which x received the information for the first time; clearly, everybody except the initiator will have only one parent, and the initiator has none. Denote by $x \succ y$ the fact that x is the parent of y; then we have the following property whose proof is left as an exercise (Exercise 6.28):

Theorem 1.12 *The parent relationship $\succ$ defines a spanning tree rooted in the initiator.*

As a consequence, it would appear that, to solve **SPT**, we just need to execute a broadcast algorithm without any real modification, just adding some local variables (Tree-neighbours) and doing some local bookkeeping.

This is generally not the case; in fact, knowing its *parent* in the tree is not enough for an entity. To solve **SPT**, when an entity x terminates its execution, it must explicitly know which neighbours are its *children* as well as which neighbour are *not* its tree-neighbours.

If not provided already by the protocol, this information can obviously be acquired. For example, if every entity sends a notification message to its parent, the parents will know their children. To find out which neighbours are *not* children is more difficult, and will depend on the original broadcast protocol.

In protocol *Shout* this is achieved by adding the "Yes" (I am your child) and "No" (I am not your child) messages to *Flooding*. In protocol *DF_Traversal* this is already achieved by the "Return" (I am your child) and the "Backedge" (I am not your child) messages; so, no additional communication is required.

This fact establishes a *computational* relationship between the broadcasting problem and the spanning-tree construction problem. If I know how to broadcast, (with minor modifications) I know how to construct a spanning-tree with a unique initiator. The converse is also trivially true: every protocol that constructs a spanning-tree solves the broadcasting problem. We shall say that these two problems are *computationally equivalent*, and denote this fact by

$$\mathbf{Bcast} \equiv \mathbf{SPT}(UI) \tag{7}$$

Since, as we have discussed in section 1.2.4, every traversal protocol performs a broadcast, it follows that, under **RI**, the execution of any traversal protocol constructs a spanning-tree.

SPT Construction by Global Protocols

Actually, we can make a much stronger statement. Call a problem *global* if every entity must participate in its solution; participation implies the execution of a communication activity: transmission of a message and/or arrival of a message (even if it triggers only the Null action, i.e., no action is taken). Both broadcast and traversal are global problems. Now, every single-initiator protocol that solves a *global problem* **P** solves also **Bcast**; thus, from Equation 7, it follows that, under **RI**,

the execution of any *solution to a global problem* **P** *constructs a spanning-tree.*

1.4.4 Considerations on the Constructed Tree

We have seen how, with few more messages than those required by *flooding*, and the same messages as a *df-traversal* we can actually construct a spanning tree.

As discussed previously, once such a tree is constructed, we can from now on perform broadcast and traversal using only $O(n)$ messages (which is optimal) instead of $O(m)$ (which could be as bad as $O(n^2)$).

IMPORTANT. Different techniques construct different spanning-trees. It is even possible that the same protocol constructs different spanning trees when executed at different times.

This is for example the case of *Shout*: since communication delays are unpredictable, subsequent executions of this algorithm on the same graph may result in different spanning trees. In fact,

every possible spanning tree of G *could be constructed by Shout.*

(See also Exercise 6.23.) Prior to its execution, it is impossible to predict which spanning tree will be constructed; the only guarantee is that *Shout* will construct *one*.

This has implications for the time costs of the strategy *Use-a-Tree* of broadcasting on the spanning tree T instead of the entire graph G. In fact, the broadcast time will be $d(T)$ instead of $d(G)$; but $d(T)$ could be much greater than $d(G)$.

For example, if G is the complete graph, the df-tree constructed by any depth-first traversal will have $d(T) = n - 1$; but $d(G) = 1$!

In general, the trees constructed by depth-fist traversal have usually terrible diameters. The ones generated by *Shout* usually perform better, but there is no guarantee on the diameter of the resulting tree.

This fact poses the problem of constructing spanning-trees which have a good diameter; that is, to find a a spanning tree T' of G such that $d(T')$ is not much more than $d(G)$. For obvious reasons, such a tree is traditionally called a *broadcast tree*.

To construct a broadcast tree we must first understand the relationship between *radius* and *diameter*. The *eccentricity* (or radius) of a node x in G is the longest of its *distances* to the other nodes:

$$r_G(x) = Max\{d_G(x, y) : y \in_V\}.$$

A node c with minimum radius (or eccentricity) is called a *center*; i.e., $\forall x \in V,\ r_G(c) \leq r_G(x)$. There might be more than one center; they all however have the same eccentricity, denoted by $r(G)$ and called the *radius of* G:

$$r(G) = Min\{r_G(x) : x \in V\}.$$

There is a strong relationship between the radius and the diameter of a graph; in fact, in every graph G,

$$r(G) \leq d(G) \leq 2r(G) \tag{8}$$

The other ingredient we need is a *breadth-first spanning tree* (*bf-tree*). A breadth-first spanning tree of G rooted in a node u, denoted by $BFT(u, G)$, has the following property: the distance between a node v and the root in the tree is the same as their distance in the original graph G.

The strategy to construct a broadcast tree with diameter $d(T') \leq 2d(G)$ is then simple to state:

STRATEGY *Broadcast-Tree Construction*
(1) determine a *center* c of G;
(2) construct a breadth-first spanning tree $BFT(c, G)$ rooted in c.

This strategy will construct the desired broadcast tree (Exercise 6.29):

Theorem 1.13 *$BFT(c, G)$ is a broadcast tree of G.*

To be implemented, this strategy requires that we solve two problems: Center Finding, and Breadth-First Spanning-Tree Construction. These problems, as we will see, are *not* simple to solve efficiently; we will examine them in later chapters.

1.4.5 Application: Better Traversal

In Section 1.3, we have discussed the general strategy *Use-a-Tree* for problem solving. Now that we know how to construct a spanning-tree (using a single initiator), let us apply the strategy to a known problem.

Consider again the traversal problem. Using the *Use-a-Tree* strategy, we can produce an efficient traversal protocol that is much simpler than all the algorithms we have considered before:

Protocol SmartTraversal.
1. Construct, using *Shout+*, a spanning-tree T rooted in the initiator.
2. Perform a traversal of T, using *DF_Traversal*.

The number of messages of *SmartTraversal* is easy to compute: *Shout+* uses $2m$ messages (Theorem 1.10), while *DF_Traversal* on a tree uses exactly $2(n-1)$ messages (equation 6). In other words,

$$\mathbf{M}[SmartTraversal] = 2(m + n - 1). \tag{9}$$

The problem with *DF_Traversal* was its time complexity: it was to reduce time that we developed more complex protocols. How about the time costs of this simple new protocol ? The ideal time of *Shout+* is exactly $d + 1$. The ideal time of *DF_Traversal* in a tree is $2(n-1)$. Hence the total is

$$\mathbf{T}[SmartTraversal] \leq 2n + d - 1. \tag{10}$$

In other words, *SmartTraversal* not only is simple but also has optimal time and message complexity !

2 MULTIPLE-INITIATORS COMPUTATIONS

In the previous section we have assumed restriction UI; that is, there is a single entity that starts the computation. Indeed, there are some problems (e.g., broadcast and traversal) that satisfy this restriction in their definition. However, for the majority of problems (including spanning-tree construction), this assumption is not "natural", is not part of their definition. For these problems, since the entities are completely autonomous and are initially unaware of the status of the others, a computation can be started by any number of them, independently of each other. In fact, an initiator entity has in general no idea of whether there are other initiators, and if so how many and where they are located.

In this section we will continue our study of some of the most basic primitive distributed computations, and examine those that do not have any *a priori* restriction on the number of the initiators. We discuss *wake up*, and continue our examination of *spanning-tree construction* in its natural setting (i.e., with multiple initiators).

2.1 Wake-Up

2.1.1 Generic Wake-Up

Very often, in a distributed environment, we are faced with the following situation: a task must be performed in which all entities must be involved; however only some of them are independently active (because of a spontaneous event, or have finished a previous computation) and ready to compute; the others are inactive, not even aware of the computation that must take place. In these situations, to perform the task we must ensure that all entities become active. Clearly, this preliminary step can only be started by the entities which are active already; however, they do *not* know which other entities (if any) are already active.

This problem is called *Wake up* (**WakeUp**): an active entity is usually called *awake*, a (still) inactive one is called *asleep*; the task is to wake all entities up.

It is not difficult to see the relationship between broadcasting and wake-up: *broadcast* is a *wake up* with only one initially awake entity; conversely, *wake up* is a broadcast with possibly many initiators (i.e., more than one entity initially has the information). In other words, broadcast is just a special case of the wake up problem.

Interestingly, but not surprisingly, the flooding strategy used for broadcasting actually solves the more general **WakeUp** problem. The modified protocol, called *WFlood* is described in Figure 12. Initially all entities are *asleep*; any *asleep* entity can become spontaneously *awake*

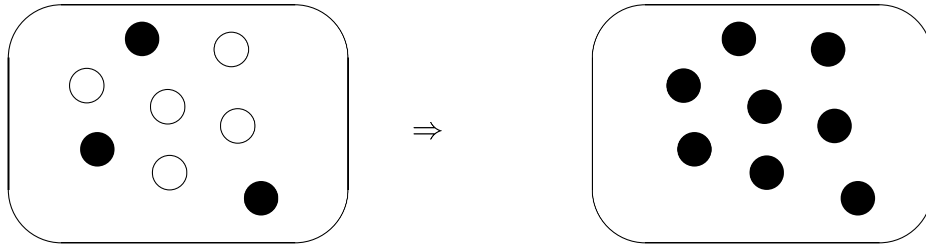

Figure 11: Wake Up Process.

PROTOCOL WFlood .

- Status Values: $\mathcal{S} = \{$ASLEEP,AWAKE$\}$;
  $\mathcal{S}_{INIT} = \{$ASLEEP$\}$;
  $\mathcal{S}_{TERM} = \{$AWAKE$\}$.
- Restrictions: **R**.

```
ASLEEP
        Spontaneously
        begin
              send(W) to N(x);
              become AWAKE;
        end

        Receiving(W)
        begin
              send(W) to N(x) - {sender};
              become AWAKE;
        end
```

Figure 12: Wake-Up by Flooding

and start the protocol.

It is not difficult to verify that the protocol correctly terminates under the standard restrictions (Exercise 6.7).

Let us concentrate on the cost of protocol *WFlood*. The number of messages is at least that of broadcast; actually, it is not much more (see Exercise 6.6):

$$2m \geq \ \mathbf{M}[WFlood] \ \geq 2m - n + 1 \tag{11}$$

Since broadcast is a special case of wake-up, no much improvement is possible (except perhaps in the size of the constant):

$$\mathcal{M}(\mathbf{WakeUp/R}) \ \geq \ \mathcal{M}(\mathbf{Bcast/RI+}) \ = \Omega(m)$$

The ideal time will in general be smaller than the one for broadcast:

$$\mathcal{T}(\textbf{Bcast/RI+}) \;\geq\; \mathcal{T}(\textbf{WakeUp/R})$$

however, in the case of a single initiator, the two cases coincide. Since upper and lower bounds coincide in order of magnitude, we can conclude that protocol *WFlood* is both *message-* and worst-case *time- optimal.*

The complexity of **WakeUp** is summarized by the the following two properties,

Property 2.1 *The message complexity of Wake-up under* **R** *is* $\Theta(m)$

Property 2.2 *The* worst case *ideal time complexity of Wake-up under* **R** *is* $\Theta(d)$

2.1.2 WakeUp in Special Networks

Trees

The cost of using protocol *WFlood* for wakeup will depend on the number of initiators. In fact, if there is only one initiator, then this is just a broadcast and costs only $n-1$ messages. On the other hand, if every entity start independently, there will be a total of $2(n-1)$ messages. Let $k_\star$ denote the number of initiators; note that this number is *not* a system parameter like n or m, it is however bounded by a system parameter: $k_\star \leq n$. Then the total number of messages when executing *WFlood* in a tree will be exactly

$$\mathbf{M}[WFlood/Tree] = n + k_\star - 2. \tag{12}$$

Labelled Hypercubes

In Section 1.1, by exploiting the properties of the hypercube and of the dimensional labelling, we have been able to construct a broadcast protocol which uses only $O(n)$ messages, instead of the $\Omega(n \log n)$ messages required by any generic protocol.

Let us see if we can achieve a similar result also for the wakeup. In other words, can we exploit the properties of a labelled hypercube to do better than generic protocols ?

The answer is unfortunately: *NO !*

Lemma 2.1 $\mathcal{M}(\textbf{WakeUp/R}\,; H) = \Omega(n \log n)$

As a consequence, we might as well employ the generic protocol *WFlood*, which uses $O(n \log n)$ messages. Summarizing,

Property 2.3 *The message complexity of wake-up under* **R** *in a k-dimensional hypercube with a dimensional labeling is* $\Theta(n \log n)$

Complete Graphs
Let us focus on wakeup in a complete graph. The use of the generic protocol *WFlood* will require $O(n^2)$ messages. We can obviously use the simplified broadcast protocol *KBcast* we developed for complete graphs. The number of messages transmitted will be $k_\star(n-1)$ where $k_\star$ denotes the number of initiators. In the worst case (when every entity is independently awake and they all simultaneously start the protocol) $O(n^2)$ messages still will be transmitted.

Let us see if, by exploiting the properties of complete graphs, we we have been able to construct a wake-up protocol that uses only $O(n)$ messages, instead of the $O(n^2)$ we have achieved so far. (After all, we have been able to do it in the case of the broadcast problem.)

Surprisingly, also in this case, the answer is *NO !*

Lemma 2.2 $\mathcal{M}$*(***WakeUp/R** *; K)* $= \Omega(n^2)$

This implies that the use of *WFlood* for wake up is a *message optimal* solution. In other words,

Property 2.4 *The message complexity of wake-up under R in a complete network is* $\Theta(n^2)$

To reduce the number of messages, a more restricted environment is required; that is, we need to make additional assumptions.

For example, if we add the restriction that the entities have unique names (restriction Initial Distinct Values (ID)), then there are protocols capable of performing wakeup with $O(n \log n)$ messages in a complete graph; they are not simple and actually solve a much more complex problem, *Election*, which we will discuss at great length in the next chapter. Strangely, nothing than that can be accomplished. In fact, let $IR + K = \mathbf{R} \cup K$; then the worst-case *message complexity* of wake-up in a complete graph under the standard restrictions **R** plus ID is:

Property 2.5 $\mathcal{M}(\textbf{Elect/R}; ID; K) \geq .5n \log n$

To see why this is true, we will construct a "bad" but possible case, which any protocol can encounter, and show that, in such a case, $O(nlogn)$ messages will be exchanged. The lower bound will hold even if there is Message Ordering. For simplicity of discussion and calculation, we will assume that n is a power of 2; the results hold also if this is not the case.

To construct the "bad" case for an (arbitrary) solution protocol A, we will consider a *game* between the entities on one side, and an *adversary* on the other: the entities obey the rules of the protocol; the adversary will try to make the worst possible scenario occur, so to force the use of as many messages as possible.

The powers of the adversary are four:
(1) it decides the initial values of the entities (they must be distinct);
(2) it decides which entities spontaneously start the execution of A, and when;

(3) it decides when a transmitted message arrives (it must be within finite time);
(4) more importantly, it decides the matching between links and labels: let $e_1, e_2, \ldots, e_k$ be the links incident on x, and let $l_1, l_2, \ldots, l_k$ be the port labels to be used by x for those links; during the execution, when x performs a "**send to** l" command, and l has not been assigned yet, the adversary will choose which of the unused links (i.e., through which no messages has been sent nor received) the label l will be assigned to.

Note. Sending a message to more than one port, will be treated as sending the message to each of those ports one at the time (in an arbitrary order).

Whatever the adversary decides, it can happen in a real execution. Let us see how bad a case can the adversary create for A.

Two sets of entities will be said to be *connected* at a time t if at least a message has been transmitted from an entities of one set to an entity of the other.

Adversary's Strategy.

(1) Initially, the adversary will wake-up only one entity s, which we will call the *seed*, and which will start the execution of the protocol. When s decides to send a message to port number l, the adversary will wake-up another entity y and assign label l to the edge from s to y. It will then delay the transmission on that link until also y decides to send a message to some port number l'; the adversary will then assign label l' to the link from y to s, and let the two messages arrive to their destination simultaneously. In this way, each message will reach an awake node, and the two entities are connected.

From now on, the adversary will act in a similar way, always ensure that messages are sent to already awake nodes, and that the set of awake nodes is connected.

(2) Consider an entity x executing a **send** operation to an unassigned label a.

1. If x has an unused link (i.e., a link on which no messages have been sent so far) connecting it to an awake node, the adversary will assign a to that link. In other words, the adversary will try to make the awake entities to send messages always to other awake entities.

2. If all links between x and the awake nodes have been used, then the adversary will create another set of awake nodes and connect the two sets.

    (a) Let $x_0, \ldots, x_{k-1}$ be the currently awake nodes, ordered according to their wake-up time (thus, $x_0 = s$ is the seed, and $x_1 = y$). The adversary will: choose k inactive nodes $z_0, \ldots, z_{k-1}$; establish a logical correspondence between x_j and z_j; assign to the new entities initial values so that the order among them is the same as the one among the values of the corresponding entities; wake-up these entities and force them to have the "same" execution (same scheduling and same delays) as the corresponding ones already did. (So, z_0 will be woken-up first, its first message will be sent to z_1, which will be woken-up next and will send a message to z_0, etc.)

    (b) The adversary will then assign label a to the link connecting x to its corresponding entity z in the new set; the message will be held in transit until z (like x did) will

need to transmit a message on an unused link (say, with label b) but all the edges connecting it to its set of awake entities have already been used.

(c) When this happens, the adversary will assign the label b to the link from z to x, and make the two messages between x and z arrive and be processed.

Let us summarize the strategy of the adversary: the adversary tries to force the protocol to send messages only to already awake entities, and awakens new entities only when it cannot do otherwise; the newly awake entities are equal in number to the already awake entities; and they are forced by the adversary to have the same execution between them as those others entities before any communication takes place between the two sets. When this happens, we will say that the adversary has started a new *stage*.

Let us now examine the situations created by the adversary with this strategy, and analyze the cost of the protocol in the corresponding executions.

Let $Active(i)$ denote the awake entities in stage i, and $New(i) = Active(i) - Active(i-1)$ the entities that the adversary woke-up in this stage; initially, $Active(0)$ is just the seed. The newly awake entities are equal in number to the already awake entities; i.e., $|New(i)| = |Active(i-1)|$).

Let $\mu(i-1)$ denote the *total* number of messages which have been exchanged before the activation of the new entities. The adversary forces the new entities to have the same execution as the entities in $Active(i-1)$, thus exchanging $\mu(i-1)$ of messages, before allowing the two sets to become connected. Thus, the total number of messages until the communication between the two sets takes place is $2\mu(i-1)$.

Once the communication takes place, how many messages (including those two) are transmitted before the next stage ?

The exact answer will depend on the protocol A; but regardless of which protocol we are using, the adversary will not start a new stage $i+1$ unless it is forced to; this will happen only if an entity x issues a "**send to** l" command (where l is an unassigned label) *and* all the links connecting x to the other awake entities have already been used. This means that x must have either sent to or received from all the entities in $Active(i) = Active(i-1) \cup New(i)$. Assume that $x \in Active(i-1)$; then, of all these messages, the ones between x and $New(i)$ have only occurred in stage i (since those entities where not active before); this means that at least $|New(i)| = |Active(i-1)|$ additional messages are sent before stage $i+1$. If instead $x \in New(i)$, these messages have all been transmitted in this stage (since x was not awake before); in other words, even in this case, $|New(i)| = |Active(i-1)|$ additional messages are sent before stage $i+1$.

Summarizing, the total cost $\mu(i-1)$ before stage i is thus doubled and *at least* an additional $|Active(i-1)|$ messages are sent before stage $i+1$. In other words:

$$\mu(i) \geq 2\ \mu(i-1) + |Active(i-1)|$$

Since the awake entities double in each stage, and initially only the seed is active, then $|Active(i)| = 2^i$. Hence, observing that $\mu(0) = 0$,

$$\mu(i) \geq 2\ \mu(i-1) + 2^{i-1} \geq i\ 2^{i-1}$$

The total number of stages is exactly $\log n$ since the awake processes double every stage. Hence, with this strategy, the adversary can force any protocol to transmit *at least* $\mu(\log n)$ messages. Since

$$\mu(\log n) \geq .5\ n \log n$$

it follows that *any* wake-up protocol will transmit $\Omega(n \log n)$ messages in the worst case even if the entities have distinct ids !

More efficient wake up protocols can be derived if instead we have in our system a "good" labeling of the links. We will return on this topic when we will discuss *Sense of Direction*.

2.2 Spanning-Tree Construction

We have started examining the *spanning-tree construction* problem in section 1.4 assuming that there is a unique initiator. This is unfortunately a very strong (and "unnatural") assumption to make, as well as difficult and expensive to guarantee.

What happens to the single-initiator protocols *Shout* and *df-SPT* if there is more than one initiator?

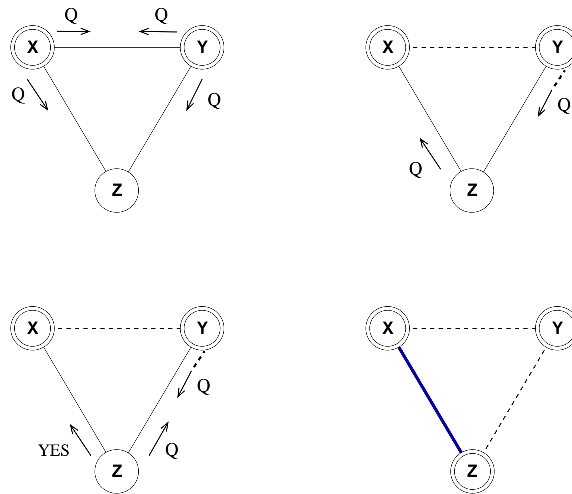

Figure 13: With multiple initiators, *Shout* creates a forest.

Let us examine first protocol *Shout*. Consider the very simple case (depicted in Figure 13) of three entities, x, y, z, connected to each other. Let both x and y be initiators and start the protocol, and let the Q message from x to z arrive there before the one sent by y.

In this case, neither the link (x, y) nor the link (y, z) will be included in the tree; hence, the algorithm creates not a spanning tree but a *spanning forest*, which is not connected.

Consider now protocol *df-SPT*, discussed in Section 1.4.3. Let us examine its execution in the simple network depicted in Figure 14 composed of a chain of four nodes x, y, z and w. Let y and z be both initiators, and start the traversal by sending the T message to x and w, respectively.

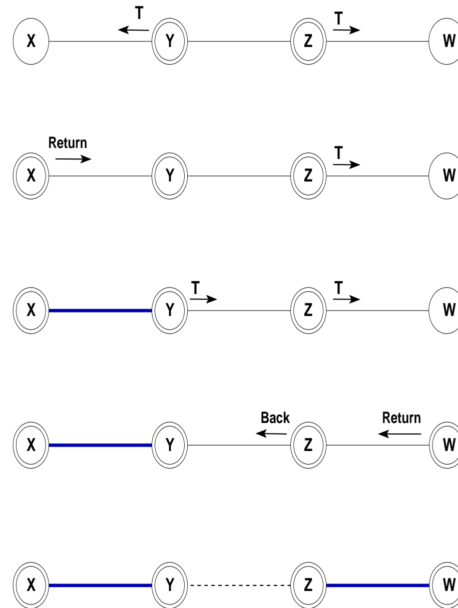

Figure 14: With multiple initiators, *df-SPT* creates a forest.

Also in this case, the algorithm will create a disconnected *spanning forest* of the graph. It is easy to verify that the same situation will occur also with the optimized versions (*DF+* and *DF**) of the protocol (Exercise 6.30).

The failure of these algorithms is not surprising, since they were developed specifically for the restricted environment of a Unique Initiator.

Removing the restriction brings out the true nature of the problem which, as we will now see, has a formidable obstacle.

2.2.1 Impossibility Result

Our goal is to design a spanning-tree protocol which works solely under the standard assumptions, and thus is independent of the number of initiators. Unfortunately, any design effort to this end is destined to *fail*! In fact

Theorem 2.1 *The* **SPT** *problem is deterministically* unsolvable *under* **R**.

Deterministically unsolvable means that there is *no deterministic protocol which always correctly terminates within finite time.*

Proof.

To see why this is the case, consider the simple system composed of three entities, x, y, and z connected by links labeled as shown in Figure 15. Let the three entities have identical initial values (the symbols x, y, z are used only for description purposes). If a solution protocol A exists, it must work under any conditions of message delays (as long as they are finite) and regardless of the number of initiators. Consider a *synchronous schedule* (i.e., an execution where communication delays are unitary) and let *all* three entities start the execution of A simultaneously. Since they are identical (same initial status and values, same port labels), they will execute the same rule, obtain the same results (thus, continuing to have the same local values), compose and send (if any) the same messages; enter the same (possibly new) status. In other words, they will remain identical. In the next time unit, all sent messages (if any) will arrive and be processed. If one entity receives a message, the others will receive the same message at the same time, perform the same local computation, compose and send (if any) the same messages; enter the same (possibly new) status. And so on. In other words, the entities will continue to be identical.

If A is a solution protocol, it must terminate within finite time. A spanning-tree of our simple system is obtained by removing one of the three links, let us say (x, y). In this case, Tree-neigbours will be the port label 2 for entity x and the port label 1 for entity y; instead, z has in Tree-neighbours both port numbers. In other words, when they all terminate, they have *distinct* values for their local variable Tree-neighbours. But this is impossible, since we just said that the entities will always be identical.

Thus, no such a solution algorithm A exists. ■

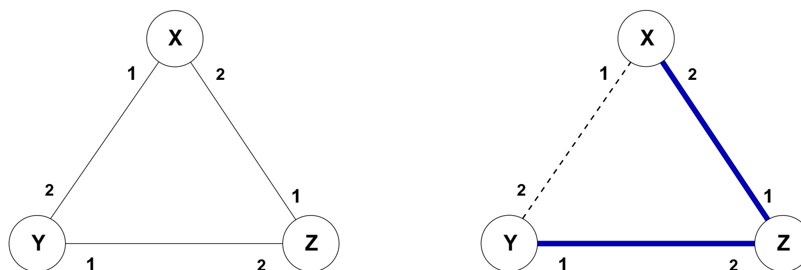

Figure 15: Proof of Theorem 2.1.

A consequence of this very negative result is that, to construct a spanning-tree without constraints on the number of initiators, we need to impose additional restrictions. To determine

the "minimal" restrictions which, added to **R**, will enable us to solve **SPT** is an interesting research problem still open. The restriction that is commonly used is a very powerful one, Initial Distinct Values, and we will discuss it next.

2.2.2 SPT with Initial Distinct Values

The impossibility result we just witnessed implies that, to solve the SPT problem we need an additional restriction. The one commonly used is *Initial Distinct Values* (ID): *each entity has a distinct initial value.* Distinct initial values are sometimes called *identifiers* or *ids* or *global names.*

We will now examine some ways in which **SPT** can be solved under $\mathbf{RD} = \mathbf{R} \cup \{\text{ID}\}$.

Multiple spanning trees
As in most software design situations, once we have a solution for a problem and are faced with a more general one, an approach is to try to find ways to re-use and re-apply the already existing solution. The solutions we have already are unique-initiator ones and, as we know, they fail in presence of multiple initiators. Let us see how can we mend their shortcomings using distinct values.

Consider the execution of *Shout* in the example of Figure 13. In this case, the reason why the protocol fails is because the entities do not realize that there are two different requests (e.g., when x receives Q from y) for spanning tree construction.

But we can now use the entities' ids to *distinguish* between requests originating from different initiators.

The simplest and most immediate application of this approach is to have each initiator construct "its own" spanning tree with a single-initiator protocol, and to use the ids of the initiators to distinguish among different constructions. So, instead of cooperating to construct a single spanning tree, we will have several spanning trees concurrently and independently built.

This implies that all the protocol messages (e.g., Q and Yes in *Shout+*) must contain also the *id* of the initiator. It also requires additional variables and bookkeeping; for example, at each entity, there will be several instances of the variable Tree-Neighbours, one for each spanning-tree being constructed (i.e., one for each initiator). Furthermore, each entity will be in possibly different status values for each of these independent spt-constructions. Recall that the number $k_\star$ of initiators is not known a priori, and can change at every execution.

The message cost of this approach depends solely on the number of initiators and on the type of unique-initiator protocol used. But it is in any case very expensive. In fact, if we employ the most efficient spt-construction protocol we know, *Shout+*, we will use $2mk_\star$ messages which could be as bad as $O(n^3)$.

Selective construction
The large message cost derives from the fact that we construct not one but $k_\star$ spanning trees. Since our goal is just to construct one, there is clearly a needless amount of communication and computation being performed.

A better approach consists of letting every initiator start the construction of its own uniquely identified spanning-tree (as before), but then suppressing some of these constructions, allowing only one to complete. In this approach, an entity faced with two different spt-constructions, will select and act on only one, "killing" the other; the entity continues this selection process as long as it receives conflicting requests.

The criterion an entity uses to decide which spt-construction to follow and which one to terminate must be chosen very carefully. In fact, the danger is to "kill" all constructions !

The criterion commonly used is based on *min-id*: since each spt-construction has a unique id (that of its initiator), when faced with different spt-constructions an entity will choose the one with the *smallest* id, and terminate all the others. (An alternative criterion would be the one based on *max-id*.)

The solution obtained with this approach has some very clear advantages over the previous solution. First of all, each entity is at any time involved only in one spt-construction; this fact greatly simplifies the internal organization of the protocol (i.e., the set of rules), as well as the local storage and bookkeeping of each entity. Secondly, upon termination, all entities have a single shared spanning-tree for subsequent uses.

However, there is still competitive concurrency: an entity involved in one spt-construction might receive messages from another construction; in our approach, it will make a choice between the two constructions. If the entity chooses the new one, it will give up all the knowledge (variables, etc) acquired so far, and start from scratch. The message cost of this approach depends again on the number of initiators and on the unique-initiator protocol used.

Consider a protocol developed using this approach using *Shout+* as the basic tool.

Informally, an entity u, at any time, participates in the construction of just one spanning-tree rooted in some initiator, x. It will ignore all messages referring to the construction of other spanning trees where the initiators have larger ids than x. If instead u receives a message referring to the construction of a spanning-tree rooted in an initiator y with an id smaller than x's, then u will stop working for x and start working for y. As we will see, this techniques will construct a spanning-tree rooted in the initiator with the smallest initial value.

IMPORTANT. It is possible that an entity has already terminated its part of the construction of a spanning tree when it receives a message from another initiator (possibly, with a smaller id).

In other words, when an entity has terminated a construction, it does not know whether it might have to restart again. Thus, it is *necessary* to include in the protocol a mechanism that ensures an effective local termination for each entity.

This can be achieved by ensuring that we use, as a building block, a unique-initiator spt-protocol in which the initiator will know when the spanning tree has been completely constructed (see Exercise 6.24). In this way, when the spanning tree rooted in the initiator s with the smallest initial value has been constructed, s will become aware of this fact (as well as that all other constructions, if any, have been "killed"). It can then notify all other entities

PROTOCOL MultiShout

- Status: $\mathcal{S}$ = {IDLE, ACTIVE, DONE}; $\mathcal{S}_{INIT}$ = {IDLE}; $\mathcal{S}_{TERM}$ = {DONE}.
- Restrictions: **R** ;ID.

```
IDLE
    Spontaneously
    begin
        root:= true;
        root_id:=v(x);
        Tree_neighbours:=∅;
        send(Q,root_id) to N(x);
        counter:=0;
        check_counter:=0;
        become ACTIVE;
    end

    Receiving(Q,id)
    begin
        CONSTRUCT;
    end

ACTIVE
    Receiving(Q,id)
    begin
        if root_id = id then
          counter:=counter+1;
          if counter=|N(x)| then done:= true; CHECK; endif
        else
          if root_id > id then CONSTRUCT;
        endif
    end

    Receiving(Yes, id)
    begin
        if root_id = id then
          Tree-neighbours:=Tree-neighbours ∪{sender};
          counter:=counter+1;
          if counter=|N(x)| then done:= true; CHECK; endif
        endif
    end

    Receiving(Check, id)
    begin
        if root_id = id then
          check_counter:=check_counter+1;
          if (done ∧ check_counter=|Children|) then TERM; endif
        endif
    end

    Receiving(Terminate)
    begin
        send(Terminate) to Children;
        become DONE;
    end
```

Figure 16: Protocol MultiShout

so they can enter a terminal status. The notification is just a broadcast; it is appropriate to perform it on the newly constructed spanning-tree (so we start taking advantage of its existence).

Protocol *MultiShout*, depicted in Figures 16 and 17, uses *Shout+* appropriately modified so to ensure that the root of a constructed tree becomes aware of termination, and includes a final broadcast (on the spanning tree) to notify all entities that the task has been indeed completed. We denote by $v(x)$ the id of x; initially all entities are *idle* and any can spontaneously start the algorithm.

Theorem 2.2 *Protocol* MultiShout *constructs a spanning tree rooted in the initiator with the smallest initial value.*

Proof. Let s be the initiator with the smallest initial value. Focus on an initiator $x \neq s$; its initial execution of the protocol will start the construction a spanning-tree T_x rooted in x. We will first show that the construction of T_x will *not* be completed. To see this, observe that T_x must include every node, including s; but when s receives a message relating to the construction of somebody's else tree (such as T_x), it will ignore it, killing the construction of that tree. Let us now show that T_s will instead be constructed. Since the id of s is smaller than all other ids, *no* entity will ignore the messages related to the construction of T_s started by s; thus, the construction will be completed. ■

Let us now consider the message costs of protocol *MultiShout*. It is clearly more efficient than protocols obtained with the previous approach. However, in the worst case, it is not much better in order of magnitude. In fact, it can be as bad as $O(n^3)$.

Consider for example the graph, shown in Fig. 18, where $n-k$ of the nodes are fully connected among themselves (the subgraph K_{n-k}), and each of the other k (nodes $x_1, x_2, \ldots, x_k$) is connected only to a node in K_{n-k}. Suppose that these k "external" nodes are the initiators and that $v(x_1) > v(x_2) > \ldots > v(x_k)$,

Consider now an execution where the Q messages from the external entities arrive to K_{n-k} in order, according to the indices (i.e., the one from x_1 arrives first).

When the Q message from x_1 arrives to K_{n-k} it will trigger the spt-construction there. Notice that the *Shout+* component of our protocol with a unique initiator will use $O((n-k)^2)$ messages inside the subgraph K_{n-k}. Assume that the entire computation inside K_{n-k} triggered by x_1 is practically completed (costing $O((n-k)^2)$ messages) by the time the Q message from x_2 arrives to K_{n-k}. Since $v(x_1) > v(x_2)$, all the work done in K_{n-k} has been wasted and every entity there must start the construction of the spanning tree rooted in x_2.

In the same way, assume that the time delays are such that the Q message from x_i arrives to K_{n-k} only when the computation inside K_{n-k} triggered by x_{i-1} is practically completed (costing $O((n-k)^2)$ messages).

Then, in this case (which is possible), work costing $O((n-k)^2)$ messages will be repeated k times, for a total of $O(k(n-k)^2)$ messages. If k is a linear fraction of n (e.g., $k = n/2$), then the cost will be $O(n^3)$.

```
Procedure CONSTRUCT
begin
      root:= false;
      root_id:= id;
      Tree_neighbours:={sender};
      parent:= sender;
      send(Yes,root_id) to {sender};
      counter:=1;
      check_counter:=0;
      if counter=|N(x)| then
        done:= true;
        CHECK;
      else
        send(Q,root-id) to N(x) - {sender};
      endif
      become ACTIVE;
end


Procedure CHECK
begin
      Children:= Tree_neighbours-{parent};
      if Children = ∅ then
       send(Check,root_id) to parent;
      endif
end


Procedure TERM
begin
      if root then
       send(Terminate) to Tree-neighbours;
       become DONE;
      else
       send(Check,root-id) to parent;
      endif
end
```

Figure 17: Routines of MultiShout

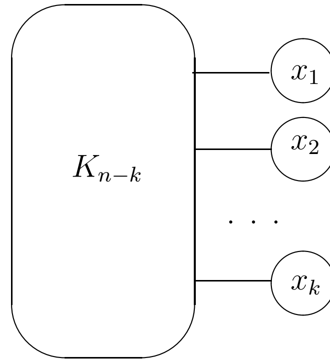

Figure 18: The execution of MultiShout can cost $O(k(n-k)^2)$ messages.

The fact that this solution is not very efficient does not imply that the approach of *selective construction* it uses is not effective. On the contrary, it can be made efficient at the expenses of simplicity. We will examine it in great details later in the book when studying the *leader election* problem.

3 SATURATION AND COMPUTATIONS IN TREES

In this section, we consider computations in *tree* networks under the standard restrictions **R**, plus clearly the common knowledge (T) that the network is tree.

Note that the knowledge of being in a tree implies that each entity can determine whether it is a *leaf* (i.e., it has only one neighbour) or an *internal node* (i.e., it has more than one neighbour).

We have already seen how to solve the Broadcast, the Wake-Up and the Traversal problems in a tree network. The first two are optimally solved by protocol *Flooding*, the latter by protocol *DF_Traversal*. These techniques constitute the first set of algorithmic tools for computing in trees with multiple initiators. We will now introduce another very basic and useful technique, *saturation*, and show how it can be employed to efficiently solve many different problems in trees regardless of the number of initiators and of their location.

Before doing so, we need to introduce some basic concepts and terminology about trees. In a tree T, the removal of a link (x, y) will disconnect T into two trees, one containing x (but not y), the other containing y (but not x); we shall denote them by $T[x-y]$ and $T[y-x]$, respectively. Let $d[x, y] = Max\{d(x, z) : z \in T[y-x]\}$ be the longest distance between x and the nodes in $T[y-x]$. Recall that the longest distance between any two nodes is called *diameter*, and it is denoted by d. If $d[x, y] = d$, the path between x and y is said to be *diametral*.

3.1 Saturation: A Basic Technique

The technique, which we shall call *Full Saturation*, is very simple and can be autonomously and independently started by *any* number of initiators.

It is composed of three stages:
(1) the *activation* stage, started by the initiators, in which all nodes are activated;
(2) the *saturation* stage, started by the leaf nodes, in which a unique couple of neighbouring nodes is selected; and
(3) the *resolution* stage, started by the selected pair.

The *activation* stage is just a wakeup: each initiator sends an activation (i.e., wake-up) message to all its neighbours and becomes *active*; any non-initiator, upon receiving the activation message from a neighbour, sends it to all its other neighbours, and becomes *active*; *active* nodes ignore all received activation messages. Within finite time, all nodes become *active*, including the leaves. The leaves will start the second stage.

Each active leaf starts the *saturation* stage by sending a message (call it M) to its only neighbour, referred now as its "parent", and becomes *processing.* (note: M messages will start arriving within finite time to the internal nodes). An internal node waits until it has received an M message from all its neighbours *but one*, sends a M message to that neighbour that will now be considered its "parent", and becomes *processing.* If a *processing* node receives a message from its parent, it becomes *saturated.*

The *resolution* stage is started by the *saturated* nodes; the nature of this stage depends on the application. Commonly, this stage is used as a *notification* for all entities (e.g., to achieve local termination).

Since the nature of the final stage will depend on the application, we will only describe the set of rules implementing the first two stages of Full Saturation.

IMPORTANT. A "truncated" protocol like this will be called a **plug-in**. In its execution, not all entities will enter a terminal status. To transform it into a full protocol, some other action (e.g., the resolution stage) must be performed so that eventually all entities enter a terminal status.

It is assumed that, initially all entities are in the same status *available.*

Let us now discuss some properties of this basic technique.

Lemma 3.1 *Exactly two* processing *nodes will become* saturated*; furthermore, these two nodes are neighbours and are each other's parent.*

Proof. From the algorithm, it follows that an entity sends a message M only to its parent, and becomes *saturated* only upon receiving an M message from its parent. Choose an arbitrary node x, and traverse the "up" edge of x (i.e. the edge along which the M message was sent from x to its parent). By moving along "up" edges, we must meet a *saturated* node s_1 since there are no cycles in the graph. This node has become *saturated* when receiving an M message from its parent s_2. Since s_2 has sent an M message to s_1, this implies that s_2 must have been *processing* and must have considered s_1 its parent; thus, when the M

PLUG-IN Full-Saturation .

- Status: $\mathcal{S}$ = {AVAILABLE, ACTIVE, PROCESSING, SATURATED}; $\mathcal{S}_{INIT}$ = {AVAILABLE};
- Restrictions: **R** ∪ T.

```
AVAILABLE
          Spontaneously
          begin
                send(Activate) to N(x);
                Initialize;
                Neighbours:= N(x);
                if|Neighbours|=1 then
                  Prepare_Message;
                  parent ⇐ Neighbours;
                  send(M) to parent;
                  become PROCESSING;
                else become ACTIVE;
                endif
          end

          Receiving(Activate)
          begin
                send(Activate) to N(x) − {sender};
                Initialize;
                Neighbours:= N(x);
                if|Neighbours|=1 then
                  Prepare_Message;
                  parent ⇐ Neighbours;
                  send(M) to parent;
                  become PROCESSING;
                else become ACTIVE;
                endif
          end

ACTIVE
          Receiving(M)
          begin
                Process_Message;
                Neighbours:= Neighbours−{sender};
                if|Neighbours|=1 then
                  Prepare_Message;
                  parent ⇐ Neighbours;
                  send(M) to parent;
                  become PROCESSING;
                endif
          end

PROCESSING
          Receiving(M)
          begin
                Process_Message;
                Resolve;
          end
```

Figure 19: Full Saturation

```
Procedure Initialize
begin
      nil;
end

Procedure Prepare_Message
begin
      M:=("Saturation");
end

Procedure Process_Message
begin
      nil;
end

Procedure Resolve
begin
      become SATURATED;
      Start Resolution stage;
end
```

Figure 20: Procedures used by Full Saturation

message from s_1 will arrive at s_2, also s_2 will become *saturated*. Thus, there exist at least two nodes which become *saturated*; furthermore these two nodes are each other's parent. Assume that there are more than two *saturated* nodes; then there exist two *saturated* nodes, x and y, such that $d(x, y) \geq 2$. Consider a node z on the path from x to y; z could not send a M message towards both x and y; therefore one of them nodes cannot be *saturated*. Therefore, the lemma holds. ■

IMPORTANT. Which entities will become saturated depends on the communication delays and, it is therefore totally unpredictable. Subsequent executions with the same initiators might generate different results. In fact

any pair of neighbours could become saturated.

The only guarantee is that *a* pair of neighbours will be selected; since a pair of neighbours uniquely identifies an edge, the one connecting them, this result is also called *edge election*.

To determine the number of message exchanges, observe that the activation stage is a wake-up in a tree and hence it will use $n + k_\star - 2$ messages (Equation 12), where $k_\star$ denotes the number of initiators. During the saturation stage, exactly one message is transmitted on each edge, except the edge connecting the two *saturated* nodes on which two M messages are transmitted, for a total of $n - 1 + 1 = n$ messages. Thus,

$$\mathbf{M}[FullSaturation] = 2n + k_\star - 2 \quad (13)$$

Notice that only n of those messages are due to the saturation stage.

To determine the ideal time complexity, let $I \subseteq V$ denote the set of initiator nodes, $L \subseteq V$ denote the set of leaf nodes; $t(x)$ the time delay, from the initiation of the algorithm, until node x becomes *active*. To become *saturated*, node s must have waited until all the leafs have become *active* and the M messages originated from them have reached s; that is, it must have waited $Max\{t(l) + d(l,s) : l \in L\}$. To become *active*, a non-initiator node x must have waited for an (”Activation”) message to reach it, while there is no additional waiting time for an initiator node; thus, $t(x) = Min\{d(x,y) + t(y) : y \in I\}$. Therefore, the total delay, from the initiation of the algorithm, until s becomes *saturated* (and, thus, the ideal execution delay of the algorithm) is

$$\mathbf{T}[FullSaturation] = Max\{Min\{d(l,y) + t(y)\} + d(l,y) : y \in I, l \in L\}. \tag{14}$$

We will now discuss how to apply the saturation technique to solve different problems.

3.2 Minimum Finding

Let us see how the saturation technique can be used to compute the smallest among a set of values distributed among the nodes of the network. Every entity x has an input value $v(x)$, and is initially in the same status; the task is to determine the minimum among those input values. That is, in the end, each entity must know whether or not its value is the smallest, and enter the appropriate status, *minimum* or *large*, respectively.

IMPORTANT. Notice that these values are *not* necessarily distinct ! So, more than one entity can have the minimum value; all of them must become *minimum*.

This problem is called *Minimum Finding* (**MinFind**) and is the simplest among the class of *Distributed Query Processing* problems that we will examine in later chapters: a set of data (e.g., a file) is distributed among the sites of a communication network; *queries* (i.e., external requests for information about the set) can arrive at any time at any site (which becomes an initiator of the processing), triggering computation and communication activities. A stronger version of this problem requires all entities to *know* the minimum value when they enter the final status.

Let us see how to solve this problem in a tree network. If the tree was *rooted*, then this task can be trivially performed. In fact, in a rooted tree not only is there a special node, the root, but also a logical orientation of the links: “up” towards the root and “down” away from the root; this correspond to the “parent” and “children” relationship, respectively. In a rooted tree, to find the minimum, the root would broadcast down the request to compute the minimum value; exploiting the orientation of the links, the entities will then perform a *convergecast* (described in more details in Section 3.7.2): starting from the leaves, the nodes determine the smallest value among the values “down”, and send it “up”. As a result of this process, the minimum value is then determined at the root, which will then broadcast it to all nodes.

Notice that convergecast can be used only in rooted trees. The existence of a root (and the additional information existing in a rooted tree) is however a very strong assumption; in

fact, it is equivalent to assuming the existence of a *leader* (which, as we will see, might not be computable).

Full saturation allows to achieve the same goals *without* a root or any additional information. This is achieved simply by including in the M message the smallest value known to the sender. Namely, in the saturation stage the leaves will send their value with the M message, and each internal node sends the smallest among its own value and all the received ones.

In other words, *MinF-Tree* is just protocol *Full-Saturation* where the procedures Initialize, Prepare_Message, and Process_Message are as shown in Fig. 21, and where the *resolution* stage is just a notification started by the two saturated nodes, of the minimum value they have computed. This is obtained by simply modifying procedure Resolve accordingly and adding the rule for handling the reception of the notification.

The correctness follows from the fact that both saturated nodes know the minimum value (Exercise 6.31).

The number of message transmission for the minimum-finding algorithm *MinF-Tree* will be exactly the same as the one experienced by Full Saturation plus the ones performed during the notification. Since a notification message is sent on every link *except* the one connecting the two *saturated* nodes, there will be exactly $n-2$ such messages. Hence

$$\mathbf{M}[MinF-Tree] = 3n + k_{\star} - 4. \tag{15}$$

The time costs will be the one experienced by Full Saturation plus the ones required by the notification. Let Sat denote the set of the two saturated nodes; then

$$\mathbf{T}[MinF-Tree] = \mathbf{T}[FullSaturation] + Max\{d(s,x) : s \in Sat, x \in V\} \tag{16}$$

3.3 Distributed Function Evaluation

An important class of problems are those of *Distributed Function Evaluation*; that is, where the task is to compute a function whose arguments are distributed among the processors of a distributed memory system (e.g., the sites of a network). An instance of this problem is the the one we just solved: minimum finding. We will now discuss how the saturation technique can be used to evaluate a large class of functions.

3.3.1 Semigroup Operations

Let f be an *associative* and *commutative* function defined over all subsets of the input values. Examples of this type of functions are: minimum, maximum, sum, product, etc, as well as logical predicates. Because of their algebraic properties, these functions are called *semigroup operations*.

IMPORTANT. It is possible that some entities do not have an argument (i.e. initial value) or that the function must only be evaluated on a subset of the arguments. We shall denote the fact that x does not have an argument by $v(x)$ =nil.

```
PROCESSING
          Receiving(Notification)
          begin
               send(Notification) to N(x)−parent;
               if   v(x) =Received_Value then
                    become MINIMUM;
               else
                        become LARGE;
               endif
          end
```

```
Procedure Initialize
begin
     min:=v(x);
end

Procedure Prepare_Message
begin
     M:=("Saturation", min);
end

Procedure Process_Message
begin
     min:= MIN{min, Received_Value};
end

Procedure Resolve
begin
     Notification:= (‘‘Resolution’’, min);
     send(Notification) to N(x)−parent;
     if   v(x) =min then
          become MINIMUM;
     else
              become LARGE;
     endif
end
```

Figure 21: New Rule and Procedures used for Minimum Finding

```
PROCESSING
          Receiving(Notification)
          begin
               result:= received_value;
               send(Notification) to N(x)−parent;
               become DONE;
          end
```

```
Procedure Initialize
begin
     if v(x) ≠ nil then
      result:=f(v(x));
     else
      result:=nil;
end

Procedure Prepare_Message
begin
     M:=("Saturation", result);
end

Procedure Process_Message
begin
     if  received_value≠ nil then
        if     result≠ nil then
           result:= f(result, received_value);
        else
           result:= f(received_value);
        endif
     endif
end

Procedure Resolve
begin
     Notification:= ("Resolution", result);
     send(Notification) to N(x)−parent;
     become DONE;
end
```

Figure 22: New Rule and Procedures used for Function-Tree

The same approach that has led us to solve Minimum Finding can be used to evaluate f.

The protocol *Function-Tree* is just protocol *Full-Saturation* where the procedures Initialize, Prepare_Message, and Process_Message are as shown in Fig. 22, and where the *resolution* stage is just a notification started by the two saturated nodes, of the final result of the function they have computed. This is obtained by simply modifying procedure Resolve accordingly and adding the rule for handling the reception of the notification.

The correctness follows from the fact that both saturated nodes know the result of the function (Exercise 6.32). For particular types of functions, see Exercises 6.33, 6.34, and 6.35.

The time and message costs of the protocol are exactly the same as the one for minimum-finding. Thus, semigroup operations can be performed *optimally* on a tree with any number of initiators and without a root or additional information.

3.3.2 Cardinal Statistics

A useful class of functions are *statistical* ones, such as *average*, *standard deviation*, etc. These functions are not semigroup operation but can nevertheless be optimally solved using the saturation technique.

We will just examine, as an example, the computation of Ave, the average of the (relevant) entities' values. Observe that $Ave \equiv Sum/Size$ where Sum is the the sum of all (relevant) values, and Size is the number of those values. Since Sum is a semigroup operation, we already know how to compute it. Also $Size$ is trivially computed using saturation (Exercises 6.36 and 6.37).

We can collect at the two saturated nodes Sum and $Size$ with a single execution of Saturation: the M message will contain two data fields M=("Saturation", sum,size), which are initialized by a each leaf node and updated by the internal ones. The *resolution* stage is just a notification started by the two saturated nodes, of the average they can have computed.

Similarly, a single execution of Full Saturation with a final notification of the result will allow the entities to compute *cardinal* statistics on the input values.

Notice that *ordinal* statistics (e.g., median) are in general more difficult to resolve. We will discuss them in the chapter on selection and sorting of distributed data.

3.4 Finding Eccentricities

The basic technique has been so far used to solve single-valued problems; that is, problems whose solution requires the identification of a single value). It can also be used to solve multi-valued problems such as the problem of determining the eccentricities of all the nodes.

The *eccentricity* of a node x, denoted by $r(x)$, is the largest distance between x and any other node in the tree: $r(x) = Max\{d(x,y) : y \in V\}$; note that a *center* is a node with smallest eccentricity. (We briefly discussed center and eccentricity already in Section 1.4.4.)

To compute its own eccentricity, a node x needs to determine the maximum distance from

```
PROCESSING
          Receiving(Notification)
          begin
               result:= received_value;
               send(Notification) to N(x)−parent;
               become DONE;
          end
```

```
Procedure Initialize
begin
     sum:=v(x);
     size:=1;
end

Procedure Prepare_Message
begin
     M:=("Saturation", sum,size);
end

Procedure Process_Message
begin
     sum:= sum + Received_sum;
     size:=size + Received_size;
end

Procedure Resolve
begin
     result := sum / size;
     Notification:= (‘‘Resolution’’, result);
     send(Notification) to N(x)−parent;
     become DONE;
end
```

Figure 23: New Rule and Procedures used for computing the Average

all other nodes in the tree. To accomplish this, x needs just to broadcast the request, making itself the root of the tree, and, using convergecast on this rooted tree, collect the maximum distance to itself. This approach would require $2(n-1)$ messages and it is clearly optimal with respect to order of magnitude. If we want *every* entity to compute its eccentricity, this however would lead to a solution which requires $2(n^2-n)$ messages.

We will now show that saturation will yield instead a $O(n)$, and thus optimal, solution.

The first step is to use saturation to compute the eccentricity of the two saturated nodes. Notice that we do not know a priori which pair of neighbours will become saturated. We can nevertheless ensure that when they become saturated they will know their eccentricity. To do so, it is enough to include, in the M message sent by an entity x to its neighbour y, the maximum of distance from x to the nodes in $T[x-y]$, increased by 1. In this way, a saturated node s will know $d[s,y]$ for each neighbour y; thus, it can determine its eccentricity (Exercise 6.38).

Our goal is to have *all* nodes determine their eccentricity, not just the saturated ones. The interesting thing is that the information available at each entity at the end of the saturation stage is *almost* sufficient to make them compute their own eccentricity.

Consider an entity u; it sent the M message to its parent v, after it received one from all its other neighbours; the message from $y \neq v$ contained $d[u,y]$. In other words, u knows already the maximum distance from all the entities *except* the ones in the tree $T[v-u]$. Thus, the only information u is missing is $d[u,v] = Max\{d(u,y) : y \in T[v-u]\}$. Notice that (Exercise 6.39)

$$d[u,v] = Max\{d(u,y) : y \in T[v-u]\} = 1 + Max\{d[v,z] : z \neq u \in N(v)\}. \qquad (17)$$

Summarizing, every node, except the saturated ones, are missing one piece of information: the maximum distance from the nodes on the other side of the link connecting it to its parent. If the parents could provide this information, the task can be completed. Unfortunately, the parents are also missing information, unless they are the saturated nodes.

The saturated nodes have all the information they need. They also have the information their neighbours are missing: let s be a saturated node, and x be an unsaturated neighbour; x is missing the information $d[x,s]$; by Equation 17, this is exactly $d[x,s] = 1 + Max\{d[s,z] : x \neq z \in N(s)\}$, and s knows all the $d[s,z]$ (they were included in the M messages it received. So, the saturated nodes s can provide the needed information to their neighbours, who can then compute their eccentricity. The nice property is that now these neighbours have the information required by their own neighbours (further away from the saturated nodes). Thus, the Resolution stage of Full Saturation can be used to provide the missing information: starting from the saturated nodes, once an entity receives the missing information from a neighbour, it will compute its eccentricity and provide the missing information to all its other neighbours.

IMPORTANT. Notice that, in the Resolution stage, an entity sends *different* information to each of its neighbours ! Thus, unlike the Resolution we used so far, it is *not* a notification.

The protocol *Eccentricities* will thus be a Full Saturation where the procedures Initialize,

Prepare_Message and Process_Message are as shown in Fig. 24. The rule for handling the reception of the message, the procedure Resolve, and the procedure to calculate the eccentricity are also shown in Fig. 24.

Notice that, even though each node receives a different message in the resolution stage, only one message will be received by each node in that stage, except the saturated nodes which will receive none. Thus, the message cost of protocol *Eccentricities* will be exactly as the one of *MinF-Tree*, and so will the time cost:

$$\mathbf{M}[\mathit{Eccentricities}] = 3n + k_\star - 4 \leq 4n - 4. \tag{18}$$

$$\mathbf{T}[\mathit{Eccentricities}] = \mathbf{T}[MinF - Tree] \tag{19}$$

3.5 Center Finding

A *center* is a node from which the maximum distance to all other nodes is minimized. A network might have more than one center. The Center Finding problem (**Center**) is to make each entity aware of whether or not it is a center by entering the appropriate terminal status *center* or *not-center*, respectively.

3.5.1 A Simple Protocol

To solve **Center** we can use the fact that a center is exactly a node with smallest eccentricity. Thus a solution protocol consists of finding the minimum among all eccentricities, combining the protocols we have developed so far:

(1) Execute protocol *Eccentricities*;
(2) Execute the last two stages (saturation and resolution) of *MinF-Tree*.

Part (1) will be started by the initiators; part (2) will be started by the leaves once, upon termination of their execution of *Eccentricities*, they know their eccentricity; the saturation stage of *MinF-Tree* will determine at two new saturated nodes the minimum overall eccentricity, and will be by them broadcasted in the notification stage. At that time, an entity can determine if it is a center of not.

This approach will cost $3n + k_\star - 4$ messages for part (1) and $n + n - 2 = 2n - 2$ for part (2), for a total of $5n + k_\star - 6 \leq 6n - 6$ messages.

The time costs are no more than: $\mathbf{T}$[Eccentricities] + $2d \leq 4d$.

```
PROCESSING
          Receiving("Resolution", dist)
          begin
               Resolve;
          end
```

```
Procedure Initialize
begin
     Distance[x]:= 0;
end

Procedure Prepare_Message
begin
     maxdist:= 1+ Max{Distance[*]};
     M:=("Saturation", maxdist);
end

Procedure Resolve
begin
     Process_Message;
     Calculate_Eccentricity;
     forall  y ∈ N(x) − {parent} do
        maxdist:= 1 + Max{Distance[z]:  z ∈ N(x) − {parent, y}};
        send(‘‘Resolution’’, maxdist) to y;
     endfor
     become DONE;
end

Procedure Process_Message
begin
     Distance[sender]:= Received_distance;
end

Procedure Calculate_Eccentricity
begin
     r(x):= Max{Distance[z]:  z ∈ N(x)};
end
```

Figure 24: New Rule and Procedures used for computing the Eccentricities

3.5.2 A Refined Protocol

An improvement can be derived by exploiting the structure of the problem in more details. Recall that $d[x,y] = Max\{d(x,z) : z \in T[y-x]\}$ is the longest distance between x and the nodes in $T[y-x]$. Let $d_1[x]$ and $d_2[x]$ be the largest and second-largest of all $\{d[x,y] : y \in N(x)\}$. The centers of a tree have some very interesting properties. Among them:

Lemma 3.2 *In a tree there is either a unique center or there are two centers and they are neighbours.*

Lemma 3.3 *In a tree all centers lie on all diametral paths*

Lemma 3.4 *A node x is a center if and only if $d_1[x] - d_2[x] \leq 1$; if strict inequality holds, then x is the only center.*

Lemma 3.5 *Let y and z be neighbours of x such that $d_1[x] = d[x,y]$ and $d_2[x] = d[x,z]$. If $d[x,y] - d[x,z] > 1$, then all centers are in $T[y-x]$.*

Lemma 3.4 gives us the tool we need to devise a solution protocol: an entity x can determine whether or not it is a center, provided it knows the value $d[x,y]$ for each of its neighbours y. But this is exactly the information that was provided to x by protocol *Eccentricities* so it could compute $r(x)$.

This means that, to solve **Center** it suffices to execute *Eccentricities.* Once an entity has all the information to compute its radius, it will check whether the largest and the second largest received values differ at most by one; if so, it becomes *center*, otherwise *not-center*. Thus, the solution protocol *Center_Tree* is obtained from *Eccentricities* adding this test and some bookkeeping (Exercise 6.40).

The time and message costs of *Center_Tree* will be exactly the same of *Eccentricities.*

$$\mathbf{M}[Center_Tree] = 3n + k_\star - 4 \leq 4n - 4. \tag{20}$$

$$\mathbf{T}[Center_Tree] = \mathbf{T}[FullSaturation] \tag{21}$$

3.5.3 An Efficient Plug-In

The solutions we have discussed are a *full protocols.* In some circumstances however, a *plug-in* is sufficient; e.g., when the centers must then start another global task. In these circumstances, the goal is just for the centers to know that they are centers.

In such a case, we can construct a more efficient mechanism, always based on saturation, using the resolution stage in a different way.

The properties expressed by Lemmas 3.4 and 3.5 give us the tools we need to devise the plug-in.

In fact, by Lemma 3.4, x can determine whether or not it is a center once it knows the value $d[x, y]$ for each of its neighbours y. Furthermore, if x is not a center, by Lemma 3.5, this information is sufficient to determine in which subtree $T[y - x]$ a center resides.

Thus, the solution is to collect such values at a node x; determine whether x is a center; and, if not, *move towards* a center until it is reached.

In order to collect the information needed, we can use the first two stages (Wakeup and Saturation) of protocol *Eccentricities*. Once a node becomes saturated, it can determine whether it is a center by checking whether the largest and the second largest received values differ at most by one. If it is not a center, it will know that the center(s) must reside in the direction from which the largest value has been received. By keeping track at each node (during the saturation stage) of which neighbour has sent the largest value, the direction of the center can also be determined. Furthermore, a saturated node can decide whether it or its parent is closest to a center.

The saturated node, say x, closest to a center will then send a "Center" message, containing the second largest received value increased by one, in the direction of the center. **explain Why.** A processing node receiving such a message will, in turn, be able to determine whether it is a center and, if not, the direction towards the center(s).

Once the message arrives at a center c, c will be able to determine if it is the only center or not (using Lemma 3.4); in case, it will know which neighbour is the other center, and will notify it.

The *Center Finding* plug-in will then be the *Full Saturation* plug-in with the addition of the "Center" message traveling from the saturated nodes to the centers. In particular, the routines Initialize, Process_Message, Prepare_Message, Resolve and the new rule governing the reception of the "Center" messages is shown in Fig. 25.

The message cost of this plug-in is easily determined by observing that, after the Full Saturation plug-in is applied, a message will travel from the saturated node s (closest to a center) to its furthermost center c; hence, $d(s, c)$ additional messages are exchanged. Since $d(s, c) \leq n/2$, then the total number of message exchanges performed is

$$\mathbf{M}[Center - Finding] = 2.5n + k_\star - 2 \leq 3.5n - 2. \tag{22}$$

3.6 Other Computations

The simple modifications to the basic technique that we have discussed in the previous sections can be applied to efficiently solve a variety of other problems.

Following is a sample of them, and the key properties employed towards their solution.

```
PROCESSING
        Receiving(‘‘Center’’, value)
        begin
            Process_Message;
            Resolve;
        end
```

```
Procedure Initialize
begin
    Max_Value := 0;
    Max2_Value := 0;
end

Procedure Prepare_Message
begin
    M:=("Saturation", Max_Value+1);
end

Procedure Process_Message
begin
    if Max_Counter < Received_value then
      Max2_Value := Max_Value;
      Max_Value := Received_Value;
      Max_Neighbour := sender;
    else
      if Max2_Value < Received_value then
        Max2_Value := Received_value;
      endif
    endif
end

Procedure Resolve
begin
    if Max_Value - Max2_Value = 1 then
      if Max_Neighbour ≠ parent then
        send(Center,Max2_Value) to Max_Neighbour;
      endif
      become CENTER;
    else
      if Max_Value - Max2_Value > 1 then
        send(Center,Max2_Value) to Max_Neighbour;
      else
        become CENTER;
      endif
    endif
end
```

Figure 25: Transforming Saturation into an efficient Plug-In for Center Finding

3.6.1 Finding a Median

A *median* is a node from which the average distance to all nodes in the network is minimized. Since a median obviously minimizes the sum of the distances to all other nodes, it is also called a *communication center* of the network.

In a tree, the key properties are:

Lemma 3.6 *In a tree there either is a unique median or there are two medians and they are neighbours.*

Given a node x, and a sub-tree T', let $g[T,x] = \Sigma_{y \in T} d(x,y)$ denote the sum of all distances between x and the nodes in T, and let $G[x,y] = g[T,x] - g[T,y] = n + 2 - 2 * |T[y-x]|$; then

Lemma 3.7 *Entity x is a median if and only if $G[x,y] >= 0$ for all neighbours y.*

Furthermore,

Lemma 3.8 *If x is not the median, there exists a unique neighbour y such that $G[y,x] < 0$; such a neighbour lies in the path from x to the median.*

Using these properties, it is simple to construct a full protocol as well as an efficient plug-in, following the same approaches used for center finding. (Exercise 6.41)

3.6.2 Finding Diametral Paths

A diametral path is a path of longest length. In a network there might be more than one diametral path. The problem we are interested in is to identify all these paths. In distributed terms, this means that each entity need to know if it is part of *a* diametral path or not, entering an appropriate status (e.g., *on-path* or *off-path*).

The key property to solve this problem is:

Lemma 3.9 *A node x is on a diametral path if and only if $d_1[x] + d_2[x] = d$.*

Thus, a solution strategy will be to determine $d, d_1[x]$ and $d_2[x]$ at every x, and then use Lemma 3.9 to decide the final status. A full protocol efficiently implementing this strategy can be designed using the tools developed so far (Exercise 6.45)

Consider now designing a plug-in instead of a full protocol; that is, we are only interested that the entities on diametral paths (and only those) become aware of it.

In this case, the other key property is Lemma 3.4: every center lies on every diametral path. This gives us a starting point to find the diametral paths: the centers. To continue, we can then use Lemma 3.9. In other words, we first find the centers (note: they know the diameter), and then propagate the information along the diametral paths. A center (or for that matter, a node on a diametral path) does not know a priori which one of its neighbours is also on a diametral path. It will thus send the needed information to *all* its neighbours which, upon receiving it, will determine whether on not they are on such a path; if so, they continue the execution. (Exercise 6.46)

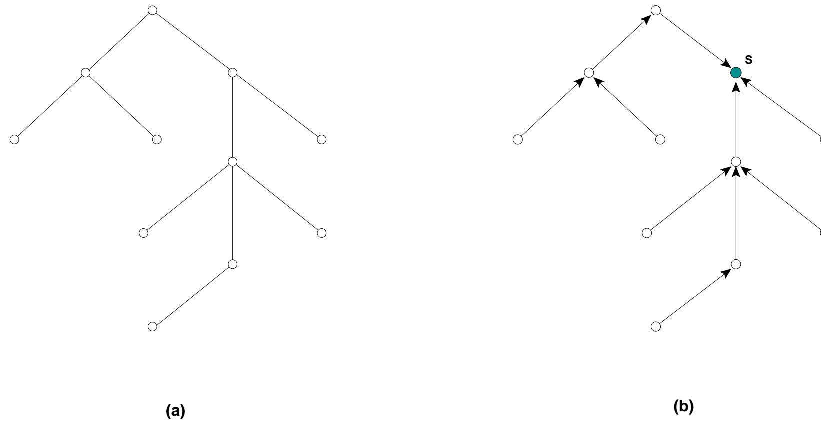

Figure 26: (a) A tree T; (b) the same tree rooted in r $T_{[r]}$.

3.7 Computing in Rooted Trees

3.7.1 Rooted Trees

In some cases, the tree T is actually *rooted*; that is, there is a distinct node, r, called the *root*, and all links are oriented towards r. In this case, the tree T will be denoted by $T_{[r]}$.

If link (x, y) is oriented from y to x, x is called the *parent* of y, and y is said to be a *child* of x. Similarly, a *descendant* of x is any entity z for which there is a directed path from z to x, and an *ancestor* of x is any entity z for which there is a directed path from x to z.

Two important properties of a rooted tree are that the root has no parent, while every other node has only one parent (see Figure 26).

Before examining how to compute in rooted trees, let us first observe the important fact that transforming a tree into a rooted one might be an *impossible* task !

Theorem 3.1 *The problem of transforming trees into rooted ones is deterministically* unsolvable *under* **R**.

Proof. Recall that deterministically unsolvable means that there is no deterministic protocol which always correctly terminates within finite time. To see why this is true, consider the simple tree composed of two entities x and y connected by links labeled as shown in Figure 27. Let the two entities have identical initial values (the symbols x, y are used only for description purposes). If a solution protocol A exists, it must work under any conditions of message delays (as long as they are finite) and regardless of the number of initiators. Consider a *synchronous schedule* (i.e., an execution where communication delays are unitary) and let both entities start the execution of A simultaneously. Since they are identical (same initial status and values, same port labels), they will execute the same rule, obtain the same

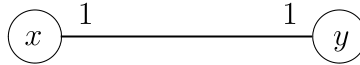

Figure 27: It is impossible to transform this tree into a rooted one.

results (thus, continuing to have the same local values), compose and send (if any) the same messages; enter the same (possibly new) status. In other words, they will remain identical. In the next time unit, all sent messages (if any) will arrive and be processed. If one entity receives a message, the other will receive the same message at the same time, perform the same local computation, compose and send (if any) the same messages; enter the same (possibly new) status. And so on. In other words, the two entities will continue to be identical. If A is a solution protocol, it must terminate within finite time; when this occurs, one entity, say x, becomes the root. But since both entities will always have the same state in this execution, also y will become root, contradicting the fact that A is correct. Thus, no such a solution algorithm A exists. ■

This means that being in a rooted tree is considerably different from being in a tree. Let us see how to exploit this difference.

3.7.2 Convergecast

The orientation of the links in a rooted tree is such that each entity has a notion of "up" (i.e., towards the root) and "down" (i.e., away from the root). If we are in a rooted tree, we can obviously exploit the availability of this globally consistent orientation. In particular, in the *saturation* technique, the process performed in the saturation stage can be simplified as follows:

Convergecast

1. a leaf sends its message to its parent;
2. each internal node waits until it receives a message from all its children; it then sends a message to its parent.

In this way, the root (that does not have a parent) will be the sole *saturated* node, and will start the resolution stage.

This simplified process is called *convergecast*. If we are in a rooted tree, we can solve all the problems we discussed in the previous section (minimum-finding, center finding, etc) using convergecast in the saturation stage.

In spite of its greater simplicity, the savings in cost due to convergecast is only 1 message (Exercise 6.47). Clearly, such an amount alone does not justify the difference between general

trees and rooted ones. There are however other advantages in rooted trees, as we will see later.

3.7.3 Totally Ordered Trees

In addition to the globally consistent orientation "up and down", a rooted tree has another powerful property. In fact, the port numbers at a node are distinct; thus, they can be sorted, e.g. in increasing order, and the corresponding links can be ordered accordingly. This means that the entire tree is *ordered*. As a consequence, also the nodes can be totally ordered e.g. according to a pre-order traversal (see Figure 28).

Note that a node might not be aware of its order number in the tree, although this information can be easily acquired in the entire tree (Exercise 6.49). This means that, *in a rooted tree the root assign unique ids to the entities.* This fact shows indeed the power of rooted trees.

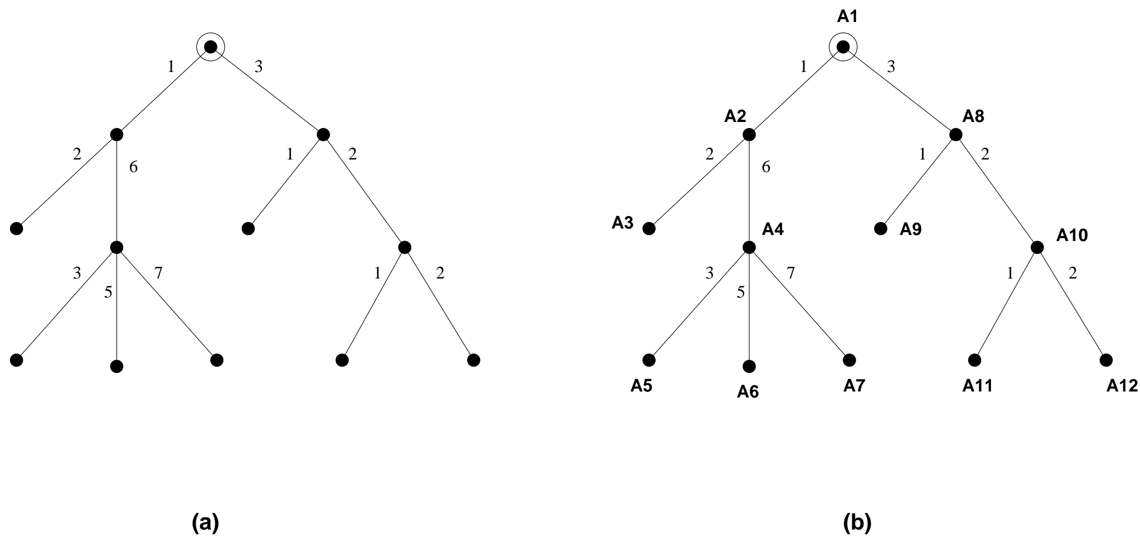

Figure 28: A rooted tree is an ordered tree and unique names can be given to the nodes.

The fact that a rooted tree is totally ordered can be exploited also in other computations. Following are two examples.

Example: Choosing a Random Entity.

In many systems and applications, it is necessary to occasionally select an entity at random. This occurs for instance in routing systems where, to reduce congestion, a message is first sent to an intermediate destination chosen at random, and then delivered from there to the final destination. The same random selection is made e.g., for coordination of a computation,

for control of a resource, etc. The problem is how to determine an entity at random. Let us concentrate on *uniform* choice; that is, every entity must have the same probability, $\frac{1}{n}$, of being selected.
In a rooted tree, it becomes easy for the root to select uniformly at random an entity. Once unique names have been assigned in pre-order to the nodes and the root knows the number n of entities, the root needs only to locally choose a number uniformly at random between 1 and n; the entity with such a name will be the selected one. At this point, the only thing that the root r still has to do is to efficiently communicate to the selected entity x the result of the selection.
Actually, it is not necessary to assign unique names to the identities; in fact, it suffices that each entity knows the number of descendents of each of its children, and the entire process (from initial notification to all to final notification to x) can be performed with at most $2(n-1) + d_T(s,x)$ messages and $2r(s) + d_T(s,x)$ ideal time units (Exercise 6.50).

Example: Choosing at Random from a Distributed Set
An interesting computation is the one of choosing at random an element of a set of data distributed (without replication) among the entities. The setting is that of a set D partitioned among the entities; that is, each entity x has a subset $D_x \subseteq D$ of the data where $\cup_x D_x = D$ and, for $x \neq y$, $D_x \cap D_y = \emptyset$.
Let us concentrate again on *uniform* choice; that is, every data item must have the same probability, $\frac{1}{|D|}$ of being selected. How can this be achieved ?

IMPORTANT. Choosing first an entity uniformly at random and then choosing an item uniformly at random in the set stored there will NOT give a uniformly random choice from the entire set !! (Exercise 6.51).

Interestingly, this problem can be solved with a technique similar to that used for selecting an entity at random, and with the same cost (Exercise 6.52).

3.7.4 Application: Termination Detection

Convergecast can be used whenever there is a rooted spanning-tree. We will now see an application of this fact.

It is a "fact of life" in distributed computing that entities can terminate the execution of a protocol at different times; furthermore, when an entity terminates is usually unaware of the status of the other entities. This is why we differentiate between *local* termination (i.e. of the entity) and *global* termination (i.e., of the entire system).

For example, with the broadcast protocol *Flooding* the initiator of the broadcast does not know when the broadcast is over. To ensure that the initiator of the broadcast becomes aware of when global termination occurs, we need to use a different strategy.

To develop this strategy, recall that, if an entity s performs a *Flood+Reply* (e.g., protocol *Shout*) in a tree, the tree will become rooted in s: the initiator is the root; for every other

node y, the neighbour x from which it receives the first broadcasted message is its parent, and all the neighbours that send the positive reply (e.g., “YES” in *Shout* and *Shout+*) are its children. This means that *convergecast* can be “appended” to any *Flood+Reply* protocol.

Strategy Broadcast with Termination Detection:

1. The initiator s uses any *Flood+Reply* protocol to broadcast and construct a spanning tree $T_{[s]}$ of the network;
2. Starting from the leaves of $T_{[s]}$, the entities perform a *convergecast* on T.

At the end of the convergecast, s becomes aware of the global termination of the broadcast (Exercise 6.48).

As for the cost, to broadcast with termination detection we need just to add the cost of the convergecast to the one of the *Flood+Reply* protocol used. For example, if we use *Shout+*, the resulting protocol that we shall call *TDCast* will then use $2m + n - 1$ messages. The ideal time of *Shout+* is exactly $r(s) + 1$; the ideal time of *convergecast* is exactly the hight of the tree $T_{[s]}$, that is $r(s)$; thus, protocol *TDCast* has ideal time complexity $2r(s) + 1$. This means that termination detection can be added to broadcast with less than twice the cost of broadcasting alone.

4 SUMMARY

4.1 Summary of Problems

- **Broadcast** [Information problem] $\Longrightarrow$ A single entity has special information, that everybody must know.
  - Unique Initiator
  - *Flooding*: Messages= $\Theta(m)$; Time=$\Theta(d)$
- **Wake Up** [Information/Synchronization problem] $\Longrightarrow$ Some entities are awake; everybody must wake-up.
  - WakeUp $\equiv$ (Broadcast with multiple initiators)
  - *WFlood*: Messages= $\Theta(m)$; Time=$\Theta(d)$
- **Traversal** [Network problem] $\Longrightarrow$ Starting form the initiator, each entity is visited sequentially.
  - Unique Initiator
  - *DF-Traversal*: Messages= $\Theta(m)$; Time=$\Theta(n)$
- **Spanning-Tree Construction** [Network problem]$\Longrightarrow$ Each entity identifies the subset of neighbours in the spanning tree.
  - SPT with unique initiator $\equiv$ Broadcast
  - Unique initiator: *Shout*: Messages= $\Theta(m)$; Time=$\Theta(d)$

- Multiple-Initiators: assume Distinct Initial Values

- **Election** [Control problem] $\Longrightarrow$ One entity becomes leader, all others enter different special status.
  - Distinct Initial Values
- **Minimum Finding** [Data problem] $\Longrightarrow$ Each entity must know whether its initial value is minimum or not.
- **Center Finding** [Network problem] $\Longrightarrow$ Each entity must know whether or not it is a center of the network.

4.2 Summary of Techniques

- **flooding**: with single initiator = broadcast; with multiple initiators = wake-up.
- **flooding with reply** (*Shout*): with single initiator, it creates a spanning-tree rooted in the initiator.
- **convergecast**: in rooted trees only !
- **flooding with replies plus convergecast (TDCast)**: single initiator only !
  initiator finds out that the broadcast has globally terminated.
- **saturation**: in trees only !
- **depth-first traversal**: single initiator only !

5 Bibliographical Notes

Of the basic techniques, *flooding* is the oldest one, still currently and frequently used. The more sophisticated refinements of adding reply and a converge-cast were discussed and employed independently by Adrian Segall [11] and Ephraim Korach, Doron Rotem and Nicola Santoro [8]. Broadcasting in a linear number of messages in unoriented hypercubes is due to Stefan Dobrev and Peter Ruzicka [5]. The use of broadcast trees was first discussed by David Wall [12].
The depth-first traversal protocol was first described by Ernie Chang [3]; the first hacking improvement is due to Baruch Awerbuch [2]; the subsequent improvements were obtained by K.B. Lakshmanan, N. Meenakshi and K. Thulasiraman [9] and independently by Israel Cidon [4].
The difficulty of performing a wake-up in labelled hypercubes and in complete graphs (the latter even with unique ids) has been proved by Stefan Dobrev and Nicola Santoro [6].

The first formal argument on the impossibility of some global computations under **R** (e.g., the impossibility result for spanning-tree construction with multiple initiators) is due to Dana Angluin [1].

The saturation technique is originally due to Nicola Santoro [10]; its application to center and median finding, as well as to to rank finding (Exercise 6.43) was developed by Ephraim Korach, Doron Rotem and Nicola Santoro [7, 8].

6 Exercises, Problems, and Answers

6.1 Exercises

Exercise 6.1 *Show that protocol* Flooding *uses exactly* $2m - n + 1$ *messages.*

Exercise 6.2 *Design a protocol to broadcast without the restriction that the unique initiator must be the entity with the initial information. Write the new problem definition. Discuss the correctness of your protocol. Analyze its efficiency.*

Exercise 6.3 *Modify* Flooding *so to broadcast under the restriction that the unique initiator* must *be an entity* without *the initial information. Write the new problem definition. Discuss the correctness of your protocol. Analyze its efficiency.*

Exercise 6.4 *We want to move the system from an initial configuration where every entity is in the same status* ignorant *except one which is* knowledgeable*, to a final configuration where every entity is in the same status. Consider this problem under the standard assumptions plus Unique Initiator.*
(a) Prove that, if the unique initiator is restricted to be one of the ignorant *entities, this problem is the same as broadcasting (same solution, same costs).*
(b) Show how, if the unique initiator is restricted to be the knowledgeable *entity, the problem can be solved* without *any communication.*

Exercise 6.5 *Design a protocol to broadcast without the Bidirectional Link restriction. Discuss its correctness. Analyze its efficiency.*

Exercise 6.6 *Prove that, in the worst case, the number of messages used by protocol* WFlood *is at most* $2m$*. Show under what conditions will such a bound be achieved. Under what conditions will the protocol use only* $2m - n + 1$ *messages ?*

Exercise 6.7 *Prove that protocol* WFlood *correctly terminates under the usual restrictions BL,C, and TR.*

Exercise 6.8 *Write the protocol that implements strategy* HyperFlood.

Exercise 6.9 *Show that the subgraph* $H_k(x)$*, induced by the messages sent when using* HyperFlood *on the* k*-dimensional hypercube* H_k *with* x *as the initiator, contains no cycles.*

Exercise 6.10 *Show that for every* x*, the eccentricity of* x *in* $H_k(x)$ *is* k*.*

Exercise 6.11 *Prove that the message complexity of traversal under* **R** *is at least* m*.* (Hint: use the same technique employed in the proof of Theorem 1.1).

Exercise 6.12 *Let* G *be a* tree*. Show that, in this case, no Backedge messages will be sent in any execution of* DF_Traversal.

Exercise 6.13 *Characterize the* virtual ring *formed by an execution of* DF_Traversal *in a tree network. Show that the ring has* $2n-2$ *virtual nodes.*

Exercise 6.14 *Write the protocol* DF++.

Exercise 6.15 *Prove that protocol* DF++ *correctly performs a depth-first traversal.*

Exercise 6.16 *Show that, in the execution of* DF++, *on some back-edges there might be two "mistakes".*

Exercise 6.17 *Determine the exact number of messages transmitted in the worst case when executing* DF* *in a* complete graph.

Exercise 6.18 *Prove that in protocol* Shout, *if an entity* x *is in* Tree-neighbours *of* y, *then* y *is in* Tree-neighbours *of* x.

Exercise 6.19 *Prove that in protocol* Shout, *if an entity sends* Yes, *then it is connected to the* initiator *by a path where on every link a* Yes *has been transmitted.* (Hint: use induction)

Exercise 6.20 *Prove that the subnet constructed by protocol* Shout *contains no cycles.*

Exercise 6.21 *Prove that* **T**[Flood+Reply] = **T**[Flooding]+1.

Exercise 6.22 *Write the set of rules for protocol* Shout+.

Exercise 6.23 *Determine under what conditions on the communication delays, protocol* Shout *will construct a breadth-first spanning tree.*

Exercise 6.24 *Modify protocol* Shout *so that the initiator can determine when the broadcast is* globally *terminated. (Hint: integrate in the protocol the convergecast operation for rooted trees)*

Exercise 6.25 *Modify protocol* DF* *so that every entity determines its neighbours in the df-tree it constructs.*

Exercise 6.26 *Prove that* f_* *is exactly the number of* leaves *of the df-tree constructed by* df-SPT.

Exercise 6.27 *Prove that, in the execution of* df-SPT, *when the initiator becomes* done, *a df-tree of the network has already been constructed.*

Exercise 6.28 *Prove that, for any broadcast protocol, the graph induced by relationship "parent" is a spanning tree of the network.*

Exercise 6.29 *Prove that the bf-tree of* G *rooted in a center is a broadcast tree of* G.

Exercise 6.30 *Verify that, with multiple initiators, the optimized version* DF+ *and* DF* *of protocol* df-SPT *will always create a spanning forest of the graph depicted in Figure 14.*

Exercise 6.31 *Prove that when a node becomes* saturated *in the execution of protocol* MinF-Tree*, it knows the minimum value in the network.*

Exercise 6.32 *Prove that when a node becomes* saturated *in the execution of protocol* Funct-Tree*, it knows the value of f.*

Exercise 6.33 *Design a protocol to determine if all the entities of a tree network have positive initial values. Any number of entities can independently start.*

Exercise 6.34 *Consider a tree system where each entity has a* salary *and a* gender*. Some external investigators want to know if all entities with a salary below* $50,000 *are* female*. Design a solution protocol which can be started by any number of entities independently.*

Exercise 6.35 *Consider the same tree system of Question 6.34. The investigators now want to know if there is at least one* female *with a salary above* $50,000*. Design a solution protocol which can be started by any number of entities independently.*

Exercise 6.36 *Design an efficient protocol to compute the number of entities in a tree network. Any number of entities can independently start the protocol.*

Exercise 6.37 *Consider the same tree system of Question 6.34. The investigators now want to know how many* female *entities are in the system. Design a solution protocol which can be started by any number of entities independently.*

Exercise 6.38 *Consider the following use of the M message: a leaf will include a value $v = 1$; an internal node will include one plus the maximum of all received values. Prove that the saturated nodes will compute their maximum distance from all other nodes.*

Exercise 6.39 *Prove that for any link (u, v), $d[u, v] = Max\{d(u, y) : y \in T[v - u]\} = 1 + Max\{d(v, y) : y \in T[u - v]\} = Max\{d[v, z] : z \neq u \in N(v)\}$.*

Exercise 6.40 *Modify protocol* Eccentricities *so it can solve* **Center***, as discussed in Section 3.5.*

Exercise 6.41 Median Finding. *Construct an efficient plug-in so that the median nodes know that they are such.*

Exercise 6.42 Diameter Finding. *Design an efficient protocol to determine the* diameter *of the tree. (Hint: use Lemma 3.2.)*

Exercise 6.43 Rank Finding in Tree. *Consider a tree where each entity x has an initial value $v(x)$; these values are not necessarily distinct. The* rank *of an entity x will be the rank of its value; that is, $rank(x) = 1 + |\{y \in V : v(y) < v(x)\}$. So, whoever has the smallest value, has rank 1. Design an efficient protocol to determine the rank of a* unique initiator *(i.e., under the additional restriction UI).*

Exercise 6.44 Generic Rank Finding. *Consider the* ranking problem *described in Exercise 6.43. Design an efficient solution protocol which is* generic, *that is it works in an arbitrary connected graph.*

Exercise 6.45 Diametral Paths. *A path whose length is d is called* diametral. *Design an efficient protocol so that each entity can determine whether or not it lies on a diametral path of the tree.*

Exercise 6.46 *A path whose length is d is called* diametral. *Design an efficient plug-in so that all and only the entities on a diametral path of the tree become aware of this fact.*

Exercise 6.47 *Show that convergecast uses only* 1 *(one) message less than the saturation stage in general trees.*

Exercise 6.48 *Prove that, when a initiator of a* TDCast *protocol receives the convergecast message from all its children, the initial broadcast is globally terminated.*

Exercise 6.49 *Show how to efficiently assign a unique id to the entities in a rooted tree.*

Exercise 6.50 Random Entity Selection $\star$ *Consider the task of selecting uniformly at random an entity in a tree rooted at s. Show how to perform this task, started by the root, with at most $2(n-1)+d_T(s,x)$ messages and $2r(s)+d_T(s,x)$ ideal time units. Prove both correctness and complexity.*

Exercise 6.51 *Show why choosing uniformly at random a site and then choosing uniformly at random an element from that site from that site is not the same as choosing uniformly at random an element from the entire set.*

Exercise 6.52 Random Item Selection $\star\star$ *Consider the task of selecting uniformly at random an item from a set of data partitioned among the nodes of a tree rooted at s. Show how to perform this task, started by the root, with at most $2(n-1)+d_T(s,x)$ messages and $2r(s)+d_T(s,x)$ ideal time units. Prove both correctness and complexity.*

6.2 Problems

Problem 6.1 *Develop an efficient solution to the Traversal problem without the Bidirectional Links assumption.*

Problem 6.2 *Develop an efficient solution to the Minimum Finding problem in a* hypercube *with a* unique initiator *(i.e., under the additional restriction UI). Note that the values might not be distinct.*

Problem 6.3 *Solve the Minimum Finding problem is a system where there is already a leader; that is, under restrictions* $\mathbf{R}\cup UI$. *Note that the values might not be distinct. Prove the correctness of your solution, and analyze its efficiency.*

6.3 Answers to Exercises

Answer to Exercise 6.13
A node appears several times in the virtual ring; more precisely, there is an instance of node z in R for each time z has received a Token or a Finished message. Let x be the initiator; node x sends a token to each of its neighbours sequentially and receive a Finished message from each. Every node $y \neq x$ receives exactly one Token (from its parent), and will send one to all its other neighbours (its children); it will also receive a Finished message from all its children and send one to its parent. In other words every node z, including the initiator x, will appear $n(z) = |N(z)|$ times in the virtual ring. The total number of (virtual) nodes in the virtual ring is therefore $\sum_{z \in V} |N(z)| = 2m = 2(n-1)$.

Answer to Exercise 6.16
Consider a Ring network with the three nodes x, y, z. Assume that entity x holds the *Token* initially. Consider the following sequence of events that take place successively in time as a result of the execution of the *DF++* protocol: x sends *Visited* messages to y and z, sends the *Token* to y and waits for a (*Visited* or *Return*) reply from y. Assume that the link (x, z) is very very slow.
When y receives the *Token* from x, it sends to z a *Visited* message and then the *Token*. Assume that when z receives the *Token*, the *Visited* message from x has not arrived yet; hence z sends *Visited* to x followed by the *Token*. This is the first mistake: *Token* is sent on a backedge to x which has already been visited.
When z finally receives the *Visited* message from x, it realizes the *Token* it sent to x was a mistake. Since it has no other unvisited neighbours, z sends a *Return* message back to y. Since y has no other unvisited neighbours, it will then send a *Return* message back to x. Assume that when x receives the *Return* message from y, x has not received yet neither the *Visited* nor the *Return* messages sent by z. Hence, x considers z as an unvisited neighbour, and sends the *Token* to z. This is the second mistake on the backedge between x and z.

Answer to Exercise 6.19
Suppose some node x is not reachable from s in the graph T induced by the "parent" relationship; This means that x never sent the Yes messages; this implies that x never received the question Q. This is impossible because, since flooding is correct, every entity will receive Q; thus, so such x exists.

Answer to Exercise 6.20
Suppose the graph T induced by the "parent" relationship (i.e., the Yes messages) contains a directed cycle $x_0, x_1, \ldots x_{k-1}$; that is, x_i is the parent of x_{i+1} (operations on the indices are modulo k). This cycle cannot contain the initiator s (because it does not sends any Yes). We know (Exercise 6.19) that in T there is a path from s to each node, including those in the cycle. This means that there will be in T a node y not in the cycle which is connected to a node x_i in the cycle. This means that x_i sent a Yes message to y; but since it is in the cycle, it also sent a Yes message to x_{i-1} (operations on the indices are modulo k). This is impossible because an entity sends no more than one Yes message.

Answer to Exercise 6.31
First show that if a node x sends M to neighbour y, N contains the smallest value in $T[x-y]$; then, since a saturated node receives by definition a M message from all neighbours, it knows the minimum value in the network. Prove that value sent by x to y in M is the minimum value in $T[x-y]$ by induction on the height h of $T[x-y]$. Trivially true if $h = 1$, i.e., x is a leaf. Let it be true up to $k \geq 1$; we will now show it is true for $h = k + 1$. x sends M to y because it has received a value from all its other neighbours $y_1, y_2, \ldots$; Since the height of $(T[y_i - x])$ is less than h, then by inductive hypothesis the value sent by y_i to x is the minimum value in $(T[y_i - x])$. This means that the smallest among $v(x)$ and all the values received by x is the minimum value in $T[x-y]$; this is exactly what x sends to y.

Answer to Exercise 6.41
It is clear that, if node x knows $|T[y-x]|$ for all neighbours y, it can compute $G[y,x]$ and decide whether x is itself a median and, if not, determine the direction of the median. Thus, to find a median, is sufficient to modify the basic technique to supply this information to the elected node from which the median is approached. This is done by providing two counters, m_1 and m_2, with each M message: when a node x sends a M message to y, then $m_1 = g[T[y-x], y] - 1$ and $m_2 = |T[y-x]| - 1$. An active node x processes all received M messages so that, before it sends M to the last neighbour y, it knows $G[T[x-z], x]$ and $|T[z-x]|$ for all other neighbours z. In particular, the elected node can determine whether it is the median and, if not, can send a message towards it; a node receiving such a message will, in turn, perform the same operations until a median is located. Once again, the total number of exchanged messages is the ones of the Full saturation plug-in plus $d(s, med)$, where s is the saturated closer to the medians, and med is the median furthermost from x.

Partial Answer to Exercise 6.48
By induction on the hight of the rooted tree, prove that, in a *TDCast* protocol, when an entity x receives the convergecast message from all its children, all its descendants have locally terminated the broadcast.

Partial Answer to Exercise 6.49
Perform first (1) a broadcast from the root to notify all entities of the start of the protocol, and (2) a convergecast, to collect at each entity the number of its descendents. Use then this information to assign distinct values to the entities according to a pre-order traversal of the tree.

Partial Answer to Exercise 6.51.
Show that data items from smaller sets will be chosen with higher probabbility than that of items from larger sets.

References

[1] Dana Angluin. Local and global properties in networks of processors. In *Proc. of the 12th ACM STOC Symposium on Theory of Computing*, pages 82–93, 1980.

[2] Baruch Awerbuch. A new distributed depth-first search algorithm. *Information Processing Letters*, 20:147–150, 1985.

[3] Ernest J. H. Chang. Echo algorithms: Depth parallel operations on general graphs. *IEEE Transactions on Software Engineering*, SE-8(4):391–401, July 1982.

[4] Israel Cidon. Yet another distributed depth-first search algorithm. *Information Processing Letters*, 26:301–305, 1987.

[5] Stefan Dobrev and Peter Ruzicka. Linear broadcasting and o (n loglog n) election in unoriented hypercubes. In *Proc. of the 4th International Colloquium on Structural Information and Communication Complexity, (Sirocco'97)*, Ascona, July 1997. To appear.

[6] Stefan Dobrev and Nicola Santoro. On the difficulty of waking up. In *print*, 2004.

[7] Ephraim Korach, Doron Rotem, and Nicola Santoro. Distributed algorithms for ranking the nodes of a network. In *13th SE Conf. on Combinatorics, Graph Theory and Computing*, volume 36 of *Congressus Numeratium*, pages 235–246, Boca Raton, February 1982.

[8] Ephraim Korach, Doron Rotem, and Nicola Santoro. Distributed algorithms for finding centers and medians in networks. *ACM Transactions on Programming Languages and Systems*, 6(3):380–401, July 1984.

[9] K.B. Lakshmanan, N. Meenakshi, and K. Thulasiraman. A time-optimal message-efficient distributed algorithm for depth-first search. *Information Processing Letters*, 25:103–109, 1987.

[10] Nicola Santoro. Determining topology information in distributed networks. In *Proc. 11th SE Conf. on Combinatorics, Graph Theory and Computing*, Congressus Numeratium, pages 869–878, Boca Raton, February 1980.

[11] Adrian Segall. Distributed network protocols. *IEEE Transactions on Information Theory*, IT-29(1):23–35, Jan 1983.

[12] David Wall. Mechanisms for broadcast and selective broadcast. Technical report, Ph.D Thesis, Stanford University, June 1980.