CHAPTER 1

Distributed Computing Environments

The universe in which we will be operating will be called a *distributed computing environment*. It consists of a finite collection $\mathcal{E}$ of computational *entities* communicating by means of *messages*. Entities communicate with other entities to achieve a common goal; for example, to perform a given task, to compute the solution to a problem, to satisfy a request either from the user (i.e., outside the environment) or from other entities. In this chapter, we will examine this universe in some detail.

1.1 ENTITIES

The computational unit of a distributed computing environment is called an *entity* . Depending on the system being modeled by the environment, an entity could correspond to a process, a processor, a switch, an agent, and so forth in the system.

Capabilities Each entity $x \in \mathcal{E}$ is endowed with local (i.e., private and nonshared) memory M_x. The *capabilities* of x include access (storage and retrieval) to local memory, local processing, and communication (preparation, transmission, and reception of messages). Local memory includes a set of *defined registers* whose values are always initially defined; among them are the *status register* (denoted by *status*(x)) and the *input value register* (denoted by *value*(x)). The register *status*(x) takes values from a finite set of system states $\mathcal{S}$; the examples of such values are "Idle," "Processing," "Waiting,"... and so forth.

In addition, each entity $x \in \mathcal{E}$ has available a local *alarm clock* c_x which it can set and reset (turn off).

An entity can perform only four types of *operations*:

- local storage and processing
- transmission of messages
- (re)setting of the alarm clock
- changing the value of the status register

Design and Analysis of Distributed Algorithms, by Nicola Santoro

Note that, although setting the alarm clock and updating the status register can be considered as a part of local processing, because of the special role these operations play, we will consider them as distinct types of operations.

External Events The behavior of an entity $x \in \mathcal{E}$ is *reactive*: x only responds to external stimuli, which we call *external events* (or just *events*); in the absence of stimuli, x is inert and does nothing. There are three possible external events:

- arrival of a message
- ringing of the alarm clock
- spontaneous impulse

The arrival of a message and the ringing of the alarm clock are the events that are external to the entity but originate within the system: The message is sent by another entity, and the alarm clock is set by the entity itself.

Unlike the other two types of events, a spontaneous impulse is triggered by forces external to the system and thus outside the universe perceived by the entity. As an example of event generated by forces external to the system, consider an automated banking system: its entities are the bank servers where the data is stored, and the automated teller machine (ATM) machines; the request by a customer for a cash withdrawal (i.e., update of data stored in the system) is a spontaneous impulse for the ATM machine (the entity) where the request is made. For another example, consider a communication subsystem in the open systems interconnection (OSI) Reference Model: the request from the network layer for a service by the data link layer (the system) is a spontaneous impulse for the data-link-layer entity where the request is made. Appearing to entities as "acts of God," the spontaneous impulses are the events that start the computation and the communication.

Actions When an external event e occurs, an entity $x \in \mathcal{E}$ will react to e by performing a finite, indivisible, and terminating sequence of operations called *action*.

An action is indivisible (or atomic) in the sense that its operations are executed without interruption; in other words, once an action starts, it will not stop until it is finished.

An action is terminating in the sense that, once it is started, its execution ends within finite time. (Programs that do not terminate cannot be termed as actions.)

A special action that an entity may take is the *null* action **nil**, where the entity does not react to the event.

Behavior The nature of the action performed by the entity depends on the nature of the event e, as well as on which status the entity is in (i.e., the value of $status(x)$) when the events occur. Thus the specification will take the form

$$\text{Status} \times \text{Event} \longrightarrow \text{Action},$$

which will be called a *rule* (or a method, or a production). In a rule $s \times e \longrightarrow A$, we say that the rule is enabled by (s, e).

The behavioral specification, or simply *behavior*, of an entity x is the set $B(x)$ of all the rules that x obeys. This set must be *complete* and *nonambiguous*: for every possible event e and status value s, there is one and only one rule in $B(x)$ enabled by (s,e). In other words, x must always know exactly what it must do when an event occurs.

The set of rules $B(x)$ is also called *protocol* or *distributed algorithm* of x.

The behavioral specification of the entire distributed computing environment is just the collection of the individual behaviors of the entities. More precisely, the *collective behavior* $B(\mathcal{E})$ of a collection $\mathcal{E}$ of entities is the set

$$B(\mathcal{E}) = \{B(x) : x \in \mathcal{E}\}.$$

Thus, in an environment with collective behavior $B(\mathcal{E})$, each entity x will be acting (behaving) according to its distributed algorithm and protocol (set of rules) $B(x)$.

Homogeneous Behavior A collective behavior is *homogeneous* if all entities in the system have the same behavior, that is, $\forall x, y \in \mathcal{E},\ B(x) = B(y)$.

This means that to specify a homogeneous collective behavior, it is sufficient to specify the behavior of a single entity; in this case, we will indicate the behavior simply by B. An interesting and important fact is the following:

Property 1.1.1 *Every collective behavior can be made homogeneous.*

This means that if we are in a system where different entities have different behaviors, we can write a new set of rules, the *same* for all of them, which will still make them behave as before.

Example Consider a system composed of a network of several identical workstations and a single server; clearly, the set of rules that the server and a workstation obey is not the same as their functionality differs. Still, a single program can be written that will run on both entities without modifying their functionality. We need to add to each entity an input register, *my_role*, which is initialized to either “workstation” or “server,” depending on the entity; for each status–event pair (s, e) we create a new rule with the following action:

$$s \times e \longrightarrow \{\ \textbf{if}\ \textit{my\_role} = \text{workstation}\ \textbf{then}\ A_{\text{workstation}}\ \ \textbf{else}\ A_{\text{server}}\ \ \textbf{endif}\ \},$$

where $A_{\text{workstation}}$ (respectively, A_{server}) is the original action associated to (s, e) in the set of rules of the workstation (respectively, server). If (s, e) did not enable any rule for a workstation (e.g., s was a status defined only for the server), then $A_{\text{workstation}} = \textbf{nil}$ in the new rule; analogously for the server.

It is important to stress that in a homogeneous system, although all entities have the same behavioral description (software), they do not have to act in the same way;

their difference will depend solely on the initial value of their input registers. An analogy is the legal system in democratic countries: the law (the set of rules) is the same for every citizen (entity); still, if you are in the police force, while on duty, you are allowed to perform actions that are unlawful for most of the other citizens.

An important consequence of the homogeneous behavior property is that we can concentrate solely on environments where all the entities have the same behavior. From now on, when we mention behavior we will always mean homogeneous collective behavior.

1.2 COMMUNICATION

In a distributed computing environment, entities communicate by transmitting and receiving *messages*. The *message* is the unit of communication of a distributed environment. In its more general definition, a message is just a *finite sequence of bits*.

An entity communicates by transmitting messages to and receiving messages from other entities. The set of entities with which an entity can communicate directly is not necessarily $\mathcal{E}$; in other words, it is possible that an entity can communicate directly only with a subset of the other entities. We denote by $N_{\text{out}}(x) \subseteq \mathcal{E}$ the set of entities to which x can transmit a message directly; we shall call them the *out-neighbors* of x . Similarly, we denote by $N_{\text{in}}(x) \subseteq \mathcal{E}$ the set of entities from which x can receive a message directly; we shall call them the *in-neighbors* of x.

The neighborhood relationship defines a directed graph $\vec{G} = (V, \vec{E})$, where V is the set of vertices and $\vec{E} \subseteq V \times V$ is the set of edges; the vertices correspond to entities, and $(x, y) \in \vec{E}$ if and only if the entity (corresponding to) y is an out-neighbor of the entity (corresponding to) x.

The directed graph $\vec{G} = (V, \vec{E})$ describes the *communication topology* of the environment. We shall denote by $n(\vec{G})$, $m(\vec{G})$, and $d(\vec{G})$ the number of vertices, edges, and the diameter of $\vec{G}$, respectively. When no ambiguity arises, we will omit the reference to $\vec{G}$ and use simply n, m, and d.

In the following and unless ambiguity should arise, the terms vertex, node, site, and entity will be used as having the same meaning; analogously, the terms edge, arc, and link will be used interchangeably.

In summary, an entity can only receive messages from its in-neighbors and send messages to its out-neighbors. Messages received at an entity are processed there in the order they arrive; if more than one message arrive at the same time, they will be processed in arbitrary order (see Section 1.9). Entities and communication may fail.

1.3 AXIOMS AND RESTRICTIONS

The definition of distributed computing environment with point-to-point communication has two basic *axioms*, one on communication delay, and the other on the local orientation of the entities in the system.

Any additional assumption (e.g., property of the network, a priori knowledge by the entities) will be called a *restriction*.

1.3.1 Axioms

Communication Delays Communication of a message involves many activities: preparation, transmission, reception, and processing. In real systems described by our model, the time required by these activities is unpredictable. For example, in a communication network a message will be subject to queueing and processing delays, which change depending on the network traffic at that time; for example, consider the delay in accessing (i.e., sending a message to and getting a reply from) a popular web site.

The totality of delays encountered by a message will be called the *communication delay* of that message.

Axiom 1.3.1 Finite Communication Delays
In the absence of failures, communication delays are finite.

In other words, in the absence of failures, a message sent to an out-neighbor will eventually arrive in its integrity and be processed there. Note that the Finite Communication Delays axiom does not imply the existence of any bound on transmission, queueing, or processing delays; it only states that in the absence of failure, a message will arrive after a finite amount of time without corruption.

Local Orientation An entity can communicate directly with a subset of the other entities: its neighbors. The only other axiom in the model is that an entity can distinguish between its neighbors.

Axiom 1.3.2 Local Orientation
An entity can distinguish among its in-neighbors.
An entity can distinguish among its out-neighbors.

In particular, an entity is capable of sending a message only to a specific out-neighbor (without having to send it also to all other out-neighbors). Also, when processing a message (i.e., executing the rule enabled by the reception of that message), an entity can distinguish which of its in-neighbors sent that message.

In other words, each entity x has a local function λ_x associating labels, also called *port numbers*, to its incident links (or *ports*), and this function is injective. We denote port numbers by $\lambda_x(x, y)$, the label associated by x to the link (x, y). Let us stress that this label is local to x and in general has no relationship at all with what y might call this link (or x, or itself). Note that for each edge $(x, y) \in \vec{E}$, there are two labels: $\lambda_x(x, y)$ local to x and $\lambda_y(x, y)$ local to y (see Figure 1.1).

Because of this axiom, we will always deal with *edge-labeled graphs* $(\vec{G}, \lambda)$, where $\lambda = \{\lambda_x : x \in V\}$ is the set of these injective labelings.

FIGURE 1.1: Every edge has two labels

1.3.2 Restrictions

In general, a distributed computing system might have additional properties or capabilities that can be exploited to solve a problem, to achieve a task, and to provide a service. This can be achieved by using these properties and capabilities in the set of rules.

However, any property used in the protocol limits the applicability of the protocol. In other words, any additional property or capability of the system is actually a *restriction* (or submodel) of the general model.

WARNING. When dealing with (e.g., designing, developing, testing, employing) a distributed computing system or just a protocol, it is crucial and imperative that *all restrictions are made explicit*. Failure to do so will invalidate the resulting communication software.

The restrictions can be varied in nature and type: they might be related to communication properties, reliability, synchrony, and so forth. In the following section, we will discuss some of the most common restrictions.

Communication Restrictions The first category of restrictions includes those relating to communication among entities.

Queueing Policy A link (x, y) can be viewed as a channel or a queue (see Section 1.9): x sending a message to y is equivalent to x inserting the message in the channel. In general, all kinds of situations are possible; for example, messages in the channel might overtake each other, and a later message might be received first. Different restrictions on the model will describe different disciplines employed to manage the channel; for example, first-in-first-out (FIFO) queues are characterized by the following restriction.

- *Message Ordering:* In the absence of failure, the messages transmitted by an entity to the same out-neighbor will arrive in the same order they are sent.

Note that Message Ordering does not imply the existence of any ordering for messages transmitted to the same entity from different edges, nor for messages sent by the same entity on different edges.

Link Property Entities in a communication system are connected by physical links, which may be very different in capabilities. The examples are simplex and full-duplex links. With a fully duplex line it is possible to transmit in both directions. Simplex lines are already defined within the general model. A duplex line can obviously be described as two simplex lines, one in each direction; thus, a system where all lines are fully duplex can be described by the following restriction:

- *Reciprocal communication:* $\forall x \in \mathcal{E}$, $N_{\text{in}}(x) = N_{\text{out}}(x)$. In other words, if $(x, y) \in \vec{E}$ then also $(y, x) \in \vec{E}$.

Notice that, however, $(x, y) \neq (y, x)$, and in general $\lambda_x(x, y) \neq \lambda_x(y, x)$; furthermore, x might not know that these two links are connections to and from the same entity. A system with fully duplex links that offers such a knowledge is defined by the following restriction.

- *Bidirectional links:* $\forall x \in \mathcal{E}$, $N_{\text{in}}(x) = N_{\text{out}}(x)$ **and** $\lambda_x(x, y) = \lambda_x(y, x)$.

IMPORTANT. The case of Bidirectional Links is special. If it holds, we use a simplified terminology. The network is viewed as an *undirected* graph $G = (V,E)$ (i.e., $\forall\, x,y \in \mathcal{E}$, $(x,y) = (y, x)$), and the set $N(x) = N_{\text{in}}(x) = N_{\text{out}}(x)$ will just be called the set of *neighbors* of x. Note that in this case, $m(\vec{G}) = |\vec{E}| = 2\,|E| = 2\,m(G)$.

For example, in Figure 1.2 a graph $\vec{G}$ is depicted where the *Bidirectional Links* restriction and the corresponding undirected graph G hold.

Reliability Restrictions Other types of restrictions are those related to reliability, faults, and their detection.

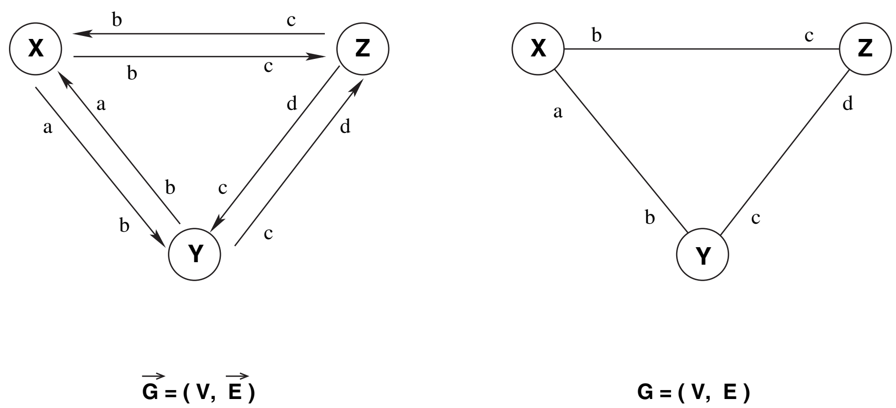

FIGURE 1.2: In a network with Bidirectional Links we consider the corresponding undirected graph.

Detection of Faults Some systems might provide a reliable fault-detection mechanism. Following are two restrictions that describe systems that offer such capabilities in regard to *component* failures:

- *Edge failure detection:* $\forall\ (x, y) \in \vec{E}$, both x and y will detect whether (x, y) has failed and, following its failure, whether it has been reactivated.
- *Entity failure detection:* $\forall x \in V$, all in- and out-neighbors of x can detect whether x has failed and, following its failure, whether it has recovered.

Restricted Types of Faults In some systems only some types of failures can occur: for example, messages can be lost but not corrupted. Each situation will give rise to a corresponding restriction. More general restrictions will describe systems or situations where there will be no failures:

- *Guaranteed delivery:* Any message that is sent will be received with its content uncorrupted.

Under this restriction, protocols do not need to take into account omissions or corruptions of messages during transmission. Even more general is the following:

- *Partial reliability:* No failures will occur.

Under this restriction, protocols do not need to take failures into account. Note that under Partial Reliability, failures might have occurred *before* the execution of a computation. A totally fault-free system is defined by the following restriction.

- *Total reliability:* Neither have any failures occurred nor will they occur.

Clearly, protocols developed under this restriction are *not* guaranteed to work correctly if faults occur.

Topological Restrictions In general, an entity is not directly connected to all other entities; it might still be able to communicate information to a remote entity, using others as relayer. A system that provides this capability for all entities is characterized by the following restriction:

- *Connectivity:* The communication topology $\vec{G}$ is strongly connected.

That is, from every vertex in $\vec{G}$ it is possible to reach every other vertex. In case the restriction “Bidirectional Links” holds as well, connectedness will simply state that G is connected.

Time Restrictions An interesting type of restrictions is the one relating to *time*. In fact, the general model makes no assumption about delays (except that they are finite).

- *Bounded communication delays:* There exists a constant Δ such that, in the absence of failures, the communication delay of any message on any link is at most Δ.

A special case of bounded delays is the following:

- *Unitary communication delays:* In the absence of failures, the communication delay of any message on any link is one unit of time.

The general model also makes no assumptions about the local clocks.

- *Synchronized clocks:* All local clocks are incremented by one unit simultaneously and the interval of time between successive increments is constant.

1.4 COST AND COMPLEXITY

The computing environment we are considering is defined at an abstract level. It models rather different systems (e.g., communication networks, distributed systems, data networks, etc.), whose performance is determined by very distinctive factors and costs.

The efficiency of a protocol in the model must somehow reflect the realistic costs encountered when executed in those very different systems. In other words, we need abstract cost measures that are general enough but still meaningful.

We will use two types of measures: the *amount of communication activities* and the *time* required by the execution of a computation. They can be seen as measuring costs from the system point of view (how much traffic will this computation generate and how busy will the system be?) and from the user point of view (how long will it take before I get the results of the computation?).

1.4.1 Amount of Communication Activities

The transmission of a message through an out-port (i.e., to an out-neighbor) is the basic *communication activity* in the system; note that the transmission of a message that will not be received because of failure still constitutes a communication activity. Thus, to measure the amount of communication activities, the most common function used is the number of message transmissions $\mathbf{M}$, also called *message cost*. So in general, given a protocol, we will measure its communication costs in terms of the number of transmitted messages.

Other functions of interest are the *entity workload* $\mathbf{L}_{\text{node}} = \mathbf{M}/|V|$, that is, the number of messages per entity, and the *transmission load* $\mathbf{L}_{\text{link}} = \mathbf{M}/|E|$, that is, the number of messages per link.

Messages are sequences of bits; some protocols might employ messages that are very short (e.g., O(1) bit signals), others very long (e.g., .gif files). Thus, for a more accurate assessment of a protocol, or to compare different solutions to the same problem that use different sizes of messages, it might be necessary to use as a cost measure the number of transmitted bits **B** also called *bit complexity*.

In this case, we may sometimes consider the bit-defined load functions: the *entity bit-workload* $\mathbf{Lb}_{\text{node}} = \mathbf{B}/|V|$, that is, the number of bits per entity, and the *transmission bit-load* $\mathbf{Lb}_{\text{link}} = \mathbf{B}/|E|$, that is, the number of bits per link.

1.4.2 Time

An important measure of efficiency and complexity is the total execution delay, that is, the delay between the time the first entity starts the execution of a computation and the time the last entity terminates its execution. Note that "time" is here intended as the one measured by an observer external to the system and will also be called real or physical time.

In the general model there is no assumption about time except that communication delays for a single message are finite in absence of failure (Axiom 1.3.1). In other words, communication delays are in general unpredictable. Thus, even in the absence of failures, the total execution delay for a computation is totally unpredictable; furthermore, two distinct executions of the same protocol might experience drastically different delays. In other words, we cannot accurately measure time.

We, however, can measure time assuming particular conditions. The measure usually employed is the *ideal execution delay* or *ideal time complexity*, **T**: the execution delay experienced under the restrictions "Unitary Transmission Delays" and "Synchronized Clocks;" that is, when the system is synchronous and (in the absence of failure) takes one unit of time for a message to arrive and to be processed.

A very different cost measure is the *causal time complexity*, $\mathbf{T}_{\text{causal}}$. It is defined as the length of the longest chain of causally related message transmissions, over all possible executions. Causal time is seldom used and is very difficult to measure exactly; we will employ it only once, when dealing with synchronous computations.

1.5 AN EXAMPLE: BROADCASTING

Let us clarify the concepts expressed so far by means of an example. Consider a distributed computing system where one entity has some important information unknown to the others and would like to share it with everybody else.

This problem is called *broadcasting* and it is part of a general class of problems called *information diffusion*. To solve this problem means to design a set of rules that, when executed by the entities, will lead (within finite time) to all entities knowing the information; the solution must work regardless of which entity had the information at the beginning.

Let $\mathcal{E}$ be the collection of entities and $\vec{G}$ be the communication topology.

To simplify the discussion, we will make some additional assumptions (i.e., restrictions) on the system:

1. Bidirectional links; that is, we consider the undirected graph G. (see Section 1.3.2).
2. Total reliability, that is, we do not have to worry about failures.

Observe that, if G is disconnected, some entities can never receive the information, and the broadcasting problem will be unsolvable. Thus, a restriction that (unlike the previous two) we *need* to make is as follows:

3. Connectivity; that is, G is connected.

Further observe that built in the definition of the problem, there is the assumption that only the entity with the initial information will start the broadcast. Thus, a restriction built in the definition is as follows:

4. Unique Initiator, that is, only one entity will start.

A simple strategy for solving the broadcast problem is the following:

"if an entity knows the information, it will share it with its neighbors."

To construct the set of rules implementing this strategy, we need to define the set $\mathcal{S}$ of status values; from the statement of the problem it is clear that we need to distinguish between the entity that initially has the information and the others: $\{initiator, idle\} \subseteq \mathcal{S}$. The process can be started only by the *initiator*; let I denote the information to be broadcasted. Here is the set of rules $B(x)$ (the same for all entities):

1. *initiator* $\times \iota \longrightarrow$ {**send**(I) **to** $N(x)$}
2. *idle* $\times$ *Receiving*(I) $\longrightarrow$ {Process(I); **send**(I) **to** $N(x)$}
3. *initiator* $\times$ *Receiving*(I) $\longrightarrow$ **nil**
4. *idle* $\times \iota \longrightarrow$ **nil**

where ι denotes the *spontaneous impulse* event and **nil** denotes the *null* action.

Because of connectivity and total reliability, every entity will eventually receive the information. Hence, the protocol achieves its goal and solves the broadcasting problem.

However, there is a serious problem with these rules:

the activities generated by the protocol never terminate.

Consider, for example, the simple system with three entities x, y, z connected to each other (see Figure 1.3). Let x be the *initiator*, y and z be *idle*, and all messages travel at the same speed; then y and z will be forever sending messages to each other (as well as to x).

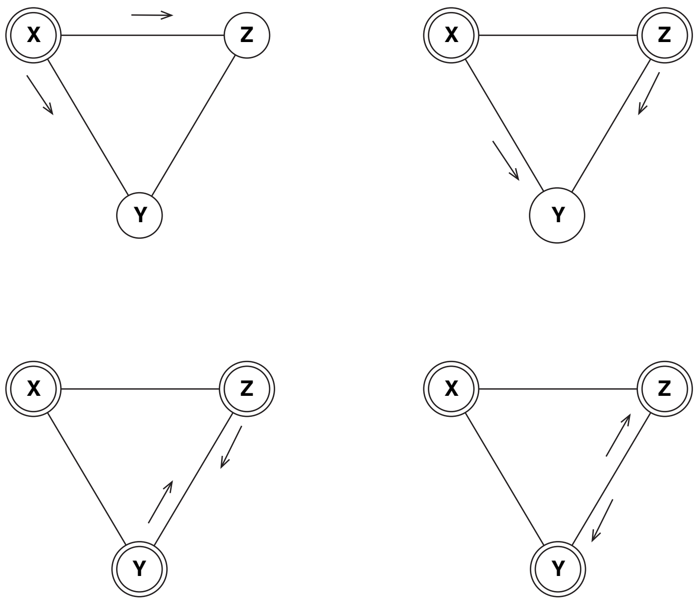

FIGURE 1.3: An execution of Flooding.

To avoid this unwelcome effect, an entity should send the information to its neighbors only once: the first time it acquires the information. This can be achieved by introducing a new status *done*; that is $\mathcal{S}$ ={*initiator*, *idle*, *done*}.

1. *initiator* $\times\iota \longrightarrow$ {**send**(I) **to** $N(x)$; **become** done}
2. *idle* $\times$ *Receiving*(I) $\longrightarrow$ {Process(*I*); **become** *done*; **send**(*I*) **to** $N(x)$}
3. *initiator* $\times$ *Receiving*(I) $\longrightarrow$ **nil**
4. *idle* $\times\ \iota \longrightarrow$ **nil**
5. *done* $\times$ *Receiving*(*I*) $\longrightarrow$ **nil**
6. *done* $\times\ \iota \longrightarrow$ **nil**

where **become** denotes the operation of changing status.

This time the communication activities of the protocol terminate: Within finite time all entities become *done*; since a *done* entity knows the information, the protocol is correct (see Exercise 1.12.1). Note that depending on transmission delays, different executions are possible; one such execution in an environment composed of three entities x, y, z connected to each other, where x is the initiator as depicted in Figure 1.3.

IMPORTANT. Note that entities terminate their execution of the protocol (i.e., become *done*) at different times; it is actually possible that an entity has terminated while others have not yet started. This is something very typical of distributed computations: There is a difference between *local termination* and *global termination*.

IMPORTANT. Notice also that in this protocol nobody ever knows when the entire process is over. We will examine these issues in details in other chapters, in particular when discussing the problem of *termination detection*.

The above set of rules correctly solves the problem of broadcasting. Let us now calculate the communication costs of the algorithm.

First of all, let us determine the number of *message transmissions*. Each entity, whether *initiator* or not, sends the information to all its neighbors. Hence the total number of messages transmitted is exactly

$$\sum_{x \in \mathcal{E}} |N(x)| = 2\,|E| = 2\,m.$$

We can actually reduce the cost. Currently, when an *idle* entity receives the message, it will broadcast the information to *all* its neighbors, including the entity from which it had received the information; this is clearly unnecessary. Recall that, by the Local Orientation axiom, an entity can distinguish among its neighbors; in particular, when processing a message, it can identify from which port it was received and avoid sending a message there. The final protocol is as before with only this small modification.

Protocol *Flooding*

1. $\mathit{initiator} \times \iota \longrightarrow$ {**send**(I) **to** $N(x)$; **become** *done*}
2. $\mathit{idle} \times \mathit{Receiving}$(I) $\longrightarrow$ {Process(I); **become** *done*; **send**(I) **to** $N(x)$-**sender**}
3. $\mathit{initiator} \times \mathit{Receiving}$(I) $\longrightarrow$ **nil**
4. $\mathit{idle} \times \iota \longrightarrow$ **nil**
5. $\mathit{done} \times \mathit{Receiving}$(I) $\longrightarrow$ **nil**
6. $\mathit{done} \times \iota \longrightarrow$ **nil**

where **sender** is the neighbor that sent the message currently being processed.

This algorithm is called *Flooding* as the entire system is "flooded" with the message during its execution, and it is a basic algorithmic tool for distributed computing. As for the number of message transmissions required by flooding, because we avoid transmitting some messages, we know that it is less than $2m$; in fact, (Exercise 1.12.2):

$$M[\mathit{Flooding}] = 2m - n + 1. \tag{1.1}$$

Let us examine now the ideal time complexity of flooding.

Let $d(x, y)$ denote the distance (i.e., the length of the shortest path) between x and y in G. Clearly the message sent by the initiator has to reach every entity in the system, including the furthermost one from the *initiator*. So, if x is the initiator, the ideal time complexity will be $r(x) = \text{Max}\ \{d(x, y) : y \in \mathcal{E}\}$, which is called the *eccentricity* (or *radius*) of x. In other words, the total time depends on which entity is the initiator and thus cannot be known precisely beforehand. We can, however, determine exactly the ideal time complexity in the worst case.

Since any entity could be the initiator, the ideal time complexity in the worst case will be $d(G) = \text{Max}\ \{r(x) : x \in \mathcal{E}\}$, which is the *diameter* of G. In other words, the ideal time complexity will be at most the diameter of G:

$$\mathbf{T}[Flooding] \leq d(G). \tag{1.2}$$

1.6 STATES AND EVENTS

Once we have defined the behavior of the entities, their communication topology, and the set of restrictions under which they operate, we must describe the initial conditions of our environment. This is done first of all by specifying the initial condition of all the entities. The initial content of all the registers of entity x and the initial value of its alarm clock c_x at time t constitute the *initial internal state* $\sigma(x, 0)$ of x. Let $\sum(0) = \{\sigma(x, 0) : x \in \mathcal{E}\}$ denote the set of all the initial internal states.

Once $\sum(0)$ is defined, we have completed the *static* specification of the environment: the description of the system *before* any event occurs and before any activity takes place.

We are, however, also interested in describing the system *during* the computational activities, as well as *after* such activities. To do so, we need to be able to describe the changes that the system undergoes over time. As mentioned before, the entities (and, thus the environments) are *reactive*. That is, any activity of the system is determined entirely by the external events. Let us examine these facts in more detail.

1.6.1 Time and Events

In distributed computing environments, there are only three types of external events: spontaneous impulse (*spontaneously*), reception of a message (*receiving*), and alarm clock ring (*when*).

When an external event occurs at an entity, it triggers the execution of an action (the nature of the action depends on the status of the entity when the event occurs). The executed action may generate new events: The operation **send** will generate a *receiving* event, and the operation **set_alarm** will generate a *when* event.

Note first of all that the events so generated might not occur at all. For example, a link failure may destroy the traveling message, destroying the corresponding *receiving* event; in a subsequent action, an entity may turn off the previously set alarm destroying the *when* event.

Notice now that if they occur, these events will do so at a later time (i.e., when the message arrives, when the alarm goes off). This delay might be known precisely in the case of the alarm clock (because it is set by the entity); it is, however, unpredictable in the case of message transmission (because it is due to the conditions external to the entity). Different delays give rise to different *executions* of the same protocols with possibly different outcomes.

Summarizing, each event e is "generated" at some time $t(e)$ and, if it occurs, it will happen at some time later.

By definition, all spontaneous impulses are already generated before the execution starts; their set will be called the set of *initial events*. The execution of the protocol starts when the first spontaneous impulses actually happen; by convention, this will be time $t = 0$.

IMPORTANT. Notice that "time" is here considered as seen by an external observer and is viewed as *real time*. Each real time instant t separates the axis of time into three parts: *past* (i.e., $\{t' < t\}$), *present* (i.e., t), and *future* (i.e., $\{t' > t\}$). All events generated before t that will happen after t are called the *future at t* and denoted by *Future(t)*; it represents the set of future events determined by the execution so far.

An execution is fully described by the sequence of events that have occurred. For small systems, an execution can be visualized by what is called a *Time × Event Diagram* (TED) . Such a diagram is composed of temporal lines, one for each entity in the system. Each event is represented in such a diagram as follows:

A *Receiving* event r is represented as an arrow from the point $t_x(r)$ in the temporal line of the entity x generating e (i.e., sending the message) to the point $t_y(r)$ in the temporal line of the entity y where the events occur (i.e., receiving the message).

A *When* event w is represented as an arrow from point $t'_x(w)$ to point $t''_x(w)$ in the temporal line of the entity setting the clock.

A *Spontaneously* event ι is represented as a short arrow indicating point $t_x(\iota)$ in the temporal line of the entity x where the events occur.

For example, in Figure 1.4 is depicted the TED corresponding to the execution of Protocol *Flooding* of Figure 1.3.

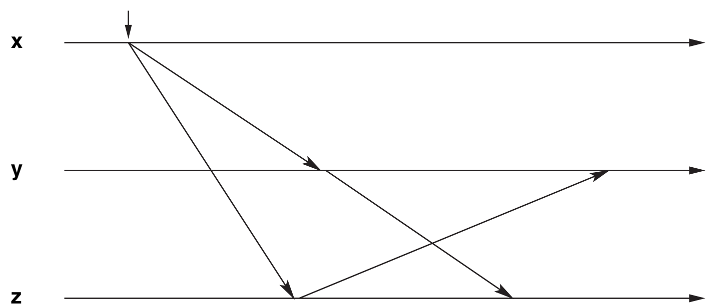

FIGURE 1.4: Time × Event Diagram

1.6.2 States and Configurations

The private memory of each entity, in addition to the behavior, contains a set of registers, some of them already initialized, others to be initialized during the execution. The content of all the registers of entity x and the value of its alarm clock c_x at time t constitute what is called the *internal state of x at t* and is denoted by $\sigma(x, t)$. We denote by $\Sigma(t)$ the set of the internal states at time t of all entities. Internal states change with time and the occurrence of events.

There is an important fact about internal states. Consider two different environments, E_1 and E_2, where, by accident, the internal state of x at time t is the same. Then x *cannot distinguish* between the two environments, that is, x is unable to tell whether it is in environment E_1 or E_2.

There is an important consequence. Consider the situation just described: At time t, the internal state of x is the same in both E_1 and E_2. Assume now that also by accident, exactly the same event occurs at x (e.g., the alarm clock rings or the same message is received from the same neighbor). Then x will perform exactly the same action in both cases, and its internal state will continue to be the same in both situations.

Property 1.6.1 *Let the same event occur at x at time t in two different executions, and let σ_1 and σ_2 be its internal states when this happens. If $\sigma_1 = \sigma_2$, then the new internal state of x will be the same in both executions.*

Similarly, if two entities have the same internal state, they *cannot distinguish* between each other. Furthermore, if by accident, exactly the same event occurs at both of them (e.g., the alarm clock rings or the same message is received from the same neighbor), then they will perform exactly the same action in both cases, and their internal state will continue to be the same in both situations.

Property 1.6.2 *Let the same event occur at x and y at time t, and let σ_1 and σ_2 be their internal states, respectively, at that time. If $\sigma_1 = \sigma_2$, then the new internal state of x and y will be the same.*

Remember: Internal states are local and an entity might not be able to infer from them information about the status of the rest of the system. We have talked about the internal state of an entity, initially (i.e., at time $t = 0$) and during an execution. Let us now focus on the state of the entire system during an execution.

To describe the *global* state of the environment at time t, we obviously need to specify the internal state of all entities at that time; that is, the set $\Sigma(t)$. However, this is *not enough*. In fact, the execution so far might have already generated some events that will occur *after* time t; these events, represented by the set *Future*(t), are integral part of this execution and must be specified as well. Specifically, the global state, called *configuration*, of the system during an execution is specified by the couple

$$\mathcal{C}(t) = (\Sigma(t), \mathit{Future}(t))$$

The *initial* configuration $C(0)$ contains not only the initial set of states $\Sigma(0)$ but also the set *Future*(0) of the spontaneous impulses. Environments that differ only in their initial configuration will be called *instances* of the same system.

The configuration $C(t)$ is like a snapshot of the system at time t.

1.7 PROBLEMS AND SOLUTIONS (⋆)

The topic of this book is how to design distributed algorithms and analyze their complexity. A distributed algorithm is the set of rules that will regulate the *behaviors* of the entities. The reason why we may need to design the *behaviors* is to enable the entities to solve a given problem, perform a defined task, or provide a requested service.

In general, we will be given a problem, and our task is to design a set of rules that will always solve the problem in finite time. Let us discuss these concepts in some details.

Problems To give a problem (or task, or service) $\mathcal{P}$ means to give a description of *what* the entities must accomplish. This is done by stating what the initial conditions of the entities are (and thus of the system), and what the final conditions should be; it should also specify all given restrictions. In other words,

$$\mathcal{P} = \langle P_{\text{INIT}}, P_{\text{FINAL}}, R \rangle,$$

where P_{INIT} and P_{FINAL} are *predicates* on the values of the registers of the entities, and R is a set of restrictions. Let $w_t(x)$ denote the value of an input register $w(x)$ at time t and $\{w_t\} = \{w_t(x) : x \in \mathcal{E}\}$ the values of this register at all entities at that time. So, for example, $\{status_0\}$ represents the initial value of the status registers of the entities.

For example, in the problem *Broadcasting* (I) described in Section 1.5, the initial and final conditions are given by the predicates

$P_{\text{INIT}}(t) \equiv$ " *only one entity has the information at time t*" $\equiv$
$\quad \exists x \in \mathcal{E}\ \ (value_t(x) = I\ \ \wedge \forall y \neq x\ (value_t(y) = \emptyset)),$

$P_{\text{FINAL}}(t) \equiv$ " *every entity has the information at time t*" $\equiv$
$\quad \forall x \in \mathcal{E}\ (value_t(x) = I).$

The restrictions we have imposed on our solution are BL (Bidirectional Links), TR (Total Reliability), and CN (Connectivity). Implicit in the problem definition there is also the condition that only the entity with the information will start the execution of the solution protocol; denote by UI the predicate describing this restriction, called *Unique Initiator*. Summarizing, for *Broadcasting*, the set of restrictions we have made is {BL, TR, CN, UI}.

Status A solution protocol B for $\mathcal{P} = \langle P_{\text{INIT}}, P_{\text{FINAL}}, R\rangle$ will specify *how* the entities will accomplish the required task. Part of the design of the set of rules $B(x)$ is the definition of the set of status values $\mathcal{S}$, that is, the values that can be held by the status register *status(x)*.

We call *initial* status values those values of $\mathcal{S}$ that can be held at the start of the execution of *B(x)* and we shall denote their set by $\mathcal{S}_{\text{INIT}}$. By contrast, *terminal* status values are those values that once reached, cannot ever be changed by the protocol; their set shall be denoted by $\mathcal{S}_{\text{TERM}}$. All other values in $\mathcal{S}$ will be called *intermediate* status values.

For example, in the protocol *Flooding* described in Section 1.5, $\mathcal{S}_{\text{INIT}}$={*initiator*, *idle*} and $\mathcal{S}_{\text{TERM}}$={*done*}.

Depending on the restrictions of the problem, only entities in specific initial status values will start the protocol; we shall denote by $\mathcal{S}_{\text{START}} \subseteq \mathcal{S}_{\text{INIT}}$ the set of those status values. Typically, $\mathcal{S}_{\text{START}}$ consists of only one status; for example, in *Flooding*, $\mathcal{S}_{\text{START}}$={*initiator*}. It is possible to rewrite a protocol so that this is always the case (see Exercise 1.12.5).

Among terminal status values we shall distinguish those in which no further activity can take place; that is, those where the only action is **nil**. We shall call such status values *final* and we shall denote by $\mathcal{S}_{\text{FINAL}} \subseteq \mathcal{S}_{\text{TERM}}$ the set of those status values. For example, in *Flooding*, $\mathcal{S}_{\text{FINAL}}$={*done*}.

Termination Protocol B terminates if, for all initial configurations $C(0)$ satisfying P_{INIT}, and for all executions starting from those configurations, the predicate

$$\textit{Terminate}\,(t) \equiv (\{status_t\} \subseteq \mathcal{S}_{\text{TERM}}) \wedge (\textit{Future}(t) = \emptyset)$$

holds for some $t > 0$, that is, all entities enter a terminal status after a finite time and all generated events have occurred.

We have already remarked on the fact that entities might not be aware that the termination has occurred. In general, we would like each entity to know at least of its termination. This situation, called *explicit termination*, is said to occur if the predicate

$$\textit{Explicit-Terminate}\,(t) \equiv (\{status_t\} \subseteq \mathcal{S}_{\text{FINAL}})$$

holds for some $t > 0$, that is, all entities enter a final status after a finite time.

Correctness Protocol B is correct if, for all executions starting from initial configurations satisfying P_{INIT},

$$\exists t > 0 : \textit{Correct}(t)$$

holds, where $\textit{Correct}(t) \equiv (\forall t' \geq t,\ P_{\text{FINAL}}(t))$; that is, the final predicate eventually holds and does not change.

Solution Protocol The set of rules B solves problem $\mathcal{P}$ if it always correctly terminates under the problem restrictions R. As there are two types of termination (simple and explicit), we will have two types of solutions:

Simple Solution$[B,\mathcal{P}]$ where the predicate

$$\exists t > 0\ (Correct(t) \wedge\ Terminate(t))$$

holds, under the problem restrictions R, for all executions starting from initial configurations satisfying P_{INIT}; and

Explicit Solution$[B,\mathcal{P}]$ where the predicate

$$\exists t > 0\ (Correct(t) \wedge\ Explicit\text{-}Terminate(t))$$

holds, under the problem restrictions R, for all executions starting from initial configurations satisfying P_{INIT}.

1.8 KNOWLEDGE

The notions of information and knowledge are fundamental in distributed computing. Informally, any distributed computation can be viewed as the process of acquiring information through communication activities; conversely, the reception of a message can be viewed as the process of transforming the state of knowledge of the processor receiving the message.

1.8.1 Levels of Knowledge

The content of the local memory of an entity and the information that can be derived from it constitute the *local knowledge* of an entity. We denote by

$$p \in \mathrm{LK}_t[x]$$

the fact that p is local knowledge at x at the global time instant t. By definition, $\lambda_x \in \mathrm{LK}_t[x]$ for all t, that is, the (labels of the) in- and out-edges of x are time-invariant local knowledge of x.

Sometimes it is necessary to describe knowledge held by more than one entity at a given time. Information p is said to be *implicit knowledge* in $W \subseteq \mathcal{E}$ at time t, denoted by $p \in \mathrm{IK}_t[W]$, if at least one entity in W knows p at time t, that is,

$$p \in \mathrm{IK}_t[W] \text{ iff } \exists x \in W\ (p \in \mathrm{LK}_t[x]).$$

A stronger level of knowledge in a group W of entities is held when, at a given time t, p is known to every entity in the group, denoted by $p \in \mathrm{EK}_t[W]$, that is

$$p \in \mathrm{EK}_t[W] \text{ iff } \forall x \in W\ (p \in \mathrm{LK}_t[x]).$$

In this case, p is said to be *explicit knowledge* in $W \subseteq \mathcal{E}$ at time t.

Consider for example *broadcasting* discussed in the previous section. Initially, at time $t = 0$, only the initiator s knows the information I; in other words, $I \in \mathrm{LK}_0[s]$. Thus, at that time, I is implicitly known to all entities, that is, $I \in \mathrm{IK}_0[\mathcal{E}]$. At the end of the broadcast, at time t', every entity will know the information; in other words, $I \in \mathrm{EK}_{t'}[\mathcal{E}]$.

Notice that, in the absence of failures, knowledge cannot be lost, only gained, that is, for all $t' > t$ and all $W \subseteq \mathcal{E}$, if no failure occurs, $\mathrm{IK}_t[W] \subseteq \mathrm{IK}_{t'}[W]$ and $\mathrm{EK}_t[W] \subseteq \mathrm{EK}_{t'}[W]$.

Assume that a fact p is explicit knowledge in W at time t. It is possible that some (maybe all) entities are not aware of this situation. For example, assume that at time t, entities x and y know the value of a variable of z, say its ID; then the ID of z is explicit knowledge in W={x, y, z}; however, z might not be aware that x and y know its ID. In other words, when $p \in \mathrm{EK}_t[W]$, the fact "$p \in \mathrm{EK}_t[W]$" might not be even locally known to any of the entities in W.

This gives rise to the highest level of knowledge within a group: common knowledge. Information p is said to be *common knowledge* in $W \subseteq \mathcal{E}$ at time t , denoted by $p \in \mathrm{CK}_t[W]$, if and only if at time t every entity in W knows p, and knows that every entity in W knows p, and knows that entity in W knows that every entity in W knows p, and ..., etcetera, that is,

$$p \in \mathrm{CK}_t[W] \text{ iff } \bigwedge_{1 \le i \le \infty} P_i,$$

where the P_i's are the predicates defined by: $P_1 = [p \in \mathrm{ES}_t[W]]$ and $P_{i+1} = [P_i \in \mathrm{EK}_t[W]]$.

In most distributed problems, it will be necessary for the entities to achieve common knowledge. Fortunately, we do not always have to go to ∞ to reach common knowledge, and a finite number of steps might actually do, as indicated by the following example.

Example (muddy forehead): Imagine n perceptive and intelligent school children playing together during recess. They are forbidden to play in the mud puddles, and the teacher has told them that if they do, there will be severe consequences. Each child wants to keep clean, but the temptation to play with mud is too great to resist. As a result, k of the children get mud on their foreheads. When the teacher arrives, she says, "I see that some of you have been playing in the mud puddle: the mud on your foreheads is a dead giveaway !" and then continues, "The guilty ones who come forward spontaneously will be given a small penalty; those who do not, will receive a punishment they will not easily forget." She then adds, "I am going to leave the room now, and I will return periodically; if you decide to confess, you must all come forward together when I am in the room. In the meanwhile, everybody must sit absolutely still and without talking."

Each child in the room clearly understands that those with mud on their foreheads are "dead meat," who will be punished no matter what. Obviously, the children do not want to confess if the foreheads are clean, and clearly, if the foreheads are dirty, they want to go forward so as to avoid their terrible punishment for those who do not confess. As each child shares the same concern, the collective goal is for the children with clean foreheads not to confess and for those with muddy foreheads to go forward simultaneously, and all of this without communication.

Let us examine this goal. The first question is as follows: can a child x find out whether his/her forehead is dirty or not ? She/he can see how many, say f_x, of the other children are dirty; thus, the question is if x can determine whether $k = f_x$ or $k = f_x + 1$.

The second, more complex question is as follows: can *all* the children with mud on their foreheads find out at the same time so that they can go forward together ? In other words, can the exact value of k become *common knowledge* ?

The children, being perceptive and intelligent, determine that the answer to both the questions is positive and find the way to achieve the common goal and thus common knowledge without communication (Exercise 1.12.6).

IMPORTANT. When working in a submodel, all the restrictions defining the submodel are common knowledge to all entities (unless otherwise specified).

1.8.2 Types of Knowledge

We can have various types of knowledge, such as knowledge about the communication topology, about the labeling of the communication graph, about the input data of the communicating entities. In general, if we have some knowledge of the system, we can exploit it to reduce the cost of a protocol, although this may result in making the applicability of the protocol more limited.

A type of knowledge of particular interest is the one regarding the communication topology (i.e., the graph $\vec{G}$). In fact, as will be seen later, the complexity of a computation may vary greatly depending on what the entities know about $\vec{G}$. Following are some elements that, if they are common knowledge to the entities, may affect the complexity.

1. *Metric Information*: numeric information about the network; for example, number $n = |V|$ of nodes, number $m = |E|$ of links, diameter, girth, etcetera. This information can be *exact* or *approximate*.
2. *Topological Properties*: knowledge of some properties of the topology; for example, "$\vec{G}$ is a ring network," "$\vec{G}$ does not have cycles," "$\vec{G}$ is a Cayley graph," etcetera.
3. *Topological Maps*: a map of the neighborhood of the entity up to distance d, a complete "map" of $\vec{G}$ (e.g., the adjacency matrix of $\vec{G}$); a complete "map" of $(\vec{G},\lambda)$ (i.e., it contains also the labels), etcetera.

Note that some types of knowledge imply other knowledge; for example, if an entity with k neighbors knows that the network is a complete undirected graph, then it knows that $n = k+1$.

As a topological map provides all possible metric and structural information, this type of knowledge is very powerful and important. The strongest form of this type is *full topological knowledge*: availability at each entity of a labeled graph isomorphic to $(\vec{G},\lambda)$, the isomorphism, and its own image, that is, every entity has a complete map of (v, λ) with the indication, "You are here."

Another type of knowledge refers to the labeling λ. What is very important is whether the labeling has some global consistency property.

We can distinguish two other types, depending on whether the knowledge is about the (input) data or the status of the entities and of the system, and we shall call them type-D and type-S, respectively.

Examples of type-D knowledge are the following: *Unique identifiers*: all input values are distinct; *Multiset*: input values are not necessarily identical; *Size*: number of distinct values.

Examples of type-S knowledge are the following: *System with leader*: there is a unique entity in status "leader"; *Reset*: all nodes are in the same status; *Unique initiator*: there is a unique entity in status "initiator." For example, in the broadcasting problem we discussed in Section 1.5, this knowledge was assumed as a part of the problem definition.

1.9 TECHNICAL CONSIDERATIONS

1.9.1 Messages

The content of a message obviously depends on the application; in any case, it consists of a finite (usually bounded) sequence of bits.

The message is typically divided into subsequences, called *fields*, with a predefined meaning ("type") within the protocol.

The examples of field types are the following: *message identifier* or *header* used to distinguish between different types of messages; *originator* and *destination* fields used to specify the (identity of the) entity originating this message and of the entity to whom the message is intended for; *data* fields used to carry information needed in the computation (the nature of the information obviously depends on the particular application under consideration).

Thus, in general, a message M will be viewed as a tuple $M = \langle f_1, f_2, \ldots f_k \rangle$ where k is a (small) predefined constant, and each f_i $(1 \leq i \leq k)$ is a field of a specified type, each type of a fixed length.

So, for example, in protocol *Flooding*, there is only one type of message; it is composed of two fields $M = \langle f_1, f_2 \rangle$ where f_1 is a message identifier (containing the information: "this is a broadcast message"), and f_2 is a data field containing the actual information I being broadcasted.

If (the limit on) the size of a message is a system parameter (i.e., it does not depend on the particular application), we say that the system has *bounded messages*. Such is, for example, the limit imposed on the message length in packet-switching networks, as well as on the length of control messages in circuit-switching networks (e.g., telephone networks) and in message-switching networks.

Bounded messages are also called *packets* and contain at most $\mu(G)$ bits, where $\mu(G)$ is the system-dependent bound called *packet size*. Notice that, to send a sequence of K bits in G will require the transmission of at least $\lceil K/\mu(G) \rceil$ packets.

1.9.2 Protocol

Notation A protocol $B(x)$ is a set of rules. We have already introduced in Section 1.5 most of the notation for describing those rules. Let us now complete the description of the notation we will use for protocols. We will employ the following conventions:

1. Rules will be grouped by *status*.
2. If the action for a (*status*,*event*) pair is **nil**, then, for simplicity, the corresponding rule will be *omitted* from the description. As a consequence, if no rule is described for a (*status*,*event*) pair, the *default* will be that the pair enables the Null action.

WARNING. Although convenient (it simplifies the writing), the use of this convention must generate extra care in the description: If we forget to write a rule for an event occurring in a given status, it will be assumed that a rule exists and the action is nil.

3. If an action contains a change of status, this operation will be the last one before exiting the action.
4. The set of status values of the protocol, and the set of restrictions under which the protocol operates will be explicit.

Using these conventions, the protocol Flooding defined in Section 1.5 will be written as shown in Figure 1.5.

Precedence The external events are as follows: spontaneous impulse (*Spontaneously*), reception of a message (*Receiving*), and alarm clock ring (*When*). Different types of external events can occur simultaneously; for example, the alarm clock might ring at the same time a message arrives. The simultaneous events will be processed sequentially. To determine the order in which they will be processed, we will use the following *precedence* between external events:

Spontaneously > *When* > *Receiving*;

that is, the spontaneous impulse takes precedence over the alarm clock, which has precedence over the arrival of a message.

At most one spontaneous impulse can always occur at an entity at any one time. As there is locally only one alarm clock, at any time there will be at most one *When* event. By contrast, it is possible that more than one message arrive at the same time to an entity from different neighbors; should this be the case, these simultaneous

PROTOCOL Flooding .

- Status Values: $\mathcal{S}$ = {INITIATOR, IDLE, DONE};
  $\mathcal{S}_{\text{INIT}}$ = {INITIATOR, IDLE};
  $\mathcal{S}_{\text{TERM}}$ = {DONE}.
- Restrictions: Bidirectional Links, Total Reliability, Connectivity, and Unique Initiator.

```
INITIATOR
      Spontaneously
      begin
            send(M) to N(x);
            become DONE;
      end

IDLE
      Receiving(I)
      begin
            Process(M);
            send(M) to N(x) − {sender};
            become DONE;
      end
```

FIGURE 1.5: Flooding Protocol

Receiving events have all the same precedence and will be processed sequentially in an arbitrary order.

1.9.3 Communication Mechanism

The communication mechanisms of a distributed computing environment must handle transmissions and arrivals of messages. The mechanisms at an entity can be seen as a system of queues.

Each link $(x, y) \in \vec{E}$ corresponds to a queue, with access at x and exit at y; the access is called *out-port* and the exit is called *in-port*.

Each entity has thus two types of ports: *out-ports*, one for each out-neighbor (or out-link), and *in-port*, one for each in-neighbor (or in-link). At an entity, each out-port has a distinct label (recall the Local Orientation axiom (Axiom 1.3.2)) called port number: the out-port corresponding to (x, y) has label $\lambda_x(x, y)$; similarly for the in-ports.

The sets N_{in} and N_{out} will in practice consist of the port numbers associated to those neighbors; this is because an entity has no other information about its neighbors (unless we add restrictions).

The command "**send** M **to** W" will have a copy of the message M sent through each of the out-ports specified by W.

When a message M is sent through an out-port l, it is inserted in the corresponding queue. In absence of failures (recall the Finite Communication Delays axiom), the communication mechanism will eventually remove it from the queue and deliver it to the other entity through the corresponding in-port, generating the *Receiving* (M) event; at that time the variable **sender** will be set to l.

1.10 SUMMARY OF DEFINITIONS

Distributed Environment: Collection of communicating computational entities.

Communication: Transmission of message.

Message: Bounded sequence of bits.

Entity's Capability: Local processing, local storage, access to a local clock, and communication.

Entity's Status Register: At any time an entity status register has a value from a predefined set of status values.

External Events: Arrival of a message, alarm clock ring, and spontaneous impulse.

Entity's Behavior: Entities react to external events. The *behavior* is dictated by a set of rules. Each rule has the form

$$STATUS \times EVENT \rightarrow Action$$

specifying what the entity has to do if a certain external event occurs when the entity is in a given status. The set of rules must be nonambiguous and complete.

Actions: An action is an indivisible (i.e., uninterruptible) finite sequence of operations (local processing, message transmission, change of status, and setting of alarm clock).

Homogeneous System: A system is homogeneous if all the entities have the same behavior. Every system can be made homogeneous.

Neighbors: The in-neighbors of an entity are those entities from which x can receive a message directly; the out-neighbors are those to which x can send a message directly.

Communication Topology: The directed graph $G = (V, E)$ defined by the neighborhood relation. If the Bidirectional Links restriction holds, then G is undirected.

Axioms: There are two axioms: local orientation and finite communication delays.

Local Orientation: An entity can distinguish between its out-neighbors and its in-neighbors.

Finite Communication Delays: In absence of failures, a message eventually arrives.

Restriction: Any additional property.

1.11 BIBLIOGRAPHICAL NOTES

Several attempts have been made to derive formalisms capable of describing both distributed systems and computations performed in such systems. A significant amount of study has been devoted to defining formalisms, which would ease the task of formally proving properties of distributed computation (e.g., absence of deadlock, liveness, etc.). The models proposed for systems of concurrent processes do provide both a formalism for describing a distributed computation and a proof system that can be employed within the formalism; such is, for example, the *Unity* model of Mani Chandi and Jayadev Misra [1]. Other models, whose intended goal is still to provide a proof system, have been specifically tailored for distributed computations. In particular, the *Input–Output Automata* model of Nancy Lynch and Mark Tuttle [4] provides a powerful tool that has helped discover and fix "bugs" in well-known existing protocols.

For the investigators involved in the design and analysis of distributed algorithms, the main concern rests with efficiency and complexity; proving correctness of an algorithm is a compulsory task, but it is usually accomplished using traditional mathematical tools (which are generally considered informal techniques) rather than with formal proof systems. The formal models of computation employed in these studies, as well as in the one used in this book, mainly focus on those factors that are directly related to efficiency of a distributed computation and complexity of a distributed problem: the underlining communication network, the communication primitives, the amount and type of knowledge available to the processors, etcetera.

Modal logic, and in particular the notion of common knowledge, is a useful tool to reason about distributed computing environments in presence of failures. The notion of knowledge used here was developed independently by Joseph Halpern and Yoram Moses [2], Daniel J. Lehmann [3], and Stanley Rosenschein [5].

The model we have described and will employ in this book uses *reactive* entities (they react to external stimuli). Several formal models (including *input–output Automata*) use instead *active* entities. To understand this fundamental difference, consider a message in transit toward an entity that is expecting it, with no other activity in the system. In an active model, the entity will attempt to receive the message, even while it is not there; each attempt is an event; hence, this simple situation can actually cause an unpredictable number of events. By contrast, in a reactive model, the entity does nothing; the only event is the arrival of the message that will "wake up" the entity and trigger its response.

Using the analogy of waiting for the delivery of a pizza, in the *active* model, you (the entity) must repeatedly open the door (i.e., act) to see if the person supposed to deliver the pizza has arrived; in the *reactive* model, you sit in the living room until the bell rings and then go and open the door (i.e., react).

The two models are equally powerful; they just represent different ways of looking at and expressing the world. It is our contention that at least for the description and the complexity analysis of protocols and distributed algorithms, the reactive model is more expressive and simpler to understand, to handle, and to use.

1.12 EXERCISES, PROBLEMS, AND ANSWERS

1.12.1 Exercises and Problems

Exercise 1.12.1 Prove that the flooding technique introduced in Section 1.5 is correct, that is, it terminates within finite time, and all entities will receive the information held by the initiator.

Exercise 1.12.2 Determine the exact number of message transmissions required by the protocol *Flooding* described in Section 1.5.

Exercise 1.12.3 In Section 1.5 we have solved the broadcasting problem under the restriction of Bidirectional Links. Solve the problem using the Reciprocal Communication restriction instead.

Exercise 1.12.4 In Section 1.5 we have solved the broadcasting problem under the restriction of Bidirectional Links. Solve the problem *without* this restriction.

Exercise 1.12.5 Show that any protocol B can be rewritten so that $\mathcal{S}_{\text{START}}$ consists of only one status. (*Hint: Introduce a new input variable.*)

Exercise 1.12.6 Consider the *muddy children* problem discussed in Section 1.8.1. Show that, within finite time, all the children with a muddy forehead can simultaneously determine that they are not clean. (*Hint: Use induction on k.*)

Exercise 1.12.7 *Half-duplex* links allow communication to go in both directions, but not simultaneously. Design a protocol that implements half-duplex communication between two connected entities, a and b. Prove its correctness and analyze its complexity.

Exercise 1.12.8 *Half-duplex* links allow communication to go in both directions, but not simultaneously. Design a protocol that implements half-duplex communication between three entities, a, b and c, connected to each other. Prove its correctness and analyze its complexity.

1.12.2 Answers to Exercises

Answer to Exercise 1.12.1
Let us prove that every entity will indeed receive the message. The proof is by induction on the distance d of an entity from the initiator s. The result is clearly true for $d = 0$. Assume that it is true for all entities at most at distance d. Let x be a process at distance $d + 1$ from s. Consider a shortest path $s \to x_1 \to \ldots \to x_{d-1} \to x$ between s and x. As process x_{d-1} is at distance $d - 1$ from s, then by the induction assumption it receives the message. If x_{d-1} received the message from x, then this means that x already received the message and the proof is completed. Otherwise, x_{d-1} received the message from a different neighbor, and it then sends the message to all its neighbors, including x. Hence x will eventually receive the message.

Answer to Exercise 1.12.2
The total number of messages sent without the improvement was $\sum_{x \in \mathcal{E}} |N(x)| = 2|E| = 2m$; in Flooding, every entity (except the initiator) will send one message less. Hence the total number of messages is $2m - (|V| - 1) = 2m - n + 1$.

Answer to Exercise 1.12.6 (Basis of Induction only)
Consider first the case $k = 1$: Only one child, say z, has a dirty forehead. In this case, z will see that everyone else has a clean forehead; as the teacher has said that at least one child has a dirty forehead, z knows that he/she must be the one. Thus, when the teacher arrives, he/she comes forward. Notice that a clean child sees that z is dirty but finds out that his/her own forehead is clean only when z goes forward.
Consider now the case $k = 2$: There are two dirty children, a and b; a sees the dirty forehead of b and the clean one of everybody else. Clearly he/she does not know about his status; he/she knows that if he/she is clean, b is the only one who is dirty and will go forward when the teacher arrives. So, when the teacher comes and b does *not* go forward, a understands that his/her forehead is also dirty. (A similar reasoning is carried out by b.) Thus, when the teacher returns the second time, both a and b go forward.

BIBLIOGRAPHY

[1] K.M. Chandi and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[2] J.Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. *Journal of the A.C.M.*, 37(3):549–587, 1987.

[3] D.J. Lehmann. Knowledge, common knowledge and related puzzles. In *3rd ACM Symposium on Principles of Distributed Computing*, pages 62–67, Vancouver, 1984.

[4] N.A. Lynch and M.R. Tuttle. Hierarchical correctness proofs of distributed algorithms. In *6th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 137–151, Vancouver, 1987.

[5] S.J. Rosenschein. Formal theories of AI in knowledge and robotics. *New Generation Computing*, 3:345–357, 1985.