Design and Analysis of Distributed Algorithms

Chapter 1:

DISTRIBUTED COMPUTING ENVIRONMENTS

Contents

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; e.g., 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 section, we will examine this universe in some detail.

1 Entities

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

Capabilities
Each entity $x \in \mathcal{E}$ is endowed with local (i.e., private, non-shared) 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 value is 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 $\mathcal{S}$; example of such values are "Idle", "Processing", "Waiting",... etc.

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.
- Change the value of the status register.

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

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

- Arrival of a message.
- Ringing of alarm clock.
- Spontaneous Impulse.

The arrival of a message and the ringing of the alarm clock are events that are external to the entity, but originate within the system: the message is sent by another entity, 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 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 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 requests is made. Appearing to entities as "acts of God", 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, terminating sequence of operations called *actions* .

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 actions).

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

Behaviour
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 occurs. Thus the specification will take the form

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

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

The behavioural specification, or simply *behaviour*, of an entity x is the set $\mathcal{B}(x)$ of all the rules that x obeys. This set must be *complete* and *non ambiguous*: for every possible event e

and status value s, there is one and only one rule in $\mathcal{B}(x)$ enabled by (s, e). In other words, x must always know exactly what it must do when an event occurs.

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

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

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

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

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

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

Property 1.1 *Every collective behaviour can be made homogeneous.*

This means that, if we are in a system where different entities have different behaviours, 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:

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

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

It is important to stress that, in a homogeneous system, although all entities have the same behavioural 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 which are unlawful for most other citizens.

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

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 $N_{out}(x) \subseteq \mathcal{E}$ the set of entities to which x can transmit a message directly; we shall call them the *out-neighbours* of x . Similarly, we denote by $N_{in}(x) \subseteq \mathcal{E}$ the set of entities from which x can receive a message directly; we shall call them the *in-neighbours* 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-neighbour 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-neighbours and send message to its out-neighbours. Messages received at an entity are processed there in the order they arrive; if more than one message arrives at the same time, they will be processed in arbitrary order (see Sect. 9). Entities and communication may fail.

Real systems described by this model differ depending on how entities can send messages to their out-neighbours. In *point-to-point communication* systems, an entity can send specific messages to a selected subset of its out-neighbours; in *wireless communication* systems, any message sent by an entity is received by all its out-neighbours. Clearly wireless communication can always be implemented with point-to-point communication, while the converse is

not true; hence, in the following, we will focus on distributed computing environments with point-to-point communication.

3 Axioms and Restrictions

The definition of distributed computing environments with point-to-point communication has two basic *axioms*, one on communication delays, 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*.

3.1 Axioms

3.1.1 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.

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

Axiom 3.1 Finite Communication Delays
In absence of failures, communication delays are finite

In other words, in absence of failures, a message sent to an out-neighbour 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 absence of failure, a message will arrive after a finite amount of time without corruption.

3.1.2 Local Orientation

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

Axiom 3.2 Local Orientation
An entity can distinguish among its in-neighbours.
An entity can distinguish among its out-neighbours.

In particular, an entity is capable of sending a message to a specific out-neighbour (without having to send it also to all other out-neighbours). 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-neighbours 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 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).

Figure 1: Every edge has two labels

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

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, 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, or to reliability, or synchrony, etc. In the following, we will discuss some of the most common.

3.2.1 Communication Restrictions

A 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 Sect. 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, FIFO queues are characterized by the following restriction:

- *Message Ordering:* In absence of failure, the messages transmitted by an entity to the same out-neighbour 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. 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 full duplex can be described by the following restriction:

- *Reciprocal Communication:* $\forall x \in \mathcal{E}, N_{in}(x) = N_{out}(x)$.

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

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 $N(x) = N_{in}(x) = N_{out}(x)$ will just be called the set of *neighbours* of x. Note that in this case, $m(\vec{G}) = |\vec{E}| = 2\ |E| = 2\ m(G)$.

For example, in Figure 2 is depicted a graph $\vec{G}$ where the *Bidirectional Links* restriction holds, and the corresponding undirected graph G.

Figure 2: In a network with Bidirectional Links we consider the corresponding undirected graph.

3.2.2 Reliability Restrictions

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

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

- *Edge Failure Detection/Recovery:* $\forall(x, y) \in \vec{E}$, both x and y will detect whether (x, y) has failed and, following its failure, whether it has been reactivated.
- *Entity Failure Detection/Recovery:* $\forall x \in V$, all in- and out- neighbours 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: e.g., 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:* No failures have occurred nor will occur.

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

3.2.3 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 relayers. A system which 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, Connectivity will simply state that G is connected.

3.2.4 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 absence of failures, the communication delay of any message on any link is at most Δ.

A special case of bounded delays is:

- *Unitary Communication Delays:* In 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.

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, ...), 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 which 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? 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?).

4.1 Amount of Communication Activities

The transmission of a message through an out-port (i.e., to an out-neighbour) is the basic *communication activity* in the system; note that, the transmission of a message which will not be received due to 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* $\mathbf{L}_{node} = \mathbf{M}/|V|$, i.e. the number of messages per entity; and the *transmission load* $\mathbf{L}_{link} = \mathbf{M}/|E|$, that is the number of messages per link.

Messages are sequences of bits; some protocols might employ messages which 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 which use different sizes of messages, it might be necessary to use as a cost measure the number of transmitted bits $\mathbf{B}$ also called *bit complexity*.

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

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 3.1). In other words, communication delays are in general unpredictable. Thus, even in 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*, $\mathbf{T}$: the execution delay experienced under the restrictions "Unitary Transmission Delays" and "Synchronized Clocks"; that is, when the system is synchronous and (in absence of failure) takes one unit of time for a message to arrive and be processed.

A very different cost measure is the *causal time complexity* $\mathbf{T}_{causal}$. 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 very difficult to measure exactly; we will employ it only once, when dealing with synchronous computations.

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, which it would like to share 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 which, 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 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:

(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:

(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 neighbours. "

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*; denote by I the information to be broadcasted. Here is the set of rules $B(x)$ (the same for all entities):

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

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

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 !

Figure 3: An execution of Flooding.

Consider for example the simple system with three entities x, y, z connected to each other (see Figure 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).

To avoid this unwelcome effect, an entity should send the information to its neighbours 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. $idle \times Receiving(\mathrm{I}) \longrightarrow$ { Process(I); **become** *done*; **send**(I) **to** $N(x)$ }
3. $initiator \times Receiving(\mathrm{I}) \longrightarrow$ **nil**
4. $idle \times \iota \longrightarrow$ **nil**
5. $done \times Receiving(\mathrm{I}) \longrightarrow$ **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 12.1).

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, *initiator* or not, sends the information to all its neighbours. 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 neighbours, 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 neighbours; 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. $initiator \times \iota \longrightarrow \{$ **send**(I) **to** $N(x)$; **become** $done$ $\}$
2. $idle \times Receiving(\mathrm{I}) \longrightarrow \{$ Process(I); **become** $done$; **send**(I) **to** $N(x)$-**sender** $\}$
3. $initiator \times Receiving(\mathrm{I}) \longrightarrow$ **nil**
4. $idle \times \iota \longrightarrow$ **nil**
5. $done \times Receiving(\mathrm{I}) \longrightarrow$ **nil**
6. $done \times \iota \longrightarrow$ **nil**

where **sender** is the neighbour 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 12.2):

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

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 $r(x) = Max\{d(x, y) : y \in \mathcal{E}\}$, which is called the *eccentricity* (or *radius*) of x. In other words, the total time depends on which entity is the initiator, and thus can not 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 $d(G) = Max\{r(x) : x \in \mathcal{E}\}$ which is the *diameter* of G. In other words, the ideal time complexity will be at most the diameter of G:

$$\mathbf{T}[Flooding] \leq d(G) \tag{2}$$

6 States and Events

Once we have defined the behaviour of the entities, their communication topology, and the set of restrictions under which they operate, we must describe what are 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 constitutes the *initial internal state* $\sigma(x, 0)$ of x. Denote by $\Sigma(0) = \{\sigma(x, 0) : x \in \mathcal{E}\}$ the set of all the initial internal states.

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

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 will trigger 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 travelling 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 is due to conditions external to the entity). Different delays give raise 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 from an external observer, and it 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 is 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 occurs (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 occurs.

For example, in Figure 4 is depicted the TED corresponding to the execution of Protocol *Flooding* of Figure 3.

Figure 4: Time×Event Diagram

6.2 States and Configurations

The private memory of each entity, in addition to the behaviour, 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 constitutes what is called the *internal state of* x *at* t and 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 *can not 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 both in 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 neighbour). 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 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 state 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 neighbour), 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 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; i.e., the set $\Sigma(t)$. However, this is *not enough* ! In fact, the execution so far might have already generated some events which 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.

7 Problems and Solutions ($\star$)

The topic of this book is how to design distributed algorithms and analyze their complexity. A distributed algorithm is the set of rules which will regulate the behaviours of the entities. The reason why we may need to design the behaviours 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 which will always solve the problem in finite time. Let us discuss these concepts in some details.

7.1 Problems

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

$$\mathcal{P} =< P_{INIT}, P_{FINAL}, R >$$

where P_{INIT} and P_{FINAL} are *predicates* on the values of the registers of the entities, and R is a set of restrictions. Denote by $w_t(x)$ the value of an input register $w(x)$ at time t, and by $\{w_t\} = \{w_t(x) : x \in \mathcal{E}\}$ 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 Sect. 5, the initial and final conditions are given by the predicates

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

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

The restrictions we have imposed in 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 $\{BL, TR, CN, UI\}$.

7.2 Status

A solution protocol $\mathcal{B}$ for $\mathcal{P} =< P_{INIT}, P_{FINAL}, R >$ will specify *how* the entities will accomplish the required task. Part of the design of the set of rules $\mathcal{B}(x)$, 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 $\mathcal{S}$ which can be held at the start of the execution of $\mathcal{B}(x)$, and denote by $\mathcal{S}_{INIT}$ their set. By contrast, *terminal* status values are those values which, once reached, can not ever be changed by the protocol; their set shall be denoted by $\mathcal{S}_{TERM}$. All other values in $\mathcal{S}$ will be called *intermediate* status values.

For example, in the protocol *Flooding* described in Sect. 5, $\mathcal{S}_{INIT} = \{initiator, idle\}$, and $\mathcal{S}_{TERM} = \{done\}$.

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

7.3 Termination

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

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

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

We have already remarked on that 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

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

holds for some $t > 0$; i.e., all entities enter a final status after a finite time.

7.4 Correctness

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

$$\exists t > 0 : \ Correct(t)$$

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

7.5 Solution Protocol

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

Simple Solution[$\mathcal{B}$,$\mathcal{P}$] 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[$\mathcal{B}$,$\mathcal{P}$] 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}.

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.

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

$$p \in LK_t[x]$$

the fact that p is local knowledge at x at the global time instant t. By definition, $\lambda_x \in LK_t[x]$ 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 $p \in IK_t[W]$, if at least one entity in W knows p at time t; that is,

$$p \in IK_t[W] \textit{ iff } \exists x \in W \ (p \in LK_t[x]).$$

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 $p \in EK_t[W]$; that is

$$p \in EK_t[W] \textit{ iff } \forall x \in W \ (p \in LK_t[x])$$

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, $I \in LK_0[s]$. Thus, at that time, I is implicitly known to all entities; i.e., $I \in IK_0[\mathcal{E}]$. At the end of the broadcast, at time t', every entity will knows the information; in other words, $I \in EK_{t'}[\mathcal{E}]$.

Notice that, in absence of failures, knowledge can not be lost, only gained; that is, for all $t' > t$ and all $W \subseteq \mathcal{E}$, if no failure occurs, $IK_t[W] \subseteq IK_{t'}[W]$ and $EK_t[W] \subseteq EK_{t'}[W]$.

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 $p \in EK_t[W]$, the fact $''p \in EK_t[W]''$ might not even be local knowledge of any of the entities in W.

This gives rises 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 $p \in CK_t[W]$, 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 . . . , etc.* That is,

$$p \in CK_t[W] \text{ iff } \bigwedge_{1 \leq i \leq \infty} P_i$$

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

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 mudpuddles, 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 mudpuddle: the mud on your forehead is a dead giveaway !" and then

continues: "The guilty ones that come forward spontaneously will be given a small penalty; those who don't, 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 forehead are "dead meat", that will be punished no matter what. Obviously, she does not want to confess if the forehead is clean ! and clearly, if the forehead is dirty, she wants to go forward so to avoid the terrible punishment for those who do not confess ! Since each child shares the same concern, the collective goal is for the children with clean forehead 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 questions is: can a child x find out whether her forehead is dirty or not ? She 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: can *all* the children with mud on their forehead find out at the same time so 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 questions is positive, and find the way to achieve the common goal and thus common knowledge without communication (Exercise 12.6).

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

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 which, if they are common knowledge to the entities, may affect the complexity.

1. *Metric Information*: numeric information about the network; e.g., number $n = |V|$ of nodes, number $m = |E|$ of links, diameter, girth, etc. This information can be *exact*, or *approximate*.

2. *Topological Properties*: knowledge of some properties of the topology; e.g., "$\vec{G}$ is a ring network", "$\vec{G}$ does not have cycles", "$\vec{G}$ is a Cayley graph", etc.

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), etc.

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

Since 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 labelled 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. 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: *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: *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 5, this knowledge was assumed as part of the problem definition.

9 Technical Considerations

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.

Examples of field types are:

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 $M =< f_1, f_2, \ldots f_k >$ 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 $M =< f_1, f_2 >$ 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.

9.2 Protocol

9.2.1 Notation

A protocol $\mathcal{B}(x)$ is a set of rules. We have introduced already in Section 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 pair (*status*,*event*), 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 Sect. 5 will be written as shown in Figure 5.

PROTOCOL Flooding .

- Status Values: $\mathcal{S}$ = {INITIATOR, IDLE, DONE};
  $\mathcal{S}_{INIT}$ = {INITIATOR, IDLE};
  $\mathcal{S}_{TERM}$ = {DONE}.
- Restrictions: Bidirectional Links, Total Reliability, Connectivity, 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 5: Flooding Protocol

9.2.2 Precedence

The external events are: spontaneous impulse (*Spontaneously*), reception of a message (*Receiving*), and alarm clock ring ($When$). Different types of external events can occur simultaneously; e.g., 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 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 ever occur at an entity at any one time. Since there is locally only one alarm clock, at any time there will be at most one *When* event. On the other hand, it is possible that more than one message arrives at the same time to an entity from different neighbours; should this be the case, these simultaneous *Receiving* events have all the same precedence, and will be processed sequentially in an arbitrary order.

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-neighbour (or out-link), and *in-port*, one for each in-neighbour (or in-link). At an entity, each out-port has a distinct label (recall the Local Orientation axiom) 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 the port numbers associated to those neighbours; this is because an entity has no other information about its neighbours (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.

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,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, spontaneous impulse.

Entity's Behaviour Entities react to external events. The behaviour 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 non-ambiguous, and complete.

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

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

Neighbours The in-neighbours of an entity are those entities from which x can receive a message directly; the out-neighbours 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-neighbours, and between its in-neighbours.

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

Restriction Any additional property.

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 which can be employed within the formalism; such is for example the *Unity* model of Mani Chandy 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 which 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 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, etc.

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 IO Automata) use instead *active* entities. To understand this fundamental difference, consider a message in transit towards 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 pizza-delivery person 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.

12 Exercises, Problems, and Answers

12.1 Exercises and Problems

Exercise 12.1 *Prove that the flooding technique introduced in Sect. 5 is correct; that is, it terminates within finite time, and all entities will receive the information held by the initiator.*

Exercise 12.2 *Determine the exact number of message transmissions required by the protocol Flooding described in Sect. 5.*

Exercise 12.3 *In Sect. 5 we have solved the broadcasting problem under the restriction of Bidirectional Links. Solve the problem using the Reciprocal Communication restriction instead.*

Exercise 12.4 *In Sect. 5 we have solved the broadcasting problem under the restriction of Bidirectional Links. Solve the problem* without *this restriction.*

Exercise 12.5 *Show that any protocol B can be rewritten so that $\mathcal{S}_{START}$ consists of only one status.* (Hint: Introduce a new input variable)

Exercise 12.6 *Consider the* muddy children *problem discussed in Section 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 12.7 Half duplex *links allow communication to go in both directions, but not simultaneously. Design a protocol which implements half-duplex communication between two connected entities, a and b. Prove its correctness and analyze its complexity.* (Hint: Treat the event of an entity wanting to send a message to a neighbour as a spontaneous impulse for that entity.)

Exercise 12.8 Half duplex *links allow communication to go in both directions, but not simultaneously. Design a protocol which implements half-duplex communication between three entities, a, b and c, connected to each other. Prove its correctness and analyze its complexity.* (Hint: Treat the event of an entity wanting to send a message to a neighbour as a spontaneous impulse for that entity.)

12.2 Answers to Selected Exercises

Answer to Exercise 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 distance at most 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 Since 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. Eventually, x will receive this message as its first message, or it will not receive it since it received an earlier message. In any case we conclude that x will eventually receive the message.

Answer to Exercise 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 less message. Hence the total number of messages is $2m - (|V| - 1) = 2m - n + 1$.

Answer to Exercise 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; since the teacher has said that at least one child has a dirty forehead, z knows that he must be the one ! Thus, when the teacher arrives, he comes forward. Notice that a clean child sees that z is dirty but finds out that 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 does not know about his status; he knows that if he is clean, b is the only dirty one and will go forward when the teacher arrives. So, when the teacher comes and b does *not* go forward, a understands that also her forehead is dirty. (A similar reasoning is carried out by b.) Thus, when the teacher returns the second time, both a and b go forward.

References

[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 envirobment. *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.