Distributed computing

Bernadette Charron-Bost, Jean-Marc Notin

Distributed computing

Bernadette Charron-Bost and Jean-Marc Notin

July 12, 2022

Chapter 1

The model

1.1 Introduction

The term *distributed algorithm* refers to a class of algorithms aimed at being executed by autonomous communicating processing units, called *processes*, *processors*, or *agents*. Such algorithms are used in various applications. They depend on many features of the distributed system (computing units, communication medium, global control, ...) in which they are run.

Communication medium. Agents can interact following two main models: in the *shared memory* model, agents communicate by reading and writing shared variables; in the *message passing* model, they exchange information by sending and receiving messages through communication channels. There are different shared memory models depending on the semantics of the memory access primitives (types of the shared objects).

In the same way, several message passing models exist: for instance, messages can be sent point-to-point or broadcasted over the network; communication may be *by hand-shake* [1] (i.e., send and receive are atomic) or *asynchronous*; communication channels may ensure that messages are received in the same order they have been sent (FIFO property).

In the context of this document, we consider only message passing distributed systems with point-to-point communications. The collection of communication channels induces a directed graph (or digraph, for short), called the *communication graph* or *communication network*, where agents are the vertices of the digraph and communication channels are its edges. This digraph will be denoted by $G = (V, E)$.

Global information and implicit knowledge. Without any specific assumptions, agents are supposed to have only local information: each agent has a complete control on incoming and outgoing information (number and identifiers of its ports) and local variables.

However, the local algorithm associated to an agent (cf. below) may make use of some global information like the size of the network or the diameter of the communication graph.

Another property that can be used in a distributed algorithm is the availability of unique identifiers: each agent may use its identifier to tag its messages.[2] Otherwise, the networked system is said to be *anonymous*.

[1] The terms *communications with blocking receipts* or (unfortunately) *synchronous communications* are also used.

[2] However, identifiers are not supposed to be *mutually known*, i.e., local algorithms cannot use the set of identifiers.

1.2 Computational model

A *distributed algorithm* for a networked system of agents is a collection of finite state automata, one per agent. Automata may be *deterministic* or not.

Agents communicate with each other by sending and receiving messages over end-to-end communication channels. The underlying communication graph is supposed to be fixed and *strongly connected*. We will assume that communications are non-faulty: no message is lost, duplicated, or garbled by the communication medium.

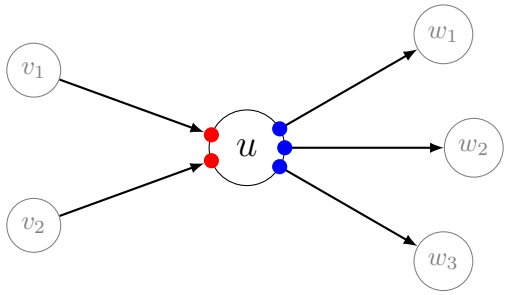


Figure 1.1: Anatomy of an agent

Each agent can send and receive messages along its incoming and outgoing channels, respectively. In Figure 1.1, the agent u has two incoming channels from v_1 and v_2, or equivalently, v_1 and v_2 are its *incoming neighbors*. Similarly, it has three outgoing neighbors, namely w_1, w_2, and w_3. The red dots are the u's incoming ports and the blue ones are its outgoing ports. The fact that an agent can distinguish its incoming ports is called the *non-masquerading property*.

1.3 Formal model

Let $\mathcal{M}$ be a non-empty set, modelling the set of all possible messages, $\mathbb{M}(\mathcal{M})$ be the set of finite multisets with elements from $\mathcal{M}$, and let $\mathbb{M}(E, \mathcal{M}) = \mathbb{M}(\mathcal{M})^E$ where E is the set of edges in the communication graph.

1.3.1 States, configurations, transitions

The following definition formally defines the local algorithm of an agent u as a transition system over a set of states Σ_u.

Definition 1.1. A local algorithm of an agent u is a sextuple $\mathcal{A}_u = (\Sigma_u, \mathcal{I}_u, \mathcal{M}, \vdash_u^i, \vdash_u^s, \vdash_u^r)$ where Σ_u is a set of states, $\mathcal{I}_u \subset \Sigma_u$ is the set of initial states, $\mathcal{M}$ is the set of all messages, $\vdash_u^i$ is a relation on $\Sigma_u \times \Sigma_u$, $\vdash_u^s$ and $\vdash_u^r$ are relations on $\Sigma_u \times \mathcal{M} \times Out_u \times \Sigma_u$ and $\Sigma_u \times \mathcal{M} \times In_u \times \Sigma_u$, respectively.

Let us consider a distributed algorithm $\mathcal{A} = (\mathcal{A}_u)_{u \in V}$; a configuration of $\mathcal{A}$ consists of the collection of local states, one per agents, and a collection of multisets[3] of messages, one per channel, i.e., an element of $\mathbb{M}(E, \mathcal{M})$. A configuration thus models a global state of the system: if $M \in \mathbb{M}(E, \mathcal{M})$ and (u, v) is a channel in E, then $M_{(u,v)}$ represents the state of this channel, that is to say the set of messages in transit along the channel. The *initial configurations* are the configurations where each agent is in an initial state and the message collection is empty.

[3]The structure of messages in transit may be refined depending on the type of communications.

The system evolves as individual agents execute actions following their local algorithms. A transition of the (global) system corresponds to a transition of a single agent u, which may affect not only the state of u but also the collection of messages in transit.

Definition 1.2. The transition system induced by a distributed algorithm for a set of agents $V = \{u_1, \ldots, u_n\}$, is the triple $(\mathcal{C}, \mathcal{I}, \vdash)$ where

- $\mathcal{C}$ is the set of *configurations*. A *configuration* $C \in \mathcal{C}$ is a tuple $(\sigma_{u_1}, \ldots, \sigma_{u_n}, M)$ where $\sigma_{u_i} \in \Sigma_{u_i}$ for every $u_i \in V$ and $M \in \mathbb{M}(E, \mathcal{M})$.
- $\mathcal{I}$ is the set of initial configurations, i.e., $C = (\sigma_{u_1}, \ldots, \sigma_{u_n}, M)$ is in $\mathcal{I}$ if for every agent u_i, σ_{u_i} is an initial state of the agent u_i and $M = \emptyset^E$.
- $\vdash$ is a binary relation on $\mathcal{C}$ defined as follows: $C \vdash C'$ if $C = (\sigma_{u_1}, \ldots, \sigma_{u_n}, M)$ and $C' = (\sigma'_{u_1}, \ldots, \sigma'_{u_n}, M')$ where there exists a unique $k \in \{1, \ldots, n\}$ such that forall $j \neq k$, $\sigma'_{u_j} = \sigma_{u_j}$ and
  - either $(\sigma_{u_k}, \sigma'_{u_k}) \in \vdash^i_{u_k}$;
  - or there exist $m \in \mathcal{M}$ and $v \in Out_{u_k}$ such that $(\sigma_{u_k}, m, v, \sigma'_{u_k}) \in \vdash^s_{u_k}$, and M' and M are identical except for the channel (u_k, v) where $M'_{(u_k,v)} = M_{(u_k,v)} \cup \{m\}$;
  - or there exist $m \in \mathcal{M}$ and $v \in In_{u_k}$ such that $m \in M_{(v,u_k)}$, $(\sigma_{u_k}, m, v, \sigma'_{u_k}) \in \vdash^r_{u_k}$, and M' and M are identical except for the channel (v, u_k) where $M'_{(v,u_k)} = M_{(v,u_k)} \setminus \{m\}$.

Let us now introduce some notation. First, if $u \in V = \{u_1, \ldots, u_n\}$ and $C = (\sigma_{u_1}, \ldots, \sigma_{u_n}, M)$ is any configuration in a distributed algorithm, then $\sigma_u(C)$ denotes the internal state of the agent u in C, namely $\sigma_u(C) = \sigma_u$. Similarly, if (u, v) is an edge of the communication graph, then $M_{(u,v)}(C)$ denotes the multiset of messages in transit from u to v, and $M(C)$ is the state of the communication medium, namely $M(C) = M$.

From Definition 1.2, it is clear that a *global transition* $C \vdash C'$ correspond to a unique *local transition* of an agent u. Conversely, a local transition of some agent u may correspond to several global transitions. More precisely, from Definition 1.1 and Definition 1.2, we immediately obtain the following applicability conditions.

Lemma 1.1. *Let C be any configuration of an algorithm and let $\vdash_u$ be a transition of some agent u.*

- *If $\vdash_u$ corresponds to $(\sigma, \sigma') \in \vdash^i_u$, then $\vdash_u$ is applicable to C if and only if $\sigma_u(C) = \sigma$.*
- *If $\vdash_u$ corresponds to $(\sigma, m, v, \sigma') \in \vdash^s_u$, then $\vdash_u$ is applicable to C if and only if $\sigma_u(C) = \sigma$.*
- *If $\vdash_u$ corresponds to $(\sigma, m, v, \sigma') \in \vdash^r_u$, then $\vdash_u$ is applicable to C if and only if $\sigma_u(C) = \sigma$ and $m \in M_{(v,u)}(C)$.*

1.3.2 Executions and computations

Definition 1.3. An *execution* of the distributed algorithm $\mathcal{A}$ corresponding to the transition system $\mathcal{A} = (\mathcal{C}, \mathcal{I}, \vdash)$ is a finite or infinite sequence of configurations $(C_0, C_1, \ldots, C_t, \ldots)$ such that:

- every C_t is a configuration of $\mathcal{A}$, i.e., $C_t \in \mathcal{C}$;
- C_0 is an initial configuration, i.e., $C_0 \in \mathcal{I}$;

- for every index t, $C_t \vdash C_{t+1}$.

An execution is *complete* if it cannot be extended, i.e., either it is infinite or the final configuration C_t is such that there is no configuration C such that $C_t \vdash C$.

Due to asynchronism, a distributed algorithm admits several complete executions even with deterministic local algorithms: they depend on the agent schedule (who takes the next step) and on the message schedule (what message in transit is received). That corresponds to *external non-determinism*, as opposed to the possible internal non-determinism in local algorithms.

Consider an execution $\alpha = (C_0, C_1, \dots, C_t, \dots)$. Any transition $C_t \vdash C_{t+1}$ in α is the occurrence of a (local) transition of a single agent u. Hence, α can be defined by giving the initial configuration C_0 and the sequence $e_1, \dots, e_t, \dots$, where e_t is called an *event*, and is the occurrence of the local transition involved in $C_{t-1} \vdash C_t$.

The event e is said to be an *internal event* (resp. a *send event*, a *receive event*) when e corresponds to a local transition of the form $\vdash^i_u$ (resp. $\vdash^s_u$, $\vdash^r_u$). In each of these three cases, e is performed by u, which is denoted $agent(e) = u$.

Note that two distinct events e_i and e_j may correspond to the same transition by the same agent u but applied to two distinct configurations C_{i-1} and C_{j-1}, which can differ on the state of M or on the states of agents other than u (cf. Example 1.3.2).

Conversely, each event in α corresponds to a *single* local transition of the algorithm. Hence, we may say that *the event e is applicable to a configuration C* if the underlying local transition is applicable to C in the sense of Lemma 1.1).

Example. Consider a system of two agents u and v such that there is a communication channel from u to v, from v to u, and the set of messages is $\mathcal{M} = \{\star\}$. The local algorithm of u and v is given by:

$$\Sigma_\square = \mathcal{I} = \{\iota\} \quad \vdash^i_\square = \emptyset \quad \vdash^s_\square = \vdash^r_\square = \{(\iota, \star, \bar{\square}, \iota)\}$$

where $\square \in \{u, v\}$ and $\bar{u} = v \wedge \bar{v} = u$.

The complete executions of this system are infinite sequences of occurrences of the four transitions $\{s_u, r_u, s_v, r_v\}$ with $s_u = (\iota, \star, v, \iota) \in \vdash^s_u$, $r_u = (\iota, \star, v, \iota) \in \vdash^r_u$, $s_v = (\iota, \star, u, \iota) \in \vdash^s_v$, and $r_v = (\iota, \star, u, \iota) \in \vdash^r_v$.

Notation:

1. If C is a configuration such that e is applicable to C, then $e \cdot C$ denotes the configuration C' obtained by applying the transition involved in e to C.

   For example, if $C = (\sigma_1, \dots, \sigma_u, \dots, \sigma_n, M)$ and e is a send event given by $(\sigma_u, m, v, \sigma'_u) \in \vdash^s_u$, then $e \cdot C = (\sigma_1, \dots, \sigma'_u, \dots, \sigma_n, M')$ where M' and M are identical except for the channel (u, v) where $M'_{(u,v)} = M_{(u,v)} \cup \{m\}$.

2. If $\alpha = (C_0, C_1, \dots, C_t, \dots)$ is an execution and x_u is a variable of the agent u, then $x_u(t)$ denotes the value of x_u in the configuration C_t. If e denotes the t-th event in α, then $x_u(e)$ denotes the value of x_u just after e, i.e., in configuration C_t.

3. Let $\hat{m}$ denotes the *occurrence* of a message[4] received in an execution α. Because of the definitions in Section 1.3.1, the receipt of $\hat{m}$ corresponds to a *unique* sending of $\hat{m}$ that does occur *before* its receipt in α. These two events in α are denoted $s(\hat{m})$ and $r(\hat{m})$.

[4]Note that $\hat{m}$ is not an element of $\mathcal{M}$. Indeed, in the context of a given execution, occurrences of messages may be distinguished in the same way as occurrences of local transitions (i.e., events).

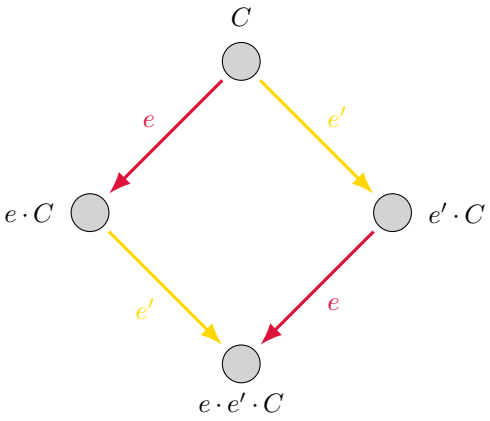


Figure 1.2: The Diamond Lemma

Definition 1.4. Let $\alpha = (C_0, e_1, \ldots, e_t, \ldots)$ be any execution of a distributed algorithm. The *direct causality relations* in the set of events in α are defined as follows:

$$e_i <_{\alpha,u} e_k \iff \begin{cases} agent(e_i) = agent(e_k) = u \\ \text{and} \\ i < k \quad (i.e.,\ e_i\ occurs\ before\ e_k\ in\ \alpha) \end{cases}$$

$$e_i \rightsquigarrow_\alpha e_k \iff e_i = s(\hat{m})\ and\ e_k = r(\hat{m})$$

The *causality relation*, denoted $\prec_\alpha$, is defined as the transitive closure of $\left(\bigcup_{u \in V} <_{\alpha,u}\right) \cup (\rightsquigarrow_\alpha)$.

The relation $\prec_\alpha$ is a partial order on the set of events in α. If e_i and e_k are not related by $\prec_\alpha$, i.e., $\neg(e_i \prec_\alpha e_k)$ and $\neg(e_k \prec_\alpha e_i)$, then e_i and e_k are said to be *concurrent in* α. The following lemma shows that the total ordering of events in α preserves the causality relation $\prec_\alpha$, i.e., α is a *linear extension of* $\prec_\alpha$.

Lemma 1.2. *If $e \prec_\alpha e'$ then e occurs before e' in α.*

Proof. Let $\alpha = (C_0, e_1, \ldots, e_t, \ldots)$ be any execution. First, if e is a direct cause of e', i.e., $e <_{\alpha,u} e'$ or $e \rightsquigarrow_\alpha e'$, then e occurs before e' in α by definition of $<_{\alpha,u}$ and $\rightsquigarrow_\alpha$ (see Definition 1.4). In the general case, by definition of the causality relation $\prec_\alpha$, there exist a finite set of indices $i_0, \ldots, i_k$ such that $e = e_{i_0}$, $e' = e_{i_\ell}$, and every e_{i_k} is a direct cause of $e_{i_{k+1}}$. From the above remark, we have that $i_k < i_{k+1}$. By transitivity, we obtain that $i_0 < i_\ell$, i.e., e occurs before e' in α. □

The following lemma is useful in many proofs of the next chapters. It involves several executions (as opposed to the previous lemma) and demonstrates some commutativity properties of the applicability relation. It is illustrated in Figure 1.2.

Lemma 1.3 (Diamond lemma)**.** *Let C be a configuration of a distributed algorithm, and let e and e' be two events of distinct agents, both applicable to C. Then, e' is applicable to $e \cdot C$, e is applicable to $e' \cdot C$, and $e' \cdot e \cdot C = e \cdot e' \cdot C$.*

Proof. The proof is by a rather tedious case analysis, and we limit it to the case where e is a send event and e' is a receive event. Let $agent(e) = u$ and $agent(e') = v$, and let m and m' denote the

messages involved in e and e', respectively. Since $u \neq v$, we have $\sigma_v(e \cdot C) = \sigma_v(C)$. Moreover, the message m' sent to v is in transit in C since e' is applicable to C. Because e is a send event, m' is still in transit in $e \cdot C$. Lemma 1.1 shows that e' is applicable to $e \cdot C$. Similarly, we obtain that e is applicable to $e' \cdot C$.

We easily check that for any agent w distinct from u and v, $\sigma_w(e' \cdot e \cdot C) = \sigma_w(e \cdot e' \cdot C) = \sigma_w(C)$. Moreover, $\sigma_u(e' \cdot C) = \sigma_u(C)$ and $\sigma_u(e' \cdot e \cdot C) = \sigma_u(e \cdot C)$. Hence, $\sigma_u(e' \cdot e \cdot C) = \sigma_u(e \cdot e' \cdot C) = \sigma_u(e \cdot C)$. In the same way, we obtain that $\sigma_v(e' \cdot e \cdot C) = \sigma_v(e \cdot e' \cdot C) = \sigma_v(e' \cdot C)$.

The same arguments show that $M_c(e' \cdot e \cdot C) = M_c(e \cdot e' \cdot C) = M_c(C) \cup \{m\}$ if c is the channel involved in e, $M_c(e' \cdot e \cdot C) = M_c(e \cdot e' \cdot C) = M_c(C) \setminus \{m'\}$ if c is the channel involved in e', and $M_c(e' \cdot e \cdot C) = M_c(e' \cdot e \cdot C) = M_c(C)$ otherwise. It follows that $e' \cdot e \cdot C = e \cdot e' \cdot C$. □

Then we give a third obvious lemma, whose proof is omitted, providing examples in which the previous lemma applies.

Lemma 1.4. *Let α be an execution of a distributed algorithm with two consecutive events e_t and e_{t+1} that are concurrent in α, and let C be the configuration in α just before applying e_t. Then the agents performing e_t and e_{t+1} are different, and e_{t+1} is applicable to C.*

Lemma 1.5. *Let α be a finite execution of an algorithm $\mathcal{A}$ and let α' be a linear extension of the causality relation in α. Then α' is an execution of $\mathcal{A}$ and the final configurations of α and α' are equal.*

Proof. Let $\alpha = (C_0, e_1, \ldots, e_\ell)$. Since α' is a linear extension of $\prec_\alpha$, there exists a permutation σ of $\{1, \ldots, \ell\}$ such that $\alpha' = (C_0, e'_1, \ldots, e'_\ell)$ with $e'_t = e_{\sigma(t)}$. Moreover, σ preserves the causality relation in α, i.e.,

$$e_{\sigma(s)} \prec_\alpha e_{\sigma(t)} \implies s < t.$$

Recall that any permutation σ is a product of transpositions. Moreover, if σ preserves the causality relation $\prec_\alpha$, then the transpositions involved in this product may be chosen such that each of them also preserves $\prec_\alpha$. To prove the lemma, it thus suffices to prove it when σ is a transposition.

Let $\tau = \tau_{i,j}$ be such a transposition with $i < j$. First, we prove that for every index t such that $i < t \leqslant j$, e_t and e_i are concurrent in α. Since $i < t$, Lemma 1.2 shows that $\neg(e_t \prec_\alpha e_i)$. Now suppose $e_i \prec_\alpha e_t$. Since τ preserves $\prec_\alpha$, we have $\tau^{-1}(i) = j < \tau^{-1}(t)$. Moreover, either $\tau^{-1}(t) = t$ or $\tau^{-1}(t) = j$, depending on whether $t < j$ or $t = j$. Both cases lead to a contradiction, and the conclusion follows. Applying Lemmas 1.4 and 1.3 repeatedly, we obtain that $\alpha' = (C_0, e_1, \ldots, e_{i-1}, e_{i+1}, \cdots, e_j, e_i, e_{j+1}, \cdots, e_\ell)$ is an execution of $\mathcal{A}$.

In the same way, we show that every index t such that $i \leqslant t < j$, e_t and e_j are concurrent in α, and thus $\alpha' = (C_0, e_1, \ldots, e_{i-1}, e_j, e_{i+1}, \ldots, e_{j-1}, e_i, e_{j+1}, \ldots, e_\ell)$ is an execution of $\mathcal{A}$. □

Theorem 1.1. *Let α be any finite execution of a distributed algorithm $\mathcal{A}$, and let α' be any permutation of the elements in α. Then the following conditions are equivalent:*

1. *α' is a linear extension of the causality relation $\prec_\alpha$;*
2. *α' is an execution of $\mathcal{A}$.*

Moreover, if the above conditions hold, then the causality relations $\prec_\alpha$ and $\prec_{\alpha'}$ coincide and the final configurations of α and α' are equal.

Proof. The implication (1) ⇒ (2) is just Lemma 1.5. To show the converse implication, suppose that α' is an execution of $\mathcal{A}$. By definition, the direct causality relations are pairwise equal in α and in α', so $\prec_\alpha$ and $\prec_{\alpha'}$ coincide. And Lemma 1.2 implies that α' is a linear extension of this common causality relation. □

If the conditions in Theorem 1.1 both hold, then α and α' are said to be *equivalent*, which is denoted by $\alpha \sim \alpha'$. Then we define the equivalence class $\mathcal{C}_\alpha$ w.r.t. the relation $\sim$, i.e., $\mathcal{C}_\alpha$ is the set of all the linear extensions of $\prec_\alpha$. This equivalence class is called a *computation* of $\mathcal{A}$.

1.3.3 Space-time diagrams

Given an execution α of a distributed algorithm, the events in α can be visualized in a *space-time diagram*: An horizontal line is drawn for every agent u (each agent is sequential) and the events of u are drawn on the line from left to right, according to their occurrence in α. And, for every pair of events e, e' such that $e \rightsquigarrow e'$, an arrow is drawn from e to e'. As an example, the space-time diagram of an execution with three agents u, v and w is given in Figure 1.3.

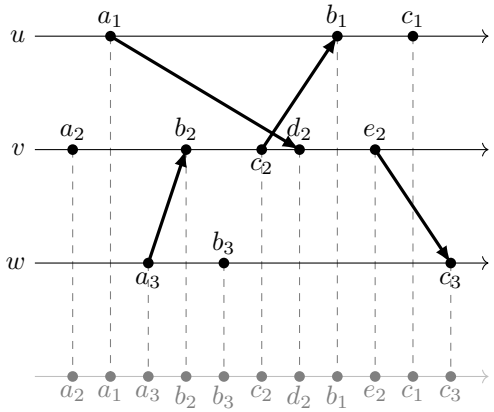


Figure 1.3: An example of a space-time diagram

Space-time diagrams help to visualizing the causality relation and the concept of computation. Indeed, the relations $\{<_u\}_{u \in V}$ and $\rightsquigarrow$ are naturally represented in the diagram, and the causality relation corresponds to the *paths*. For instance, one can easily see that a_3 is a cause of c_1 in Figure 1.4. In the same way, one can easily get that the events b_3 and d_2 are concurrent.

Given a space-time diagram, we can derive an execution by projecting the events on a single line (similar to "physical time"). A space-time diagram can also be used to generate an equivalent execution: events may be moved along a line, as long as

- the orderings of events on the same lines are all preserved;
- the arrows are drawn from left to right.

In other words, a space-time diagram actually pictures a computation. The diagram in Figure 1.5 is obtained from the diagram in Figure 1.3 by translating events on each line. The resulting execution is equivalent — w.r.t. the causality relation — to the original execution.

1.4 Complexity of a distributed algorithm

Usual complexity measures used for sequential algorithms (*space* and *time* complexity) does not encompass the peculiarities of distributed algorithms, and may be irrelevant when comparing

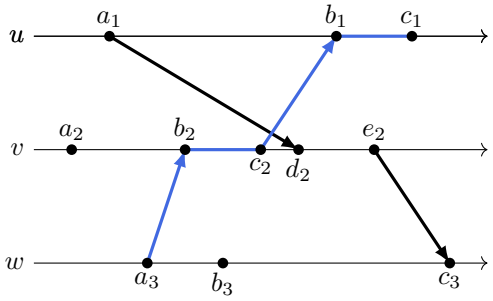


Figure 1.4: Causality in a space-time diagram

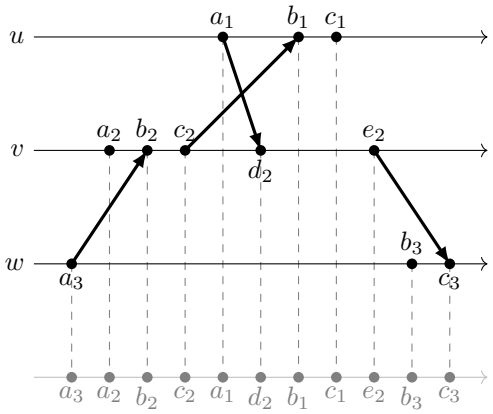


Figure 1.5: An equivalent execution

distributed algorithms. Two specific complexity measures may be defined, namely the *message complexity* and the *time complexity*.

The message complexity of an execution is defined as the total number of messages exchanged in the execution.

For time complexity, we first introduce the relation $<$ defined on the set of messages exchanged in an execution as follows:

$$m < m' \iff r(m) \prec s(m')$$

The relation $<$ is a partial ordering, and we may consider the chains of messages connected by $<$. The time complexity of an execution is then defined as the length of the longest chain of messages in the execution.

Other complexity measures may be considered: for instance, the *bit complexity* measure that takes into account the total number of bits exchanged in the execution.

1.5 Pseudo-code for distributed algorithms

The description of local algorithms as transition systems is clearly too low level, and the use of pseudo-codes is thus highly preferable. We use a *non-deterministic* pseudo-code, based on *guarded*

commands that allows agents (automata) to have a high degree of (internal) non-determinism. An algorithm written in the non-deterministic pseudo-code has the following form:

$$(g_1 \to cmd_1 \Box \cdots \Box\, g_n \to cmd_n)^*$$

where $g_1, \ldots, g_n$ are boolean expressions called *guards* and $cmd_1, \ldots, cmd_n$ are sequences of statements called *commands*. The operator $\Box$ carries out a non-deterministic choice among open guards, i.e., selects one command block cmd_{k_0} among the set $\{cmd_k \mid g_k\}$ and executes cmd_{k_0}. Finally, the operator $*$ executes infinitely many times the block of guarded commands $(g_1 \to cmd_1 \Box \cdots \Box g_k \to cmd_k)$.

Commands. The syntax used for writing commands is similar to the one used for usual (sequential) pseudo-code. The language includes:

- *Expressions*, that are well formed and well typed terms built from variables, constant values and operators. Variables, constants and expressions may be of base type (boolean, integer or real), of array type or of set type. Operators are the usual operators defined for the corresponding types, e.g., $+$, $-$, $/$ and $*$ for integer and real expressions; $\neg$, $\wedge$, $\vee$ and comparison operators ($<$, $\leqslant$, $=$, $\neq$, ...) for boolean expressions; and usual set operators ($\cup$, $\cap$, $\setminus$, ...) for expressions involving sets.
- *Statements*, including assignments (Figure 1.6a), conditionals (Figure 1.6b) and loops (Figure 1.6c).

$x \leftarrow x + 1$

$Rec_u[v] \leftarrow \textbf{true}$

(a) Assignements

if <*cond*> **then**
  <*body*>
else
  <*body*>
end if

(b) Conditionals

for all <*cond*> **do**
  <*body*>
end for

while <*cond*> **do**
  <*body*>
end while

(c) Loops

Figure 1.6: Statements available in pseudo-code

Guards. A *guard* is a boolean expression that may involve the variables of the agent and the state of the incoming channels (see the next paragraph). It is always associated to some command (hence the term *guarded commands* used earlier). In a given configuration C, a guard is *opened* if the boolean expression evaluates to **true**, in which case the associated command may be selected for execution. If more than one guard is opened, a non-deterministic choice is made among opened guards and only one command is executed.

Here is an example of a guarded command:

$$\begin{array}{l} \mathbf{D_u} :: \ \{\forall v \in In_u \ rec_u[v] = \textbf{true}\} \\ \quad dec_u \leftarrow \textbf{true} \end{array}$$

$\mathbf{D_u}$ is a (optional) label, used for referencing the guarded command; the expression "$\forall v \in In_u \ rec_u[v] = \textbf{true}$" is the guard; the statement "$dec_u \leftarrow \textbf{true}$" is the associated command.

send $\langle msg \rangle$ to v

(a) Sending messages

{A message $\langle m \rangle$ arrives from v}
  receive $\langle m \rangle$
  …

(b) Receiving messages

Figure 1.7: Handling of messages in pseudo-code

Communications. We introduce a new instruction dedicated to sending messages, given in Figure 1.7a, where msg is an expression and v is an outgoing neighbor (i.e., $v \in Out_u$)[5]. Sends are non-blocking, in the sense that the messages to be sent are immediately forwarded to the communication medium and cannot block the execution of the algorithm.

For receipts, we introduce a two-fold syntax, as shown in Figure 1.7b. In addition to usual boolean expressions, guards can use informal statements related to the state of incoming channels like "A message $\langle m \rangle$ arrives from v". Note that the "from v" part is optional; when present, it binds v to the port number on which the message has been received. Hence, the semantics of the receive primitive as well as this type of conditionals allow agents to inspect a message before receiving it. Secondly, the special instruction "receive $\langle m \rangle$" performs the actual receipt of $\langle m \rangle$, i.e., the removal of $\langle m \rangle$ from the communication medium. It must be used exclusively related to a guard involving the availability of the message to be received. Note that, in this context, the "receive" instruction is non-blocking.

{A message $\langle \mathbf{tok}, a, b \rangle$ arrives from v}
  receive $\langle \mathbf{tok}, a, b \rangle$
  …

Figure 1.8: Message tuples

The above description of communication primitives allows algorithms to handle various kinds of messages: messages may be tuples whose first entry is a constant value used to filter specific messages. Consider for example the snippet given in Figure 1.8. The guard will be opened when a message $\langle m \rangle$ arrives such that m is a 3-tuple with a first entry that is the constant value **tok**; the second and third entries will be bound to the variables a and b, respectively.

An example of a distributed algorithm written in non-deterministic pseudo-code is given in Algorithm 1. Note that in the example, the $\square$ and $*$ operator are left implicit, i.e., the algorithm should be read as $(\mathbf{I_1} \square \mathbf{I_2} \square \mathbf{S} \square \mathbf{R} \square \mathbf{D})^*$.

The non-deterministic pseudo-code allows to deal with two kinds of non-determinism: an *external* one and an *internal* one. The communication model ensures that messages are transmitted in an arbitrary (but finite) time. Thus, messages arrive in an arbitrary ordering, which provides a (external) source of non-determism in the execution of agents. On the other side, the $\square$ operator enforces non-determinism in the execution of internal actions. In Algorithm 1, both $\mathbf{I_1}$ and $\mathbf{I_2}$ guards are opened initially. The $\square$ operator results in a non-deterministic choice among $\mathbf{I_1}$ and $\mathbf{I_2}$ (and potentially other opened guards)

[5] According to the model of communications given in Section 1.2, v stands for the port number corresponding to the communication channel from u to v.

Algorithm 1 A algorithm in non-deterministic pseudo-code

State variables for agent u:
x, y, with values in $\{\textbf{red}, \textbf{yellow}, \perp\}$, initially $\perp$
$rcount, ycount$, integers, intially 0

$\mathbf{I_1}$ :: $\{x = \perp\}$
$x \leftarrow \textbf{red}$
$rcount \leftarrow rcount + 1$

$\mathbf{I_2}$ :: $\{x = \perp\}$
$x \leftarrow \textbf{yellow}$
$ycount \leftarrow ycount + 1$

S :: $\{x \neq \perp$, once$\}$
for $v \in Out_u$ **do**
send $\langle x \rangle$ to v
end for

R :: {A message $\langle m \rangle$ arrives}
receive $\langle m \rangle$
if $m =$ **red then**
$rcount \leftarrow rcount + 1$
else
$ycount \leftarrow ycount + 1$
end if

D :: $\{rcount + ycount = N\}$
if $rcount \geqslant ycount$ **then**
$y \leftarrow \textbf{red}$
else
$y \leftarrow \textbf{yellow}$
end if

1.6 Traversal vs. round-based algorithms

In this section, two special classes of distributed algorithms will be discussed, namely the *traversal* algorithms for which all events in any execution are totally ordered by the causality relation, and the *round-based* algorithms whose executions are structured into communication closed layers called *rounds*.

Definition 1.5. A traversal algorithm is a distributed algorithm with the following two properties.

1. In each execution, there is only one initiator, which starts the algorithm by sending out exactly one message.

2. Upon the receipt of a message, any agent sends out one message.

In each reachable configuration of a traversal algorithm, there is either one message in transit, or exactly one agent that has just received one message and not (yet) sent a message in reply. In a more abstract way, the messages exchanged in any execution, if taken together, can be seen as a single object (a *token*) that is handed from agent to agent, and so "visits" agents.

Classically traversal algorithms are obtained by running a token algorithm on top of any distributed algorithm $\mathcal{A}$: every guard in $\mathcal{A}$ is augmented with the condition that the agent hands the token, and the token travels along communication paths in $\mathcal{A}$. The resulting algorithm is thus a traversal version of $\mathcal{A}$. As an example, the traversal version of the Echo algorithm coincides with the Tarry algorithm (cf. Chapter 2).

Algorithm 2 The token algorithm

Initialization:

$token_u \in \{\mathbf{false}, \mathbf{true}\}$, initially **true** if u is the initiator and **false** otherwise

Guarded commands:

$\mathbf{RT_u}$ :: {A message $\langle \mathbf{tok} \rangle$ has arrived}
receive $\langle \mathbf{tok} \rangle$
$token_u \leftarrow \mathbf{true}$

$\mathbf{ST_u}$ :: $\{token_u = \mathbf{true}\}$
select $v \in Out_u$ ▷ usually according to $\mathcal{A}$
send $\langle \mathbf{tok} \rangle$ to v
$token_u \leftarrow \mathbf{false}$

Similarly to the use of a token, the round-based structure kills a certain degree of nondeterminism, but it does it in a totally opposite way that consists in enforcing concurrency (instead of sequentiality).

Algorithms are structured in *synchronized rounds*: In each round, an agent first emits messages, then receives messages, and finally makes a state transition (internal events). Rounds are *synchronized* in the sense that any message received at round t has been sent in round t. Observe that *all* agents are initiators in a round-based algorithm.

Any algorithm $\mathcal{A}$ can be easily structured into synchronized rounds using empty messages (w.r.t. $\mathcal{A}$'s semantics) and simple integer counters. Algorithm 3 for round simulation can be run on top of any distributed algorithm $\mathcal{A}$: at every round, an agent sends messages it has to send in $\mathcal{A}$ if any; otherwise it sends an empty message (first instruction in $\mathbf{S_u}$). Moreover, the guarded command $\mathbf{I}_u$ is completed with the state transitions in $\mathcal{A}$ triggered by the receipts in $\mathbf{R}_u$.

In the resulting algorithm, called the *round-based version of* $\mathcal{A}$, agents operate in lock-step rounds. It prohibits some possible interleavings of $\mathcal{A}$'s events, and thus reduces the set of possible executions. Moreover, it induces a coarser grain of atomicity: the desired safety properties have to be verified only at the end of each round.

Question 1. *Give the pseudo-code of the round-based version of the Phase algorithm (cf. Chapter 2) and compare the set of its executions with the set of the executions of the Phase algorithm.*

Algorithm 3 Simulation of synchronized rounds

Initialization:

$status_u \in \{S, R, I\}$, initially S
$c_u \in \mathbb{N}$, initially 1
$rcount_u \in \mathbb{N}$, initially 0

Guarded commands:

$\mathbf{S_u} :: \{status_u = S\}$
send $\langle m_v, c_u \rangle$ to every outgoing neighbor v ▷ m_v depends on $\mathcal{A}$
$status_u \leftarrow R$

$\mathbf{R_u} :: \{status_u = R$ and a message $\langle m, c \rangle$ with $c_u = c$ has arrived$\}$
receive $\langle m, c \rangle$
$rcount_u \leftarrow rcount_u + 1$
if $rcount_u = |In_u|$ **then**
$rcount_u \leftarrow 0$
$status_u \leftarrow I$
end if

$\mathbf{I_u} :: \{status_u = I\}$
$c_u \leftarrow c_u + 1$
$status_u \leftarrow S$

Chapter 2

Distributed computation of a function

We consider a non-empty set $\mathcal{V}$ and a function $f : \mathcal{V}^n \mapsto \mathcal{V}$ stable by permutation, i.e.,

$$\forall v \in \mathcal{V}^n, \forall \sigma \in \Sigma_n,\ f(\mu_{\sigma(1)}, \ldots, \mu_{\sigma(n)}) = f(\mu_1, \ldots, \mu_n).$$

In other words, the value of f depends on the multi-set denoted $\{\!\{\mu_i \mid i \in \{1, \ldots, n\}\}\!\}$. We may also consider the restricted class of functions whose values only depend on the set $\{\mu_i \mid i \in \{1, \ldots, n\}\}$, i.e., functions $f : 2^{\mathcal{V}} \mapsto \mathcal{V}$. Besides, we consider the distributed algorithms over a directed graph $G = (V, E)$ where each agent u (1) has an initial value $\mu_u \in \mathcal{V}$ and (2) a *write once* variable $dec_u \in \mathcal{V} \cup \{\perp\}$, initialized to $\perp$. When assigning a value different from $\perp$ to dec_u, the agent u is said *to decide*. Such an algorithm $\mathcal{A}$ is said *to compute* f if any of its complete executions satisfies the following properties

$$\begin{array}{lll} \textit{Termination:} & \exists u \in V,\ \exists t, & dec_u(t) \neq \perp \\ \textit{Validity:} & \forall u \in V,\ \forall t, & dec_u(t) \neq \perp \implies dec_u(t) = f(\{\!\{\mu_u, u \in V\}\!\}). \end{array}$$

Roughly speaking, the two above properties specify that every agent eventually decides and that the sole possible decision value is $f(\{\!\{\mu_u \mid u \in V\}\!\})$. Termination may be strengthened by requiring that all the agents (and not only one) make a decision.

We now present some classical algorithms that compute functions of the *set* of initial values. Each of them require some specific properties on the communication graph G and some global knowledge or global control (e.g., identifiers, knowledge of the network size, ...).

2.1 The Gossip algorithm

The first algorithm that we present, called the *Gossip* algorithm, assumes agents to have unique identifiers. First, each agent tags its initial value with its identifier. Then it repeatedly collects messages from all its incoming neighbors and sends the set of collected initial values to all its outgoing neighbors. It decides after collecting N initial values, where N is a parameter of the algorithm. The pseudo-code for the Gossip algorithm is given in Algorithm 4.

Theorem 2.1. *The Gossip algorithm computes any function of the multi-set of initial values if the communication graph is strongly connected and the parameter N is the network size.*

Question 2. *Prove Theorem 2.1.*

Algorithm 4 The Gossip algorithm

Initialization:

$HO_u \in 2^{V \times \mathcal{V}}$, initially $\{(u, \mu_u)\}$
$new_u \in \{\mathbf{false}, \mathbf{true}\}$, initially **true**
$dec_u \in \mathcal{V} \cup \{\perp\}$, initially $\perp$

Guarded commands:

$\mathbf{R_u}$ :: {A message $\langle HO \rangle$ has arrived}
receive $\langle HO \rangle$
if $HO \not\subseteq HO_u$ **then**
$HO_u \leftarrow HO_u \cup HO$
$new_u \leftarrow$ **true**
end if

$\mathbf{S_u}$ :: $\{new_u =$ **true**$\}$
for all $v \in Out_u$ **do**
send $\langle HO_u \rangle$ to v
end for
$new_u \leftarrow$ **false**

$\mathbf{D_u}$ :: $\{|HO_u| = N$, once$\}$
$dec_u \leftarrow f(HO_u[2])$

2.2 The Tree algorithm

The second algorithm, called the *Tree* algorithm, assumes bidirectional links. The set of (incoming or outgoing) neighbors of u is denoted by N_u. Informally, the algorithm proceeds as follow. Each agent u can send only one message. Moreover, it can do it only when either it has received a message from each of its neighbors except one denoted by v or it has received a message from all its neighbors. In the first case, u is allowed to send its message to v while it sends it to any of its neighbor in the second case (non-deterministic choice). The agent u makes a decision when it has received a message from all its neighbors. The pseudo-code for the Tree algorithm is given in Algorithm 5.

Algorithm 5 The Tree algorithm (for a function f of sets)

Initialization:

$HO_u \in 2^{\mathcal{V}}$, initially $\{\mu_u\}$
$rec_u \in \{\mathbf{true}, \mathbf{false}\}^{N_u}$, initially $[\mathbf{false}, \ldots, \mathbf{false}]$
$sent_u \in \{\mathbf{true}, \mathbf{false}\}$, initially **false**
$dec_u \in \mathcal{V} \cup \{\perp\}$, initially $\perp$

Guarded commands:

$\mathbf{R_u}$ :: {A message $\langle HO \rangle$ has arrived from v}
receive $\langle HO \rangle$
$HO_u \leftarrow HO_u \cup HO$
$rec_u[v] \leftarrow \mathbf{true}$

$\mathbf{S_u}$ :: $\{(\exists v \in N_u,\ \forall w \neq v\ rec_u[w] = \mathbf{true}) \wedge sent_u = \mathbf{false}\}$
send $\langle HO_u \rangle$ to v
$sent_u \leftarrow \mathbf{true}$

$\mathbf{D_u}$ :: $\{\forall v \in N_u\ rec_u[v] = \mathbf{true}$, once$\}$
$dec_u \leftarrow f(HO_u)$

Observe that the Tree algorithm can be used in the general context of an anonymous network.

Theorem 2.2. *The Tree algorithm computes any function of the set of initial values if the communication graph is a bidirectional tree.*

We start by the following lemma, which proves termination.

Lemma 2.1. *In any complete execution of the Tree algorithm on a bidirectional tree, there are at least two deciders.*

Proof. Let α be any complete execution of the Tree algorithm on a bidirectional tree. From the code, we deduce that α is finite; let C be the final configuration in α. Since the communication graph is a bidirectional tree, there are exactly $2(n-1)$ edges in G, where $n = |V|$. Hence, there are $2(n-1)$ bits $rec_u[v]$. Let F_u be the number of $rec_u[v]$ bits that are **false** in C, and let $F = \sum_{u \in V} F_u$. The number of agents u with $F_u = 0$, $F_u = 1$, and $F_u \geqslant 2$ are denoted by ν_0, ν_1, and ν_{2^+}, respectively. Hence, we have

$$n = \nu_0 + \nu_1 + \nu_{2^+}.$$

Finally, let ν_d be the number of agents that have decided in C, and let ν_s be the number of agents that have sent a message in C. Since α is complete, there is no message in transit in C, and thus

$$|E| = 2(n-1) = F + \nu_s.$$

Moreover,

$$\nu_d = \nu_0 \quad \text{and} \quad \nu_s = \nu_0 + \nu_1.$$

Therefore,

$$2(n-1) = F + \nu_s \geqslant \nu_1 + 2\nu_{2^+} + \nu_0 + \nu_1 = 2n - \nu_d.$$

It follows that $\nu_d \geqslant 2$, as required. □

For validity, we first show that if some agent u decides in an execution α of the Tree algorithm, then any agent $v \neq u$ has sent a message previously and the decision by u is a consequence of all these send events.

Lemma 2.2. *If an agent u makes a decision in an execution α of the Tree algorithm, then every agent v different from u sends a message in α and this send event is a cause of the decision event by u.*

Proof. Let us first introduce some notation. Let d_u denote the decision event by u, and if it occurs, let s_v denote the sole send event by v corresponding to the commands $\mathbf{S}_v$. For any non-negative integer δ, let S_δ be the set of agents at distance δ from u, i.e.,

$$S_\delta = \{w \in V \mid d(w,u) = \delta\}.$$

The causality relation in α is denoted by $\prec$ as no confusion may arise.

By induction on $\delta \geqslant 1$, we first prove that if $v \in S_\delta$, then s_v occurs in α, v sends its message to an agent in $S_{\delta-1}$, and $s_v \prec d_u$.

1. Base case $\delta = 1$. Let $v \in S_1$, i.e., v is a neighbor of u. The decision rule of the Tree algorithm enforces the agent u to decide only if it has received a message from all its neighbors, in particular from v. Moreover, it holds that

   $$s_v \prec r_u^v \prec d_u$$

   where r_u^v denotes the receipt corresponding to s_v, i.e., $s_v \to r_u^v$.

2. Suppose that the above claim holds for $\delta \geqslant 1$, and let $v \in S_{\delta+1}$, that is there exists a neighbor w of v that is in S_δ. By the inductive hypothesis, the agent w sends a message to an agent x in $S_{\delta-1}$ and $s_w \prec d_u$. The guard in $\mathbf{S_w}$ implies that

   (a) either w has received a message from all of its neighbors, in particular from v;

   (b) or w has received a message from all of its neighbors but x. Since $x \in S_{\delta-1}$, we have $v \neq x$.

   In both cases, v has sent a message to $w \in S_\delta$ and $s_v \prec s_w \prec d_u$. Since the communication graph is a tree, it is strongly connected, and thus $V = \cup_{\delta=0}^{n} S_\delta$, which completes the proof.

□

Lemma 2.3. *If an agent u decides in a configuration C, then its variable HO_u in C is equal to the set of initial values.*

Proof. Lemma 2.2 shows that

$$\forall v \in V \setminus \{u\}, \quad s_v \prec d_u.$$

The lemma immediately follows from the above causality relations, the contents of the messages exchanged in the algorithm, and the initialization and update rules for the variables HO_v. □

Question 3. *Prove that there are exactly two deciders in any complete execution which are neighboring in the communication graph.*

Answer: We start by showing that 2 deciders are necessarily neighbors. Let u and v be two deciders in an execution α (Lemma 2.1 shows that there is at least 2 deciders in any execution). We proceed by contradiction. Suppose that u and v are not neighbors, i.e., $d(u, v) = \delta \geqslant 2$. Consider w such that $d(u, w) = 1$ and $d(v, w) = \delta - 1 \geqslant 1$. From the proof of Lemma 2.2, we have:

- w has sent a message to u;
- w has sent a message to some agent $x \in S^v_{\delta-2}$.

However, w can send only one message and $x \neq u$ since $u \in S^v_\delta$. We obtain a contradiction. Hence, u and v are neighbors.

Now, consider u, v and w three agents that decides in α. From the previous claim, u, v and w are pairwise neighbors. Hence, there exists a cycle in the communication graph G, which is impossible since G is a tree.

The Tree algorithm (cf. Algorithm 5) computes any function of the *set* of initial values, but does not compute functions of the *multi-set* of initial values. However, the above analysis shows that the variant of the Tree algorithm in Algorithm 6 works for any function of the multi-set of initial values (the symbol "⊔" denotes the adjunction operator of multi-sets).

Question 4.
1. *Explain the role of the $flag_u$ variables.*
2. *Prove that Algorithm 5 does not compute f if f is a function of multi-sets.*
3. *Show that Algorithm 6 copes with functions of multi-sets.*

2.3 The Echo algorithm

In this section, we assume bidirectional links, unique identifiers (non-anonymous network), and a distinguished agent called the *leader*. The *Echo* algorithm is an algorithm with a specific local algorithm for the leader: it spontaneously sends a message to all its neighbors, and then waits for incoming messages. When it has received a message from all its neighbors, the leader may decide.

Any other agent u waits for receiving a first message: upon this first receipt (of a message sent by v), the agent u sends a message to all its neighbors except v. When u has received a message from all its neighbors, it sends a message back to v.

The pseudo-codes for the leader and the non-leaders are given in Algorithm 7 and Algorithm 8, respectively. The key point of this algorithm is that the set of links (u, v) in E such that the first message received by v has been sent by u forms a spanning directed tree, with a root that coincides with the leader.

Theorem 2.3. *The Echo algorithm computes any function of the multi-set of initial values if the communication graph is a bidirectional connected graph.*

Algorithm 6 The Tree algorithm (for a function f of multi-sets)

Initialization:

$HO_u \in \mathbb{M}(\mathcal{V})$, initially $\{\!\{u\}\!\}$ ▷ where elements in $\mathbb{M}(\mathcal{V})$ are multi-sets of values in $\mathcal{V}$
$rec_u \in \{\mathbf{true}, \mathbf{false}\}^{N_u}$, initially $[\mathbf{false}, \ldots, \mathbf{false}]$
$sent_u \in \{\mathbf{true}, \mathbf{false}\}$, initially **false**
$dec_u \in \mathcal{V} \cup \{\perp\}$, initially $\perp$
$flag_u \in \{\mathbf{true}, \mathbf{false}\}$, initially **false**

Guarded commands:

R$_u$ :: {A message $\langle HO, flag\rangle$ has arrived from v}
receive $\langle HO, flag\rangle$
if $flag_u = \mathbf{false}$ **then**
$HO_u \leftarrow HO_u \uplus HO$
else
$HO_u \leftarrow HO$
end if
$rec_u[v] \leftarrow \mathbf{true}$
if $\forall w \in In_u\ rec_u[w] = \mathbf{true}$ **then**
$flag_u \leftarrow \mathbf{true}$
end if

S$_u$ :: $\{(\exists v \in N_u,\ \forall w \neq v\ rec_u[w] = \mathbf{true}) \wedge sent_u = \mathbf{false}\}$
send $\langle HO_u, flag_u\rangle$ to v
$sent_u \leftarrow \mathbf{true}$

D$_u$ :: $\{\forall v \in N_u\ rec_u[v] = \mathbf{true}$, once$\}$
$dec_u \leftarrow f(HO_u)$

Question 5. *Prove Theorem 2.3.*

Question 6. *What can be computed by the Echo algorithm in the absence of identifiers?*

2.4 Tarry's algorithm

Tarry's algorithm is a "sequentialization" of the Echo algorithm, with the use of one virtual token moving over the communication graph according to the following two rules (see Section 1.6 in Chapter 1):

1. an agent never forwards the token twice through the same link;
2. a non-leader agent forwards the token to its *father* (the neighbor from which it first received the token) only if there is no other link possible according to the previous rule.

Therefore, Tarry's algorithm is a traversal algorithm: in every execution, events are totally ordered by the causality relation, and the first and last events in every complete execution are performed by the same agent.

Algorithm 7 The Echo algorithm (leader)

Initialization:

$HO_u \in 2^{V \times \mathcal{V}}$, initially $\{(u, \mu_u)\}$
$rec_u \in \mathcal{V}^{N_u}$, initially $[\textbf{false}, \ldots, \textbf{false}]$
$dec_u \in \mathcal{V} \cup \{\perp\}$, initially $\perp$

Guarded commands:

$\mathbf{R_u}$ :: {A message $\langle HO \rangle$ arrives from v}
receive $\langle HO \rangle$
$HO_u \leftarrow HO_u \cup HO$
$rec_u[v] \leftarrow \textbf{true}$

$\mathbf{S_u}$ :: {**true**, once}
for all $v \in N_u$ **do**
send $\langle HO_u \rangle$ to v
end for

$\mathbf{D_u}$ :: {$\forall v \in N_u \; rec_u[v] = \textbf{true}$, once}
$dec_u \leftarrow f(HO_u[2])$

The pseudo-codes for the leader and the non-leaders are given in Algorithm 9 and Algorithm 10, respectively, in the case where f is a function of the *set* of initial values.

Algorithm 8 The Echo algorithm (non-leader)

Initialization:

$HO_u \in 2^{V \times \mathcal{V}}$, initially $\{(u, \mu_u)\}$
$rec_u \in \mathcal{V}^{N_u}$, initially [**false**, ..., **false**]
$father_u \in V \cup \{\perp\}$, initially $\perp$

Guarded commands:.

$\mathbf{R_u}$ :: {A message $\langle HO \rangle$ arrives from v}
receive $\langle HO \rangle$
$rec_u[v] \leftarrow$ **true**
$HO_u \leftarrow HO_u \cup HO$
if $father_u = \perp$ **then**
$father_u \leftarrow v$
for all $w \in N_u \setminus \{v\}$ **do**
send $\langle HO_u \rangle$ to w
end for
end if

$\mathbf{S_u}$ :: {$\forall v \in N_u$ $rec_u[v] =$ **true**, once}
send $\langle HO_u \rangle$ to $father_u$

Algorithm 9 Tarry's algorithm (the leader)

Initialization:

$sent_u \in \{\textbf{true}, \textbf{false}\}^{N_u}$, initially $[\textbf{false}, \ldots, \textbf{false}]$
$rec_u \in \mathbb{N}$, initially $\mathbf{0}$
$HO_u \in 2^{\mathcal{V}}$, initially $\{\mu_u\}$
$dec_u \in \mathcal{V} \cup \{\perp\}$, initially $\perp$

Guarded commands:

$\mathbf{I_u}$ :: {once}
    choose $v \in N_u$
    send $\langle HO_u \rangle$ to v
    $sent_u[v] \leftarrow \textbf{true}$

$\mathbf{R_u}$ :: {A message $\langle HO \rangle$ arrives from v}
    receive $\langle HO \rangle$
    $rec_u \leftarrow rec_u + 1$
    $HO_u \leftarrow HO_u \cup HO$
    if $\exists v \in N_u,\ sent_u[v] = \textbf{false}$ **then**
        choose v_0 such that $sent_u[v_0] = \textbf{false}$
        send $\langle HO_u \rangle$ to v_0
        $sent_u[v_0] \leftarrow \textbf{true}$
    end if

$\mathbf{D_u}$ :: {$rec_u = |N_u|$, once}
    $dec_u \leftarrow f(HO_u)$

Algorithm 10 Tarry's algorithm (non-leader)

Initialization:

$father_u \in V \cup \{\perp\}$, initially $\perp$
$sent_u \in \{\textbf{true}, \textbf{false}\}^{N_u}$, initially $[\textbf{false}, \ldots, \textbf{false}]$
$HO_u \in 2^{\mathcal{V}}$, initially $\{\mu_u\}$

Guarded commands:

$\mathbf{R_v}$ :: {A message $\langle HO \rangle$ arrives from v}
receive $\langle HO \rangle$
$HO_u \leftarrow HO_u \cup HO$
if $father_u = \perp$ **then**
$father_u \leftarrow v$
end if
if $\exists w \in N_u,\ sent_u[w] = \textbf{false} \wedge w \neq father_u$ **then**
choose w_0 such that $sent_u[w_0] = \textbf{false} \wedge w_0 \neq father_u$
send $\langle HO_u \rangle$ to w_0
$sent_u[w_0] \leftarrow \textbf{true}$
else if $sent_u[father_u] = \textbf{false}$ **then**
send $\langle HO_u \rangle$ to $father_u$
$send_u[father_u] \leftarrow \textbf{true}$
end if

2.5 The Phase algorithm

In the *Phase* algorithm, all agents share the same local algorithm: Each agent repeatedly sends a message containing the collected values to its outgoing neighbors in one shot. The i-th sending is allowed only if the agent has received at least $i-1$ messages from each of its incoming neighbors. An agent may decide when it has received at least D messages, where D is any fixed integer, hence a parameter of the Phase algorithm. The pseudo-code for the Phase algorithm is given in Algorithm 11. The Phase algorithm computes any function of the multi-set of initial values if each agent "knows" an upper bound of the diameter of the communication graph, in a sense that is made precise in the following theorem.

Algorithm 11 The Phase algorithm

Initialization:

$HO_u \in 2^{V \times \mathcal{V}}$, initially $\{(u, \mu_u)\}$
$rcount_u \in \mathbb{N}^{In_u}$, initially $[\mathbf{0}, \ldots, \mathbf{0}]$
$scount_u \in \mathbb{N}$, initially $\mathbf{0}$
$dec_u \in \mathcal{V} \cup \{\perp\}$, initially $\perp$

Guarded commands:

$\mathbf{R_u}$ :: {A message $\langle HO \rangle$ arrives from v}
receive $\langle HO \rangle$
$HO_u \leftarrow HO_u \cup HO$
$rcount_u[v] \leftarrow rcount_u[v] + 1$

$\mathbf{S_u}$ :: $\{\forall v \in In_u\ scount_u \leqslant rcount_u[v] \wedge scount_u < D\}$
for all $w \in Out_u$ **do**
send $\langle HO_u \rangle$ to w
end for
$scount_u \leftarrow scount_u + 1$

$\mathbf{D_u}$ :: $\{\forall v \in In_u\ rcount_u[v] \geqslant D$, once$\}$
$dec_u \leftarrow f(HO_u[2])$

Theorem 2.4. *The Phase algorithm computes any function of the multi-set of initial values if the communication graph G is a strongly connected digraph and the parameter D is at least equal to G's diameter.*

We start the proof of Theorem 2.4 by a series of lemmas on any fixed execution α of the Phase algorithm. Actually, the first lemma is totally general in the sense that it holds for any algorithm.

Lemma 2.4. *Let (u, v) be any link of the communication graph and let $s^{(i)}$ and $r^{(i)}$ denote the i-th send event and the i-th receive event of a message exchanged along this link in α, if exist. Then, if $r^{(i)}$ occurs in α, then $s^{(i)}$ also occurs in α and is a cause of $r^{(i)}$.*

Proof. Let i be any positive integer such that $s^{(i)}$ and $r^{(i)}$ are two events that occur in α. As above, the causality relation in α is just denoted by $\prec$.

First, observe that if the communication channels ensure the FIFO property, we have $s^{(i)} \rightsquigarrow r^{(i)}$ and the lemma trivially holds.

In the general case, consider $m(j)$ such that $s^{(m(j))} \rightsquigarrow r^{(j)}$. Since messages are received exactly once, all the integers $m(1), \cdots, m(i)$ are pairwise distinct and

$$|\{m(j) \mid 1 \leqslant j \leqslant i\}| = i$$

By the pigeon hole principle, there exists $j \in \{1, \ldots, i\}$ such that $i \leqslant m(j)$. Then if the four events $s^{(i)}, s^{(m(j))}, r^{(j)}$, and $r^{(i)}$ occur in α, then they are related by

$$s^{(i)} \preccurlyeq s^{(m(j))} \prec r^{(j)} \preccurlyeq r^{(i)}$$

and thus $s^{(i)} \prec r^{(i)}$. The first relation is because $i \leqslant m(j)$, the second one is by definition of $m(j)$ which leads to $s^{(m(j))} \rightsquigarrow r^{(j)}$, and the last one holds because $j \leqslant i$. The same arguments show that the existence of $r^{(i)}$ implies that of $r^{(j)}$, then that of $s^{(m(j))}$, and finally the existence $s^{(i)}$. □

Lemma 2.5. *If some agent u decides in the execution α (event d_u) and D is an upper bound on the diameter of the communication graph, then any agent v executes at least one send event in α and the first one is a cause of d_u.*

Proof. Since the communication graph $G = (V, E)$ is strongly connected, there exists a path $v = w_0, w_1, \cdots, w_k = u$ from v to u. Moreover, this path may be chosen in such way that its length k is at most equal to G's diameter (geodesic), and so at most equal to D. Let $s^{(i)}_{w_j}$ and $r^{(i)}_{w_j}$ denote the i-th send event and the i-th receive event in α along the links (w_j, w_{j+1}) and (w_{j-1}, w_j), if any. Both are executed by the agent w_j. Then, all the events $s^{(1)}_{w_0}, \cdots, s^{(k)}_{w_{k-1}}$ and $r^{(1)}_{w_1}, \cdots, r^{(k)}_{w_k}, \cdots, r^{(D)}_{w_k}$ occur in α and satisfy

$$s^{(1)}_{w_0} \prec r^{(1)}_{w_1} \prec s^{(2)}_{w_1} \prec \cdots \prec r^{(k-1)}_{w_{k-1}} \prec s^{(k)}_{w_{k-1}} \prec r^{(k)}_{w_k} \preccurlyeq r^{(D)}_{w_k} \prec d_u.$$

The red causality relations are by Lemma 2.4, the blue ones are because of the guard in send commands, the green ones follow from $k \leqslant D$, and the black one is due to the guard in the decision command ($\mathbf{D_u}$). □

We are now in position to prove Theorem 2.4.

Proof. Let α be an execution of the Phase algorithm with a communication graph of finite diameter (strongly connected) at most equal to D.

For the Validity property, assume that some agent u decides in α. Lemma 2.5 shows that every agent v sends at least one message and this first event $s^{(1)}_v$ satisfies $s^{(1)}_v \prec d_u$. From the update rules of the HO variables, it follows that

$$HO_v(s^{(1)}_v) \subseteq HO_u(d_u).$$

Since initially $HO_v = \{(v, \mu_v)\}$ and $HO_u(d_u) \subseteq \{(v, \mu_v) \mid v \in V\}$, it follows that $HO_u(d_u) = \{(v, \mu_v) \mid v \in V\}$, as required.

For Termination, let us assume that α is complete; then by construction of the algorithm, α is finite, and let e denote the last event in α. Let u denote one agent that realizes the minimum of the *scount* counters at the end of α, i.e.,

$$scount_u(e) = \min_{w \in V} scount_w(e).$$

Since α is complete, there is no message in transit in the final configuration of α. In particular, for every incoming neighbor v of u, we have $scount_v(e) = rcount_u[v](e)$, which implies

$$rcount_u[v] \geqslant scount_u.$$

It follows that $scount_u(e) \geqslant D$ because, otherwise, the agent u would be allowed to send a message, and thus α would not be complete. Hence for every u's incoming neighbor v, it holds that $rcount_u[v](e) = scount_v(e) \geqslant D$, and hence u is allowed to decide. Since α is complete, the event d_u occurs in α, which completes the proof of Termination. □

Question 7. *What can be computed by the Phase algorithm in the absence of identifiers?*

2.6 Finn's algorithm

Finn's algorithm is similar to the Gossip algorithm with a more clever decision rule that does not require any agent to know the network size. Nevertheless, like the Gossip algorithm, this algorithm relies on the assumption of unique identifiers. In addition to the variable HO_u containing pairs of the form (v, μ_v), each agent u maintains a variable OK_u in which it collects some identifiers. The agent u repeatedly sends messages of the type $\langle HO_u, OK_u \rangle$. It may decide when $|OK_u| = |HO_u|$. The pseudo-code for Finn's algorithm is given in Algorithm 12.

Theorem 2.5. *The Finn's algorithm computes any function of the multi-set of initial values if the communication graph is strongly connected.*

For the proof of the above theorem, we start with several lemmas stating general properties of an arbitrary execution α of Finn's algorithm. As above, the causality relation in α is simply denoted by "$\prec$" as no confusion may arise.

Lemma 2.6. *For every event e in α and every agent u, it holds that*

$$OK_u(e) \subseteq HO_u[1](e)$$

Proof. The proof is by induction on the sequence of events in α: the base case is a direct consequence of the initializations of HO_u and OK_u, and the inductive step is because of their update rules. □

Lemma 2.7. *If e and e' are two events occurring in α such that $e \prec e'$, then*

$$HO_u(e) \subseteq HO_v(e') \wedge OK_u(e) \subseteq OK_v(e')$$

where $u = agent(e)$ and $v = agent(e')$.

Proof. We consider the following two base cases:

1. $e <_u e'$. The code of the algorithm implies that both the variables HO_u and OK_u may not decrease. We then directly derive the two inclusions of the lemma in that case.
2. $e \rightsquigarrow e'$.[1] The command in $\mathbf{R}_v$ gives $HO_v(e') = HO_u(e) \cup HO_v(e'_-)$ and $OK_v(e') = OK_u(e) \cup OK_v(e'_-)$ where $HO_v(e'_-)$ and $OK_v(e'_-)$ denote the values of the variables HO_v and OK_v in α just before e', respectively.

[1] In fact, e' does not stand for the sole receipt, namely the first instruction in the command $\mathbf{R}_u$, but it stands for the last one. Many thanks to Maxime P. !

Algorithm 12 Finn's algorithm

Initialization:

$dec_u \in \mathcal{V} \cup \{\perp\}$, initially $\perp$
$HO_u \in 2^{V \times \mathcal{V}}$, initially $\{(u, \mu_u)\}$
$OK_u \in 2^V$, initially $\emptyset$
$rec_u \in \{\mathbf{false}, \mathbf{true}\}^{In_u}$, initially $[\mathbf{false}, \ldots, \mathbf{false}]$
$new_u \in \{\mathbf{false}, \mathbf{true}\}$, initially **true**

Guarded commands:

$\mathbf{A_u} ::$ $\{\forall v \in In_u\ rec_u[v] = \mathbf{true}$, once$\}$
$OK_u \leftarrow OK_u \cup \{u\}$
$new_u \leftarrow \mathbf{true}$

$\mathbf{D_u} ::$ $\{|OK_u| = |HO_u|$, once$\}$
$dec_u = f(HO_u[2])$

$\mathbf{S_u} ::$ $\{new_u\}$
for all $v \in Out_u$ **do**
send $\langle HO_u, OK_u \rangle$ to v
end for
$new_u \leftarrow \mathbf{false}$

$\mathbf{R_u} ::$ {A message $\langle HO, OK \rangle$ arrives from v}
receive $\langle HO, OK \rangle$
$rec_u[v] \leftarrow \mathbf{true}$
if $OK \not\subseteq OK_u \vee HO \not\subseteq HO_u$ **then**
$HO_u \leftarrow HO_u \cup HO$
$OK_u \leftarrow OK_u \cup OK$
$new_u \leftarrow \mathbf{true}$
end if

The two inclusions of the lemma in the general case, namely when $e \prec e'$, are obtained by using the definition of the causality relation and the transitivity of this relation and of inclusion. □

For any agent u, we denote a_u the single event in α corresponding to the assignment of OK_u in the guarded command $\mathbf{A}_u$, if executed.

Lemma 2.8. *If u belongs to some variable $OK_v(e)$, then a_u occurs in α and $a_u \prec e$.*

Proof. From Algorithm 12, we obtain that u is the sole agent allowed to add its identifier in OK_u. Moreover, it can do it only when executing the command in $\mathbf{A}_u$. This shows u belongs to some variable OK_v *after* and *only after* a_u in any execution of the algorithm.

Suppose now $\neg(a_u \prec e)$. By Theorem 1.1, there exists an execution α' equivalent to α in which a_u occurs after e in α'. Since α and α' are equivalent, the variables evolve in the same way in the two executions, which contradicts the above claim. □

Proposition 2.1. *Any execution of Finn's algorithm with a communication graph that is strongly connected satisfies the Validity property.*

Proof. Let α be any execution of Finn's algorithm with a communication graph that is strongly connected and in which some agent u makes a decision. Let d_u denote the corresponding event. The proof consists in showing that $HO_u[2](d_u)$ is the multi-set of initial values $\{\!\{\mu_v \mid v \in V\}\!\}$, i.e., $HO_u(d_u)[1] = V$.

Let v be any agent in V. Since the communication graph is strongly connected, there exists a directed path $w_\ell = v, \ldots, w_1 = u$ from v to u. We now show by induction on i, $1 \leqslant i \leqslant \ell$, that a_{w_i} occurs in α with $a_{w_i} \prec d_u$. For any agent w, let f_w the first event executed by w in α, if any.

1. Base case $i = 1$. The identifier u belongs to $HO_u[1]$ from the initialization, i.e., $u \in HO_u[1](f_u)$. By Lemma 2.7, u also belongs to $HO_u[1](d_u)$. The decision rule implies that $HO_u[1](d_u) = OK_u(d_u)$, and by Lemma 2.8, $a_u = a_{w_1}$ occurs in α with $a_{w_1} \prec d_u$.

2. Assume now that a_{w_i} occurs in α with $a_{w_i} \prec d_u$ for some integer i, such that $1 \leqslant i < \ell$. Then we have
$$f_{w_{i+1}} \preccurlyeq s^1_{w_{i+1}} \prec r^1_{w_i} \prec a_{w_i}$$
where $s^1_{w_{i+1}}$ and $r^1_{w_i}$ denote the first send and receive event along the link from w_{i+1} to w_i. The last relation is because of the guard in $\mathbf{A}_{w_i}$, and the previous one is by Lemma 2.4. Since w_{i+1} belongs to $HO_{w_{i+1}}[1](f_{w_{i+1}})$, it also belongs to $HO_{w_{i+1}}[1](s^1_{w_{i+1}})$. By Lemma 2.7, $w_{i+1} \in HO_u[1](d_u)$, and the guard in $\mathbf{D}_u$ enforces $HO_u[1](d_u) = OK_u(d_u)$. Lemma 2.8 shows that $a_{w_{i+1}}$ occurs in α with $a_{w_{i+1}} \prec d_u$, as required.

Therefore,
$$f_{w_\ell} \prec a_{w_{\ell-1}} \prec d_u,$$
and thus $v = w_\ell \in HO_u[1](d_u)$. It follows that $HO_u[1](d_u) = V$, which completes the proof. □

For Termination, observe that any complete execution α of Finn's algorithm is finite. Let us denote the last element in α by e.

Lemma 2.9. *Every agent eventually decides in any complete execution of Finn's algorithm.*

Proof. Since α is complete, each agent v executes the command $\mathbf{S}_u$ at least once. Moreover, there is no message in transit in the final configuration of α. This shows each agent v executes the command $\mathbf{A}_v$, i.e., a_v occurs in α. Hence, $v \in OK_v(e)$.

Let u be any agent in V. Since the communication graph is strongly connected, for every agent v, there exists a directed path $w_\ell = v, \ldots, w_1 = u$ from v to u. The role of the variables new is to guarantee that in the final configuration of any complete execution, we have

$$OK_{w_\ell} \subseteq \cdots \subseteq OK_{w_1}.$$

It follows that $OK_v(e) \subseteq OK_u(e)$. Therefore, $OK_u(e)$ contains the identifier v since $v \in OK_v(e)$, which shows $OK_u(e) = V$.

From the latter equality and Lemma 2.6, we obtain that $HO_u[1](e) = V$. Therefore, u decides in α. □

Chapter 3

Leader election

The problem of *leader election* has been posed by G. Le Lann in 1977, who also proposed the first solution. The problem consists in reaching a configuration where exactly one agent is in the state *leader* and all other agents are in the state *lost*. The non-leaders ought or ought not know the identity of the leader as part of the problem; in any case, an extra phase may be added where the leader broadcasts its identity to the other agents.

A typical situation where election must be held is when a centralized algorithm (e.g., the Echo algorithm) is to be executed and there is no *a priori* candidate to serve as initiator of this algorithm. For example, this is the case when agents may leave or join the system: because the set of agents is unknown in advance, it is not possible to assign one agent once and for all to the role of leader.

Traditionally, the leader election problem has been viewed and studied as a way to understand the effects of symmetry in a distributed system: roughly speaking, a leader election algorithm consists in breaking it down.

A large number of results about the election problem exist. The results in this chapter has been selected for inclusion with the following criteria:

1. The networked system is supposed to be fully asynchronous, and all components (agents and channels) are reliable.
2. The agents are supposed to have unique identifiers (cf. Theorem 3.1).
3. We will concentrate on developing general methods for the design of leader election algorithms, rather than presenting a collection of disparate algorithms. In particular, we explain how to derive solutions from the algorithms studied in Chapter 2.

3.1 Preliminaries

Each agent u is supposed to have an output variable $state_u$ initially equal to *sleep*.

Definition 3.1. A leader election algorithm is an algorithm that satisfies the following properties:

1. Each agent has the same local algorithm.
2. The algorithm is decentralized, i.e., any non-empty subset of agents may initiate[1] an execution of the algorithm.

[1] An agent u is an initiator of an execution if the first event by u in this execution is not a receipt.

3. In every complete execution of the algorithm, there is exactly one leader u and the others are non-leader, i.e.,

$$\exists u \in V, \quad (state_u = leader) \ \wedge \ (\forall v \in V \setminus \{u\}, \ state_v = lost).$$

The algorithms that meet the first condition are said to be *symmetric*.

Question 8. *Consider the algorithm in which each agent has the code in Algorithm 13. Is it a leader election algorithm?*

Algorithm 13

Initialization:
$state_u \in \{\textbf{passive}, \textbf{active}\}$, initially **passive**
$leader_u \in \{\textbf{yes}, \textbf{no}, \perp\}$, initially $\perp$

guarded commands:
$\mathbf{I_u} :: \{state_u = passive\}$
$state_u \leftarrow$ **active**
if $u = 1$ **then**
$leader_u \leftarrow$ **yes**
else
$leader_u \leftarrow$ **no**
end if

In this chapter, we study the leader election problem under the following assumptions:

1. The system is supposed to be fully asynchronous.
2. Every component, to be an agent or a channel, is reliable.
3. Each agent is identified by a unique name, its identity, and it may use it in its local algorithm. The identities are drawn from the total ordered set $[N] = \{1, \dots, N\}$. For simplicity, we assume that the identity of the agent u is u, i.e., $V \subseteq [N]$.

3.1.1 Leader election in anonymous networks

The following theorem states that there is no election algorithm for anonymous networks, i.e., when agents do not have an identifier. The proof uses the symmetry (of the network and of the local algorithms) to build an infinite execution in which no agent may declare leader.

Theorem 3.1. *The leader election problem is not solvable on an anonymous ring, even if the ring size is known.*

Proof. The proof is by contradiction: suppose that $\mathcal{A}$ is a leader election algorithm for such a system. We inductively construct an infinite execution of $\mathcal{A}$, denoted $\alpha = (C_0, C_1, \dots, C_t, \dots)$ such that every configuration C_{kn}, with $n = |V|$, is *symmetric*, i.e., satisfies

$$\forall u, v \in V, \quad C_{kn}[u] = C_{kn}[v] \ \wedge \ C_{kn}[M.u] = C_{kn}[M.v], \tag{3.1}$$

where $C_t[u]$ denotes u's state in the configuration C_t and $C_t[M.u]$ denotes the multiset of messages in transit to u in C_t. Hence, no leader is elected in any configuration C_{kn}, which leads to a contradiction.

The initial configuration C_0 is symmetric since agents have no identifiers and there is no message in transit. Suppose that the configuration $C = C_{kn}$ is symmetric. Since the algorithm $\mathcal{A}$ is also symmetric and all the agents have the same incoming neighborhood, each agent u can execute the same local transition in C: if e_u denotes the corresponding event, and $C^u = e_u \cdot C$, then we have

$$\forall u, v \quad (C^u[u] = C^v[v]) \ \wedge \ (C^u[M.u] = C^v[M.v]).$$

Lemma 1.3 (Diamond Lemma) shows that e_u is applicable to C^v, e_v is applicable to C^u and $e_u \cdot C^v = e_v \cdot C^u$. By applying the events $e_1, \ldots, e_n$ successively to C, we obtain a new configuration $C' = C_{(k+1)n}$ such that

$$\forall u, v \quad C'[u] = C'[v] \wedge C'[M.u] = C'[M.v].$$

□

Observe that Theorem 3.1 also holds for arbitrary communication graphs in which all the nodes have the same incoming neighborhood: for instance, it holds for bidirectional rings or for a complete communication graph. Hence, to break symmetry, one can take advantage of some asymmetry in the communication graph or, more simply, one may assume unique identifiers. This is precisely the role of the above third assumption. Under this assumption, the strategy may then consists in electing the agent with the smallest (or greatest) identity. In this case, the leader election problem is referred to as the *extra-finding* problem.

3.2 The Le Lann algorithm

In the Le Lann algorithm, given in Algorithm 14, each initiator computes the set of identifiers of all the initiators and registers as leader or non-leader depending on its own identifier being the minimum of the computed set. This algorithm assumes that the topology of the network is an oriented ring with communication channels enforcing the FIFO property.

3.3 Extinction

As agents have unique identifiers, it is possible to use a wave algorithm, like those given in Chapter 2, to perform an election. In short, it consists in using the wave algorithm to compute the set of all identifiers; the elected leader is then the agent with the smallest identifier. However, the chosen wave algorithm must meet the requirements given in the introduction of the chapter: it must be decentralized and may tolerate any subset of agents as initiators.

Given these preconditions, it is clear that the Echo algorithm (see Section 2.3 in Chapter 2) cannot be used directly for leader election, as it is centralised. In a similar way, the Tree algorithm (see Section 2.2) is not a good candidate either: the initiators of the Tree algorithm are the leaves of the network. Hence, it cannot tolerate any subset of agents as initiators.

The extinction principle applies to wave algorithms that are centralized (i.e., such that there is only one initiator) and such that only the initiator decides. An example of such an algorithm is the Echo algorithm, as introduced in Section 2.3 of Chapter 2.

In short, the extinction consists in applying the following rules:

- Each initiator starts a wave, using its own identifier to uniquely identify the wave.

Algorithm 14 The Lelann Algorithm

Variables:

$state_u \in \{\textbf{sleep}, \textbf{cand}, \textbf{lost}, \textbf{leader}\}$, initially **sleep**
$S_u \subseteq V$, initially $\emptyset$

Guarded Commands:

I$_u$ :: $\{state_u = \textbf{sleep}\}$
send $\langle \textbf{tok}, u \rangle$ to $next_u$
$S_u \leftarrow \{u\}$
$state_u \leftarrow \textbf{cand}$

R$_u$ :: {A message $\langle \textbf{tok}, v \rangle$ arrives}
receive $\langle \textbf{tok}, v \rangle$
if $state_u \in \{\textbf{sleep}, \textbf{lost}\}$ **then**
$state_u \leftarrow \textbf{lost}$
send $\langle \textbf{tok}, v \rangle$ to $next_u$
else if $v = u$ **then**
if $u = \min(S_u)$ **then**
$state_u \leftarrow \textbf{leader}$
else
$state_u \leftarrow \textbf{lost}$
end if
else
$S_u \leftarrow S_u \cup \{v\}$
send $\langle \textbf{tok}, v \rangle$ to $next_u$
end if

- When an initiator u receives a message belonging to a wave with an identifier greater than its own, u drops it. Otherwise, u behaves accordingly to the wave algorithm.
- When an initiator u decides in the wave it started, it proclaims itself as the leader.

3.3.1 The Chang-Roberts Algorithm

The Chang-Roberts algorithm (see Algorithm 16) can be obtained by applying the extinction principle to the Ring algorithm, given in Algorithm 15. The Ring algorithm applies when the communication graph is an oriented ring; each agent u has a parameter $next_u$ denoting the successor of u in the ring. It is a centralised algorithm: one agent, the initiator, sends a message $\langle \mathbf{tok} \rangle$ to its successor. Any other agents (the so-called non-initiators) awaits for receiving the message and then forwards it to their successor. The algorithm terminates when the initiator receives $\langle \mathbf{tok} \rangle$.

Algorithm 15 The Ring algorithm

Variables:

$dec_u \in \{\mathbf{false}, \mathbf{true}\}$, initially **false**

Guarded Commands:

$\mathbf{I_u}$ :: $\{dec_u = \mathbf{false}$, once$\}$
send $\langle \mathbf{tok} \rangle$ to $next_u$

$\mathbf{D_u}$ :: {A message $\langle \mathbf{tok} \rangle$ arrives }
receive $\langle \mathbf{tok} \rangle$
$dec_u \leftarrow \mathbf{true}$

(15.a) If u is the initiator

Guarded Commands:

$\mathbf{R_v}$ :: {A message $\langle \mathbf{tok} \rangle$ arrives}
receive $\langle \mathbf{tok} \rangle$
send $\langle \mathbf{tok} \rangle$ to $next_v$

(15.b) If v is not the initiator

An initiator u of the Chang-Roberts algorithm starts an instance of the Ring algorithm by sending a message containing its identifier to its successor. When receiving a message, agents obey the following rules:

- An initiator receiving a message with an identifier greater than its own drops it.
- An initiator receiving a message containing its own identifier proclaims itself as the leader.

The pseudo-code for the Chang-Roberts algorithm is given in Algorithm 16.

Correction of the Chang-Roberts algorithm To prove the Chang-Roberts algorithm correct, we consider α a complete execution of the algorithm and w the initiator in α with the smallest identity. We denote the sequence of agents on the oriented ring as $u_1, \ldots, u_n$, with $u_1 = w$. We then start by proving a few lemmas.

Lemma 3.1. *For all configurations in α, $state_{u_1} \neq \mathbf{lost}$.*

Proof. As $u_1 = w$ is an initiator, its first event in α corresponds to $\mathbf{I_w}$. After that, $state_w = \mathbf{cand}$ and, as w has the smallest identity among initiators, it can only change to the value **leader** (see the pseudo-code for $\mathbf{R_w}$ in Algorithm 16). □

Algorithm 16 The Chang-Roberts algorithm

Variables:

$init_u \in \{\mathbf{false}, \mathbf{true}\}$, initially **true** iff u is an initiator
$state_u \in \{\mathbf{sleep}, \mathbf{cand}, \mathbf{lost}, \mathbf{leader}\}$, initially **sleep**

Guarded commands:

$\mathbf{I_u}$ :: $\{init_u = \mathbf{true} \wedge state_u = \mathbf{sleep}\}$
  $state_u \leftarrow \mathbf{cand}$
  send $\langle \mathbf{tok}, u \rangle$ to $next_u$

$\mathbf{R_u}$ :: {a message $\langle \mathbf{tok}, v \rangle$ arrives}
  receive $\langle \mathbf{tok}, v \rangle$
  if $state_u \in \{\mathbf{sleep}, \mathbf{lost}\} \vee v < u$ **then**
    $state_u \leftarrow \mathbf{lost}$
    send $\langle \mathbf{tok}, v \rangle$ to $next_u$
  else if $v = u$ **then**
    $state_u \leftarrow \mathbf{leader}$
  end if

Lemma 3.2. *If v is an initiator such that $v \neq w$ then v never receives the message $\langle \mathbf{tok}, v \rangle$.*

Proof. Let $u_i = v$ (so $i \neq 1$) and let r_j (resp. s_j) denotes the event where u_j receives (resp. sends) the message $\langle \mathbf{tok}, v \rangle$. From the pseudo-code, if r_i occurs in α, we have:

$$r_i \succ s_{i-1} \succ r_{i-1} \succ \cdots \succ s_1 \succ r_1$$

Yet, $state_{u_1}$ just before r_1 is not **lost** (from Lemma 3.1). Thus, $w = u_1$ never sends a message $\langle \mathbf{tok}, v \rangle$, i.e., the event s_1 does not occur in α. Hence, r_i does not occur in α. □

Lemma 3.3. *For all i such that $1 \leqslant i \leqslant n$, u_i receives the message $\langle \mathbf{tok}, u_1 \rangle$.*

Proof. Consider i such that $2 \leqslant i \leqslant n-1$ and u_i receives $\langle \mathbf{tok}, u_1 \rangle$. Following the pseudo-code of $\mathbf{R_{u_i}}$, if u_i is not an initiator or has lost the election, the message is forwarded to u_{i+1}. Otherwise, u_i is an initiator and $u_1 < u_i$ by definition, so the message is also forwarded. In both cases, u_{i+1} receives $\langle \mathbf{tok}, u_1 \rangle$. As u_1 is an initiator, its first event in α is sending a message to u_2, so u_2 also receives $\langle \mathbf{tok}, u_1 \rangle$. Finally, u_n receives $\langle \mathbf{tok}, u_1 \rangle$ and as stated precedently, it forwards the message to its successor in the ring, namely u_1, so u_1 also receives $\langle \mathbf{tok}, u_1 \rangle$. □

Theorem 3.2. *The Chang-Roberts algorithm is correct if the graph of communications is an oriented ring.*

Proof. Let α be a complete execution and $v \in V$.

1. If v is not an initiator, then $state_v = \mathbf{lost}$ after the first receipt (direct from the pseudo-code).

2. Otherwise, v is a initiator.

- If $v \neq w$, then v receives the message $\langle \mathbf{tok}, u_1 \rangle$ (from Lemma 3.3). Before the receipt, $state_v$ is either **lost** or **cand** as v never receives the message $\langle \mathbf{tok}, v \rangle$ (by Lemma 3.2). Hence $state_v = \mathbf{lost}$ after receiving $\langle \mathbf{tok}, u_1 \rangle$.
- If $v = u_1$ then, from Lemma 3.3, v receives $\langle \mathbf{tok}, u_1 \rangle$

□

3.4 The Itai-Rodeh algorithm

The Itai-Rodeh algorithm is a randomized algorithm that solves the leader election problem in an anonymous oriented ring, and terminates with probability 1. All agents execute the same local algorithm, which is parametrized by two integers n and N such that n is the size of the network and $N \geqslant n$. The base principle of the algorithm is that each initiator picks a random number in the set $\{1, \ldots, N\}$ as its identifier and then runs the Chang-Roberts algorithm (cf. Algorithm 16).

Obviously, it may happen that two distinct nodes u and v choose the same identifier k. Two problems may then arise: First, following Chang-Roberts, it may happen that v receives a message $\langle \mathbf{tok}, k \rangle$ sent initially by u, and wrongly elects itself as the leader. This problem can be circumvented by attaching a counter *hops* to each token that counts the number of times the token is transmitted. In this way, u receives its own token when the hop counter equals n.

However, in the case where k is minimal, both u and v are elected as leaders. To detect this situation, each token also carries a boolean value *unique*, which is **true** at the moment the token is generated, but set to **false** if it is forwarded by an agent with the same identifier. An agent can thus become leader when it receives its own token ($hops = n$) with $unique = \mathbf{true}$.

The above sketched scheme may not terminate when the minimal identifier is selected by two distinct agents. If so, each agent with the minimal identifier starts a new Chang-Roberts algorithm, with a new identifier and at a higher level: at the next level, a new identifier is chosen randomly in $\{1, \ldots, N\}$ by each agent that initiates the new level. The level is also indicated in the token, and tokens of some level abort all activity of smaller levels. The complete pseudo-code for the Itai-Rodeh algorithm is given in Algorithm 17.

Proposition 3.1. *In every execution of the Itai-Rodeh algorithm, there is at most one leader.*

Proof. Let α be an execution of the Itai-Rodeh algorithm. For any positive integer ℓ, let S_ℓ denote the set of agents that generate a token at level ℓ in α. We first prove the following two lemmas.

Lemma 3.4. *For any positive integer ℓ, $S_{\ell+1} \subseteq S_\ell$.*

Proof. An agent may generate a token at level $\ell + 1$ only if it receives its own token at level ℓ. The lemma immediately follows. □

Lemma 3.5. *If a token with level ℓ is generated in the execution α, then either exactly one agent becomes a leader at that level and no token at level $\ell + 1$ is generated, or there is at least one agent that generates a token at level $\ell + 1$.*

Proof. Suppose that $S_\ell \neq \emptyset$ and $S_{\ell+1} = \emptyset$, and let $S_\ell^{\min} \neq \emptyset$ the subset of agents in S_ℓ with minimal identifier (at level ℓ). Since $S_{\ell+1} = \emptyset$, Lemma 3.4 shows that $S_{\ell'} = \emptyset$ for any integer $\ell' > \ell$. It follows that the tokens generated by any agent u at level ℓ cannot be purged by an agent at a higher level. Hence, the token generated by $u \in S_\ell^{\min}$ returns to u. Two cases then arise: either $S_\ell^{\min} = \{u\}$, or $\left|S_\ell^{\min}\right| \geqslant 2$. In the first case, the token returns with $unique = \mathbf{true}$,

Algorithm 17 The Itai-Rodeh algorithm

Initialization:

$state_u \in \{\textbf{passive}, \textbf{active}\}$, initially **passive**
$leader_u \in \{\textbf{yes}, \textbf{no}, \perp\}$, initially $\perp$
$id_u \in \{1, \ldots, N\}$
$level_u \in \mathbb{N}$, initially 0

Guarded commands:

$\mathbf{I_u}$ :: $\{state_u = \textbf{passive}\}$
$state_u \leftarrow \textbf{active}$
$id_u \leftarrow rand(\{1, \ldots, N\})$
$level_u \leftarrow 1$
send $\langle \textbf{tok}, id_u, 1, \textbf{true}, level_u \rangle$ to $next_u$

$\mathbf{R_u}$ :: {a message $\langle \textbf{tok}, id, hops, unique, \ell \rangle$ arrives}
receive $\langle \textbf{tok}, id, hops, un, \ell \rangle$
if $(state_u = \textbf{passive}) \vee (leader_u = \textbf{no}) \vee (level_u < \ell) \vee (level_u = \ell \wedge id < id_u)$ **then**
$leader_u \leftarrow \textbf{no}$
send $\langle \textbf{tok}, id, hops + 1, unique, level \rangle$ to $next_u$
else if $(level_u = \ell) \wedge (id_u = id)$ **then**
if $hops < n$ **then**
send $\langle \textbf{tok}, id, hops + 1, \textbf{false}, \ell \rangle$ to $next_u$
else
if $unique = \textbf{true}$ **then**
$leader_u \leftarrow \textbf{yes}$
else
$id_u \leftarrow rand(\{1, \ldots, N\})$
$level_u \leftarrow level_u + 1$
send $\langle \textbf{tok}, id_u, 1, \textbf{true}, level_u \rangle$ to $next_u$
end if
end if
end if

and u is elected. In the second case, the token returns with *unique* = **false**, and $S_\ell^{\min} = S_{\ell+1}$, a contradiction. □

We can now conclude the proof of Proposition 3.1. Assume that the agents u and v are elected as leaders at level ℓ_u and ℓ_v, respectively. Then $S_{\ell_u} \neq \emptyset$ and $S_{\ell_v} \neq \emptyset$. Lemma 3.5 implies that $\ell_u = \ell_v$, and then $u = v$. □

Proposition 3.2. *The Itai-Rodeh algorithm terminates with probability* 1.

Proof. Let us fix the parameter N with $N \geqslant n$. Let p_k be the probability that among k agents that choose an identifier in the set $\{1, \ldots, N\}$, exactly one agent obtains the minimal identifier. In particular, $p_1 = 1$ and $p_k > p_{k+1}$. We set

$$p = \min_{k \in [n]} p_k = p_n.$$

Since $N \geqslant n$, we have $p > 0$.

If a token is generated at level ℓ, then Lemma 3.5 implies that the probability not to elect a leader at level ℓ is less than $1 - p < 1$. Hence the probability not to have elected a leader by level ℓ is less than $(1-p)^\ell$, and the result follows from $\lim_{\ell \mapsto \infty}(1-p)^\ell = 0$. □

The probabilistic analysis by Itai and Rodeh reveals that the expected number of levels is bounded by $eN(N-1)$. Moreover, the expected complexity of the algorithm is $O(N \log N)$ messages.

Chapter 5

Detection of distributed termination

A computation of a distributed algorithm *terminates* when the algorithm has reached a *terminal configuration*, that is a configuration in which no further steps are applicable. At that point, it is important to distinguish the fact that an agent has reached a *terminal state* in the algorithm (e.g., by making a decision) from the one in which it does not participate to the algorithm. Indeed, upon the receipt of a message, an agent in a terminal state may be required to participate again in the algorithm (e.g., to allow other agents to enter a terminal state). If the agents in terminal states are said to be *passive*, then termination corresponds to the global state in which all agents are passive and there is no message in transit. As an example, the Tree algorithm actually terminates when every agent has sent one message (in which case it is declared as passive) and all messages have been received.

Detecting termination is crucial for debugging and managing distributed systems. Only after termination can the variables of a distributed algorithm be discarded, demonstrating the relationship between termination and distributed garbage collection. Also, a *deadlock* of a distributed algorithm results in a terminal configuration, and detecting deadlocks amounts to detecting termination.

The point is thus to design a distributed algorithm on the top of the base algorithm that *correctly* detects its termination: There is no false detection (safety) and termination is eventually detected (liveness). The algorithm for termination detection is a *control algorithm* that is superimposed on the *base algorithm*: it must just observe the base algorithm and may not interfere with it. For instance, it must not stop the activity in the base algorithm. Finally, termination may be detected by all the agents or just by a subset of agents (e.g., one single agent).

As above exemplified, the semantics of termination depends on the semantics of the *passive* states. Formally, the problem is specified in an abstract way as follows:

R1 Each agent is either active or passive.

R2 Only an active agent can send a message.

R3 Each active agent can become passive spontaneously.

R4 A passive agent can become active only upon the receipt of a message.

Termination is achieved when all the agents are passive and all the communication channels are empty. Because of the rules R1-4, termination is a stable property.

Algorithm 18 The (dummy) base algorithm

State variables for agent u
$state_u \in \{\textbf{active}, \textbf{passive}\}$

$\mathbf{S_b} ::$ $\{state_u = \textbf{active}\}$
Send $\langle \textbf{msg} \rangle$ to some agent q

$\mathbf{R_b} ::$ {A base message $\langle \textbf{msg} \rangle$ arrives from v}
Receive $\langle \textbf{msg} \rangle$
$state_u \leftarrow \textbf{active}$

$\mathbf{I_b} ::$ $\{state_u = \textbf{active}\}$
$state_u \leftarrow \textbf{passive}$

- In the case of synchronous communications, the predicate on communication channels is trivially verified. Thus, termination can be detected if all agents are in the passive state.
- We consider the *atomic model*, where sequences of base events are collapsed into one event. Thus, in a single event, a passive agent receives a base message, becomes active, performs some computation, eventually sends some messages to its neighbors, and then become passive again. With this restriction, agents are always passive and termination can be detected when the communication channels are empty.

5.1 The 4-counter algorithm

The 4-counter algorithm applies in the atomic model. Its main principle is to count the messages being sent and received along the communication channels to spot messages in transit. It also assumes the availability of a ring substructure $\{u_n\}_{0 \leqslant n < N}$ such that $\{u_n\}_{0 \leqslant n < N} = V$ and, forall k such that $0 \leqslant k < N$, there is a communication channel from $u_{(k+1) mod N}$ to u_k.

Each agent u holds two state variables r_u and s_u, for counting the messages received and sent by u, respectively. The initiator starts a wave on the ring substructure to compute the accumulated sum of the counter s_u and r_u, by sending a token with two counters. When an agent u receives the token, it adds the values r_u and s_u and forwards the token to the next agent in the ring. One wave compute two values $R^\star$ and $S^\star$, and defines a cut in the execution. Unfortunately, a cut may be inconsistent as $R^\star$ and $S^\star$ can not discriminate messages. This is illustrated in Figure 5.1: At time t_0, the token has visited all the agents and $R^\star = S^\star$, but termination does not hold. Indeed, the token has recorded the sending of message m_1 by w but not its receipt by v, while it recorded the receipt of message m_2 by w but not its sending by v.

Therefore, stating $R^\star = S^\star$ is not sufficient to detect termination. Using two consecutive *non overlapping* waves, we obtain four counters $R^\star$, $S^\star$ (from the first wave), $R'^\star$ and $S'^\star$ (from the second wave). The termination can now be detected when $R^\star = S'^\star$.

Lemma 5.1. *There is no false detection.*

Proof. We denote $S^{(t)}$ (resp. $R^{(t)}$) the total number of messages sent (resp. received) at time t. Observe that we have $\forall t \ R^{(t)} \leqslant S^{(t)}$ from causality, and that S (and R) is increasing: $\forall t, t' \ t \leqslant t' \implies S^{(t)} \leqslant S^{(t')}$.

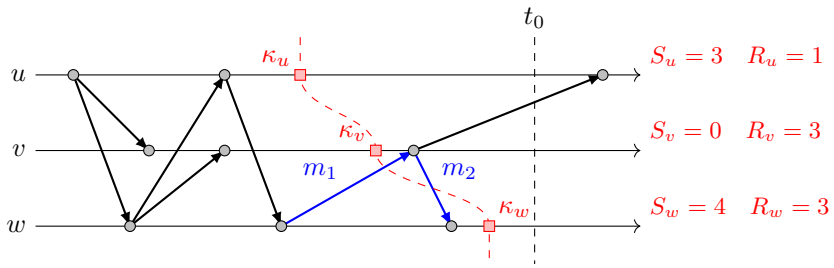


Figure 5.1: Counter-example in the case of 2 counters

Consider t_1 and t_2 (resp. t_3 and t_4) the begin and end time of the first (resp. second wave). Note that, since two waves are distinct, we have: $t_2 < t_3$. Moreover, by definition of the algorithm, we have $R^\star \leqslant R^{(t_2)}$ and $S^{(t_3)} \leqslant S'^\star$. Putting it all together, we obtain:

$$R^\star \leqslant R^{(t_2)} \leqslant S^{(t_2)} \leqslant S^{(t_3)} \leqslant S'^\star$$

Hence, from $R^\star = S'^\star$, we can deduce that $R^{(t_2)} = S^{(t_2)}$. Then, at time t_2 the communication channels are empty, which, in the atomic model, implies termination.

□

Lemma 5.2. *Termination is always detected.*

Proof. Suppose that the base algorithm is terminated at time t. Then we have: $R^{(t)} = S^{(t)}$ and $\forall t, t'\ S^{(t')} = S^{(t)}$. So any two consecutive waves started after t will compute 4 values such that $S^\star = R^\star = S'^\star = R'^\star$.

□

5.2 The Dijkstra-Feijen-Van Gasteren algorithm

The Dijkstra-Feijen-Van Gasteren algorithm can be used to detect the termination of a centralised base computation. It applies in the context of synchronous communications, and assumes the existence of a spanning oriented ring substructure as in Section 5.1. As communications are synchronous, the communication channels are empty at all times. The termination can then be detected when all agents are passive.

The main principle of the algorithm is to forward a token along the underlying ring structure. All agents holds a color attribute (**red** or **yellow**) that relates to its status. The token also gets a color that relates to the state of the visited agents. The initiator starts a wave by sending a yellow token to its neighbor in the ring. The rules for forwarding the token and updating the color attributes are the following:

R1 The token is initially held by the initiator u_0.

R2 Only a passive agent can forward the token.

R3 When agent u_i sends a message to a some agent u_j such that $i < j$, it changes its color attribute to **red**.

R4 A red agent forwarding the token sets the token color to **red** and sets its own color to **yellow**.

R5 When the detection fails, the initiator starts a new wave.

Rule R3 comes from the fact that a passive agent visited by the token may be turned into the active state by receiving a message from an active agent. This situation is depicted in Figure 5.2: while the token is held by agent u_k, agent u_i sends a message to u_j, where $j < k < j$.

The termination is detected when: the token is back at u_0; the token is yellow; u_0 is yellow and passive.

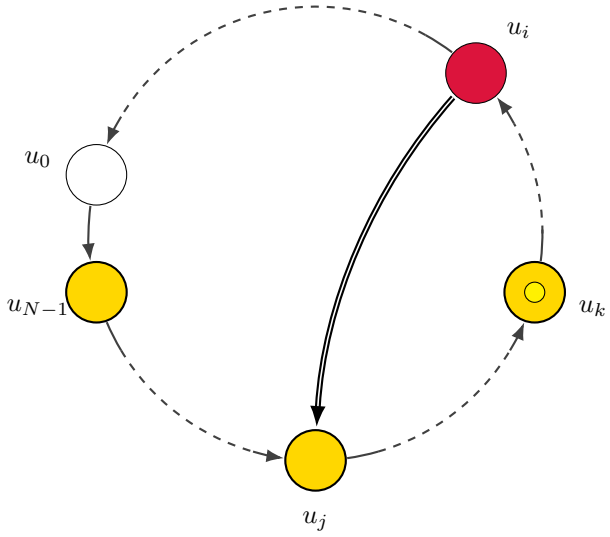


Figure 5.2: DFG: Illustration of R3

Question 9. *Prove that the Dijkstra-Feijen-Van Gasteren algorithm is safe, i.e., there is no false detection.*

Question 10. *Show that if the system terminates during the i-th wave, then it is detected no later than the i+2-th wave.*

5.3 The Djikstra-Scholten algorithm

The Dijkstra-Scholten algorithm assumes *diffusive computations*: all but one of the agent are initially passive. The sole initially active agent is denoted by e. Moreover, the communication graph is supposed to be bidirectional and (strongly) connected.

The basic idea of the algorithm is to maintain a dynamic tree whose vertices are the agents that either are active or do not know whether the activity they induced has terminated. As a consequence, termination will be detected by the sole agent e from which activity results. Hence the agent e is the unique initiator of the Dijkstra-Scholten algorithm (leader), and the unique agent in charge of termination detection.

The mechanism for an agent to detect that the activity it triggered has terminated is the use of acknowledgements (control messages). Each agent u maintains two control variables, namely C_u and $father_u$: C_u is a local counter that u increments each time it sends a (base) message,

and that it decrements upon the receipt of an acknowledgement. It follows that u knows that the activity it triggered terminates when $C_u = 0$.

Upon the receipt of a message from v, the agent u, if passive, declares v as its father or sends an acknowledgement to v otherwise. The message that made it active comes from its father, and u acknowledges this message only when it becomes passive and $C_u = 0$. The pseudo-code for u is given below (Algorithm 19).

Let us consider any finite execution α of the algorithm, and let C be the final configuration. The correctness proof of the Dijkstra-Scholten algorithm relies on the properties of a directed tree defined with respect to C. We start with a series of preliminary lemmas about the values of the variables C_u and $father_u$ in C.

Lemma 5.3. *If $father_u = \perp$, then $C_u = 0$ and u is passive.*

Proof. This is a direct consequence of the initializations of $father_u$ and C_u, the fact that all agents other than e are initially passive, and the guarded commands S_u, R_u, I_u, and RA_u. □

Lemma 5.4. *The value of the counter C_u is equal to the total number of agents whose father is u plus the number of messages sent by u and in transit in C, plus the number of acknowledgements sent to u and in transit in C.*

Proof. The property on C_u holds initially. Moreover, it is invariant under the guarded commands S_u, R_u, I_u, and RA_u. □

Lemma 5.5. *If $father_u = v$, then $father_v \neq \perp$.*

Proof. The proof is by contradiction: assume that $father_v = \perp$. Lemma 5.3 shows that $C_v = 0$. By Lemma 5.4, it follows that there is no agent whose father is v, a contradiction. □

The same argument as above proves the following lemma.

Lemma 5.6. *If u has sent a base message that is in transit in C or if it is the receiver of an acknowledgement that is in transit in C, then $father_u \neq \perp$.*

Lemmas 5.5 and 5.6 lead to define the directed graph $T = (V_T, E_T)$, where:

$$\begin{aligned} V_T &= \{u \in V \mid father_u \neq \perp\} \cup \{msg \in \mathcal{M} \mid msg \text{ in transit in } C\} \\ &\quad \cup \{ack \mid ack \text{ in transit in } C\} \end{aligned}$$

$$\begin{aligned} E_T &= \{(father_v, v) \mid v \in V_T \cap V\} \cup \{(v, msg) \mid msg \text{ in transit in } C \text{ and sent by } v\} \\ &\quad \cup \{(v, ack) \mid ack \text{ in transit in } C \text{ and sent to } v\} \end{aligned}$$

Moreover, every edge of the form $(father_v, v)$ corresponds to an event in α associated to the guarded command $\mathbf{R_v}$; if this event is the t-th event in α, then the edge is labelled by t.

As an immediate consequence of the definition of E_T, we obtain the following Lemma.

Lemma 5.7. *Every node in T that is not an agent has no outgoing edges.*

Proof. By definition of V_T and E_T. □

Lemma 5.8. *Let u, v, and w be three agents. If (u, v) and (v, w) are two distinct edges in T labelled by t and t', respectively , then $t' > t$.*

Algorithm 19 The Dijkstra-Scholten algorithm

Initialization:

$state_u \in \{\textbf{active}, \textbf{passive}\}$, initially **passive** if $u \neq e$, $state_e$ initially **active**
$father_u \in V \cup \{\perp\}$, initially $\perp$ if $u \neq e$, $father_e$ initially e
$C_u \in \mathbb{N}$, initially 0

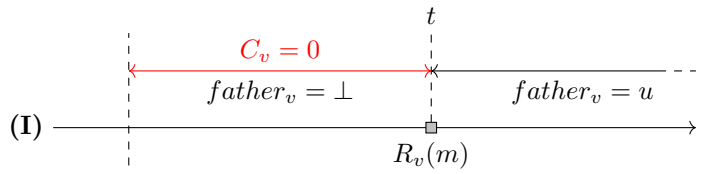
Guarded commands:

S$_u$ :: $\{state_u = \textbf{active}\}$
send $\langle \textbf{msg} \rangle$ to some agent v
$C_u \leftarrow C_u + 1$

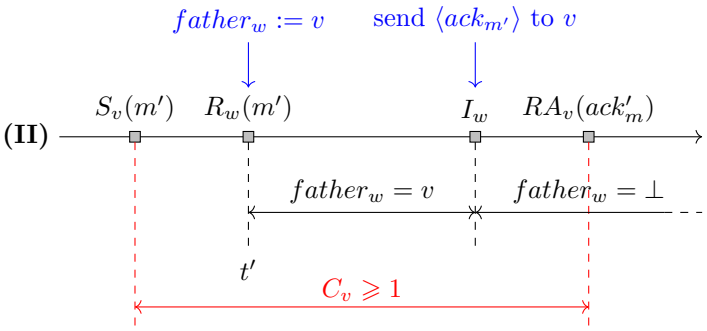
R$_u$ :: {A base message $\langle \textbf{msg} \rangle$ arrives from v}
receive $\langle \textbf{msg} \rangle$
$state_u \leftarrow \textbf{active}$
if $father_u = \perp$ **then**
$father_u \leftarrow v$
else
send $\langle \textbf{ack} \rangle$ to v
end if

I$_u$ :: $\{state_u = \textbf{active}\}$
$state_u \leftarrow \textbf{passive}$
if $C_u = 0$ **then**
if $father_u \neq u$ **then**
send $\langle \textbf{ack} \rangle$ to $father_u$
else
announce termination
end if
$father_u \leftarrow \perp$
end if

RA$_u$ :: {A message $\langle \textbf{ack} \rangle$ arrives}
receive $\langle \textbf{ack} \rangle$
$C_u \leftarrow C_u - 1$
if $C_u = 0$ and $state_u = \textbf{passive}$ **then**
if $father_u \neq u$ **then**
send $\langle \textbf{ack} \rangle$ to $father_u$
else
announce termination
end if
$father_u \leftarrow \perp$
end if



(a) Edge t



(b) Edge t'

Figure 5.3: Sequential construction of T

Proof. The proof is by contradiction: let us assume $t' \leqslant t$.
If $t = t'$, then the two edges correspond to the same event and thus are equal. It follows that $t' < t$.
From the code in the guarded command $\mathbf{R_v}$, it follows that just before t in α, the agent v has no father. Lemma 5.3 shows that then v is passive and the counter $C_v = 0$ (cf. Figure 5.3a).

The t'-th event in α corresponding to the receipt of a base message m' sent by v to w is preceded by the sending of m' since α is a linear extension of the causality relation; let $S_v(m')$ denote the event containing this sending. The code of the algorithm implies that $C_v \geqslant 1$ in a period starting at $S_v(m')$. This period is finite since $t' < t$ and because of the values of the counter C_v. It may end only upon the receipt of an acknowledgement $ack_{m'}$ sent by w to v; let $I_w(ack_{m'})$ and $RA_v(ack_{m'})$ the two events in α containing the sending and the receipt of $ack_{m'}$, respectively (cf. Figure 5.3b). The code of the command $\mathcal{I}_{\sqsupseteq}$ implies that w has no father in a period starting at $I_w(ack_{m'})$, a contradiction with the fact the edge (v, w) is labelled by t' in T. □

Combining Lemmas 5.7 and 5.8, we obtain that the directed graph T is acyclic.

Lemma 5.9. *If $V_T \neq \emptyset$, then T is a directed tree rooted at the node e.*

Proof. Assume that $V_T \neq \emptyset$, and let $j \in V_T$ with two incoming neighbors i and i' in T. By definition of E_T, both i and i' are in V. We consider the three following cases.

1. The node j is in V. Hence $i = father_j$ and $i' = father_j$, which shows that $i = i'$.

2. The node j is a base message msg. Thus msg has been sent by i and i'. Since any message has a unique sender, we obtain that $i = i'$.

3. The node j is an acknowledgement. Since any message (in particular, any acknowledgement) has a unique receiver, we also have $i = i'$ in this case.

This shows every node of T has at most one incoming neighbor.

Moreover, Lemma 5.5 shows that every node j in $V_T \cap V$ has an incoming neighbor, and Lemma 5.6 yields the same conclusion for the other nodes in V_T, namely the messages and the acknowledgements that are in transit. Hence every node of T has exactly one incoming neighbor. Since T is acyclic, it follows that T is a directed tree. □

Theorem 5.1. *The Dijkstra-Scholten algorithm correctly detects termination of any diffusing computation.*

Proof. Suppose that the agent e detects termination in some configuration C. Then $father_e = \perp$ holds in C, and Lemma 5.9 shows that T is empty, i.e., every agent has no father. Combining Lemmas 5.3 and 5.4, we obtain that all agents are passive and there is no base message in transit, which completes the safety proof.

Liveness is a straightforward consequence of the pseudo-code. □

Chapter 6

The Consensus Problem and the FLP Result

Roughly speaking, the consensus problem consists in reaching a configuration where agents decide on a common value. It is a fundamental problem in distributed computing and has many real-world applications, especially in any application that involve replication.

Formally, let $\mathcal{V}$ be a non-empty set of values such that $\perp \notin \mathcal{V}$. We assume that each agent u has an initial value $\mu_u \in \mathcal{V}$ and an output variable $y_u \in \mathcal{V} \cup \{\perp\}$, which is initially set to $\perp$. When the agent u assigns some value $v \in \mathcal{V}$ to its variable y_u, u is said to *decide* v. An algorithm solves the distributed consensus problem if in every execution of this algorithm each agent decides exactly once, all the decision values are equal, and the common decision value is one initial value.

In a system without any failure, this problem is easy to solve: if $\mathcal{V}$ is totally ordered, then any algorithm that computes the maximum initial value (cf. Chapter 2) achieves consensus.

As explained above, the consensus problem is the paradigm of fault-tolerance, and hence one fundamental issue is to solve it in the context of *failures*. Various types of failures of various severity levels may happen: for example, messages may be lost or duplicated; or some agents may crash (i.e., stop to take steps forever), may omit to execute some transitions, or they may misbehave arbitrarily. In this chapter, we will assume that channels are reliable, and at most f agents are prone to crash failures.

6.1 Computing model

We consider a networked system of n agents, where the communication graph is complete. The set of agents may be denoted by $V = \{1, \ldots, n\}$. The class of algorithms under consideration is restricted to the algorithms with only two types of guarded commands:

$R_{u,m}$ :: {a message $\langle m \rangle$ arrives}
  receive $\langle m \rangle$
  $state_u \leftarrow f_u(m, state_u)$
  send $\langle m_{i_1} \rangle$ to v_{i_1}, …, send $\langle m_{i_k} \rangle$ to v_{i_k} % $\{v_{i_1}, \ldots, v_{i_k}\}$ is any (possibly empty) set of agents

$R_{u,\emptyset}$ :: {**true**}
  $state_u \leftarrow f_u(\emptyset, state_u)$

send $\langle m_{i_1} \rangle$ to v_{i_1}, ..., send $\langle m_{i_k} \rangle$ to v_{i_k} % $\{v_{i_1}, \ldots, v_{i_k}\}$ is any (possibly empty) set of agents

where f_u is a transition function, and $\{v_{i_1}, \cdots, v_{i_k}\}$ as well as the messages $\{m_{i_1}, \cdots, m_{i_k}\}$ are determined by u's current state and some sending function g_u. Local algorithms are thus deterministic and differ on their transition and sending functions.

Definition 6.1. A *step* is any pair (u, m) where u is an agent and $m \in \mathcal{M} \cup \{\emptyset\}$. A *schedule* σ is a (possibly infinite) non-empty sequence of steps.

Let $\mathcal{A}$ be an algorithm of the above form. There is a unique guarded command $R_{u,m}$ associated to the step (u, m). This step is applicable to the configuration C of the algorithm $\mathcal{A}$ if the message $\langle m \rangle$ is in transit towards u in C, while $(u, \emptyset)$ is applicable to any configuration of $\mathcal{A}$. If $e = (u, m)$ is applicable to C, denoted $e \models C$, then there is a unique resulting configuration (each local algorithm is deterministic) which is denoted $e \cdot C$.

Definition 6.2. A *schedule* σ is a (possibly infinite) non-empty sequence of steps $e_1, \ldots, e_l, \ldots$. It is applicable to a configuration C of an algorithm $\mathcal{A}$, denoted $\sigma \overset{\mathcal{A}}{\models} C$, if

- e_1 is applicable to $C_0 = C$
- For all $k \geqslant 0$, e_{k+1} is applicable to C_k and $C_{k+1} = e_{k+1} \cdot C_k$.

For convenience, we will write $\sigma = \ldots e_l \ldots e_1$.

There is a clear one-to-one correspondance between the executions of the algorithm $\mathcal{A}$ and the pairs (C_0, σ) where C_0 is an initial configuration of $\mathcal{A}$, and σ is a schedule applicable to C_0. The sole steps applicable to an initial configuration are of the form $(u, \emptyset)$. Observe that the notions of step and schedule relate only to agents and messages, and can be defined independently to some algorithm. However the corresponding guarded commands, the applicability of schedules and the resulting executions are semantical notions in the sense that they depend on the algorithm under consideration.

Lemma 6.1. *Let $\mathcal{A}$ be an algorithm, and let C be one of its configurations. If the step $e = (u, m)$ and the finite schedule σ are both applicable to C, and if e does not occur in σ, then e is applicable to $\sigma \cdot C$.*

Proof. If $m = \emptyset$, then $R_{u,m}$ is applicable to any configuration. Otherwise, $m \in \mathcal{M}$ is in transit in configuration C. Since u does not execute any action in σ, m is still in transit in $\sigma \cdot C$, so $R_{u,m}$ is applicable to $\sigma \cdot C$. The same argument applies to σ and $R_{u,m} \cdot C$. Moreover, for all agent $v \neq u$, the state of v in configuration $R_{u,m} \cdot C$ is the same as in C, and the state of u in $\sigma \cdot C$ is the same as in C. Hence, $R_{u,m} \cdot \sigma \cdot C = \sigma \cdot R_{u,m} \cdot C$

□

Since $(u, \emptyset)$ is applicable to any configuration, a complete execution of any algorithm of the above form is infinite. An agent u that has not crashed in an infinite schedule σ takes an infinite number of steps in σ, in which case u is said to be *correct*; otherwise, it is *faulty*. This terminology of correct and faulty agent is naturally extended to complete executions (of an algorithm).

Definition 6.3. A complete execution α of an algorithm $\mathcal{A}$ is *admissible* if

1. every message sent in α to a correct agent is eventually received in α;

2. there is at most one faulty agent in α.

Admissible executions thus model executions with reliable channels and at most one crash failure.

We now prove a key lemma which is a "macro-version" of the Diamond Lemma (Lemma 1.3, Chapter 1).

Lemma 6.2. *Let $\mathcal{A}$ be an algorithm, and let C be one of its configurations. If σ_1 and σ_2 are two schedules both applicable to C such that the sets of agents taking steps in σ_1 and σ_2 are disjoint, then, σ_2 is applicable to $\sigma_1 \cdot C$, σ_1 is applicable to $\sigma_2 \cdot C$ and $\sigma_2 \cdot \sigma_1 \cdot C = \sigma_1 \cdot \sigma_2 \cdot C$*

Proof. The proof is straightforward by a repeated application of Lemma 6.1. □

For convenience, we define the following equivalence relation on complete executions. Let $\alpha = (C_0, \sigma)$ and $\alpha' = (C'_0, \sigma')$ be two infinite executions and $u \in V$. The relation of *indistinguishability with respect to* u, denoted $\sim_u$, is defined as

$$\alpha \sim_u \alpha' \iff \forall t \in \mathbb{N}^* \quad state_u(\sigma(1:t) \cdot C_0) = state_u(\sigma'(1:t) \cdot C'_0)$$

For every configuration C of an algorithm, we can naturally construct the directed graph $\mathcal{G}_C = (\mathcal{C}_C, \mathcal{E}_C)$ where $\mathcal{C}_C$ is the set of configurations reachable from C, and $(C_1, C_2) \in \mathcal{E}_C$ if there exists a step e such that $C_2 = e \cdot C_1$. Clearly, $\mathcal{G}_C$ is a directed tree, rooted at C.

6.2 The impossibility result

Theorem 6.1 (FLP)**.** *There is no consensus algorithm that tolerates one crash failure, even with reliable channels and a complete communication graph.*

We prove this result for the problem of *binary consensus* where the set of values is $\mathcal{V} = \{0, 1\}$. The proof is by contradiction: we suppose that there exists a binary consensus algorithm $\mathcal{A}$ that tolerates one crash failure. This means that every admissible (complete) execution of $\mathcal{A}$ satisfies:

Termination: every correct agent eventually decides.

Agreement: if u and v decide ν and ν', then $\nu = \nu'$.

Integrity: if u decides the value ν, then ν is one of the initial value.

Any initial configuration C_0 of $\mathcal{A}$ corresponds to a mapping from V to $\{0, 1\}$, and thus may be denoted by $C_0 = (\mu_1, \cdots, \mu_n)$ if for every agent k, μ_k is its initial value.

A configuration C is *ν-decided*, $\nu \in \{0, 1\}$, if there exists $v \in V$ such that $y_v = \nu$ in C. The agreement property implies that if C is ν- and ν'-decided, then $\nu = \nu'$. If C is neither 0-decided nor 1-decided, then C is said to be *non-decided*. Observe that if C is ν-decided, then every configuration in $\mathcal{G}_C$ is ν-decided.

We define the *valency* of a configuration C as the set of values ν such that there exists a ν-decided configuration in $\mathcal{C}_C$. The configuration C is *ν-valent* if there exists one ν-decided configuration in $\mathcal{C}_C$. A configuration C is *bivalent* if it is both 0-valent and 1-valent; otherwise it is *monovalent*. Obviously, if C is ν-monovalent, then any configuration in $\mathcal{C}_C$ is also ν-monovalent.

Lemma 6.3. *There exists an initial bivalent configuration.*

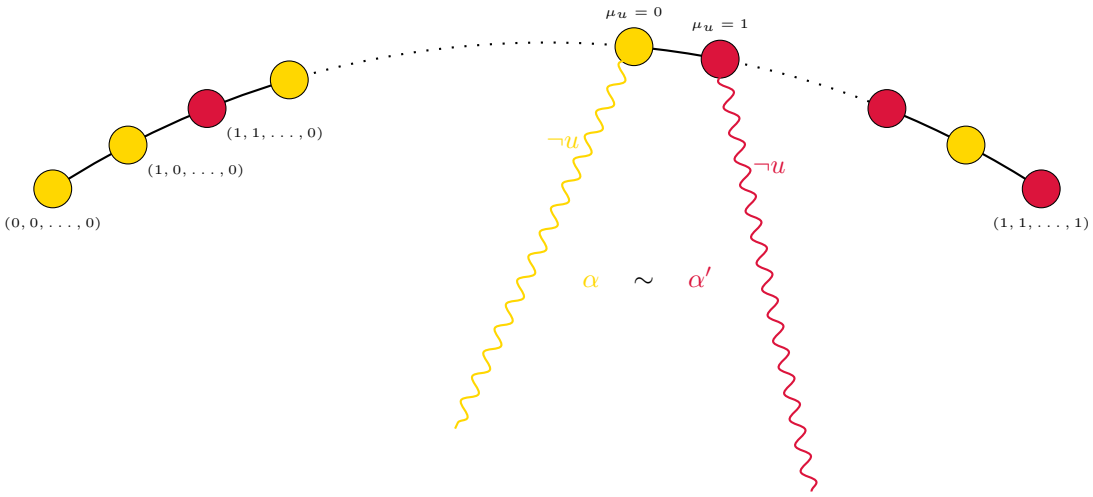


Figure 6.1: Initial bivalence configuration

Proof. The proof is by contradiction: suppose that there is no bivalent initial configuration. Consider a chain $\{C_i\}_{0 \leqslant i \leqslant n}$ of initial configurations where

$$C_i = (1, \ldots, 1, 0, \ldots, 0)$$

denotes the initial configuration where all the entries are equal to 0 except the first i entries that are equal to 1. The validity property implies that C_0 is 0-monovalent. Similarly, C_n is 1-monovalent. By assumption, there is no bivalent initial configuration, and so each configuration C_i is either 0-monovalent or 1-monovalent. It follows that, along the chain of initial configurations $C_0, C_1, \ldots, C_n$, there exist two neighboring configurations C_j and C_{j+1} such that C_j is 0-monovalent and C_{j+1} is 1-monovalent. This is depicted in Figure 6.1. Let u be the single agent whose initial values in C_j and C_{j+1} are 0 and 1, respectively.

Consider an infinite schedule σ applicable to C_j such that u takes no step in σ. As C_{j+1} differs only from C_j by the initial value of u, σ is also applicable to C_{j+1}. This yields two admissible executions α and α' of $\mathcal{A}$ that are indistinguishable from the viewpoint of any agent v different from u:

$$\forall v \in V \setminus \{u\}, \quad \alpha \sim_v \alpha'.$$

From some point in α and α', all the configurations are 0-decided and 1-decided, respectively, a contradiction. □

Let C be any reachable configuration of $\mathcal{A}$, and let e be a step applicable to C, i.e., $e \models C$. We introduce the following two sets of (reachable) configurations of $\mathcal{A}$:

1. $\mathcal{C} = \{\sigma \cdot C \mid \sigma \models C \wedge e \notin \sigma\}$
2. $\mathcal{D} = e \cdot \mathcal{C} = \{e \cdot C' \mid C' \in \mathcal{C} \wedge e \models C'\}$.

Lemma 6.4. *If C is bivalent, then $\mathcal{D}$ contains at least one bivalent configuration.*

Proof. For the sake of a contradiction, suppose that $\mathcal{D}$ does not contain any bivalent configuration. Under this assumption, we first show two claims.

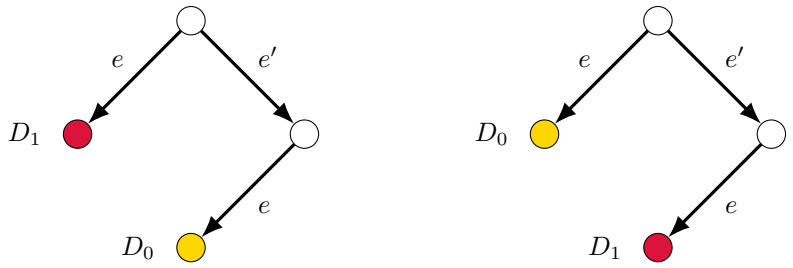
Claim 1. The set $\mathcal{D}$ contains two configurations D_0 and D_1 which are 0-monovalent and 1-monovalent, respectively.

Proof. Since C is bivalent, there exist two schedules leading to a 0-monovalent configuration and 1-monovalent configuration from C. Let σ_0 and σ_1 be such schedules of minimal lengths, and let $C_i = \sigma_i \cdot C$. For each $i \in \{0, 1\}$, we consider the two following cases:

1. If $C_i \in \mathcal{C}$, then Lemma 6.1 shows that e is applicable to C_i. We set $D_i = e \cdot C_i$.
2. Otherwise, $C_i = \sigma_i^2.e.\sigma_i^1 \cdot C$, with $e \notin \sigma_i^1$. The configuration $e \cdot \sigma_i^1 \cdot C$ is in $\mathcal{D}$, and so is monovalent. Since the schedule σ_i is of minimal length, σ_i^2 is empty, i.e., $C_i = e \cdot \sigma_i^1 \cdot C$. Then we set $D_i = C_i$.

In both cases, the configuration D_i is i-monovalent and is in $\mathcal{D}$. □

Claim 2. The directed tree $\mathcal{C}_C$ contains one of the two following patterns



where D_0 and D_1 are 0-monovalent and 1-monovalent, respectively.

Proof. We consider the subtree of configurations in $\mathcal{C}$, and the underlying undirected graph, which is connected. In this graph, every configuration C' may be colored in red or yellow according to the following rule:

1. if $e \cdot C'$ is 0-monovalent, then C' is yellow;
2. otherwise, C' is red.

The above claim shows that there is at least one yellow configuration and one red configuration. There exists two neighboring configurations in the graph with different colors since it is connected, which proves the claim. □

We complete the proof of Lemma 6.3 in the case the first pattern occurs (the proof for the second pattern is similar), i.e.,

$$D_0 = e \cdot C_0 \quad \text{and} \quad D_1 = e \cdot C_1 = e.e' \cdot C_0.$$

Let u and v the two agents in e and e', respectively. There are two cases to consider:

1. $u \neq v$. A direct application of the Diamond lemma leads to a contradiction.
2. $u = v$. Let σ an infinite schedule applicable to C_0 in which the agent u takes no step. The termination property ensures that from some point in σ, the configurations in the corresponding execution are all monovalent. Let Γ be the first one, and let ν its valency. The Diamond Lemma applied to the configuration C_0 and the two disjoint schedules e and σ yields $\nu = 0$. Then this lemma applied to C_0 and the two disjoint schedules σ and $e.e'$ gives $\nu = 1$, a contradiction.

Consequently, the set of configurations $\mathcal{D}$ contains a bivalent configuration. □

We now construct an admissible execution of the algorithm $\mathcal{A}$ in which all the configurations are bivalent, which is in contradiction with the Termination property, and thus prove the FLP theorem. For that, we proceed with a *round robin argument*: at each level k, we start with a bivalent configuration C^{k-1} and a specific step e_k applicable to C^{k-1}. Then we construct a finite schedule σ_k that is applicable to C^{k-1} and ends with e_k, and such that $C^k = \sigma^k \cdot C^{k-1}$ is bivalent.

1. The first configuration C^0 is an initial bivalent configuration of $\mathcal{A}$ given by Lemma 6.3.

2. Suppose that the above construction up to level $k-1$ is achieved. Let $e_k = (u, m)$ be the step where $u = k \bmod n$ and m is the first message sent to u and in transit in C^{k-1} if exists, and $m = \emptyset$ otherwise. Lemma 6.4 ensures that there is a finite schedule σ_k that is applicable to C^{k-1}, ends with e_k, and leads to a bivalent configuration C^k.

With the above construction, the limit schedule (for the Cantor topology) $\sigma = \lim_{k\to\infty} \sigma_k \cdots \sigma_1$ is an admissible schedule (no agent failure and no message loss). Therefore, the corresponding execution (C^0, σ) is admissible and violates the Termination property.