11 Coordination Algorithms

11.1 INTRODUCTION

Distributed applications rely on specific forms of coordination among processes to accomplish their goals. Some tasks of coordination can be viewed as a form of preprocessing. Examples include clock synchronization, spanning tree construction, and consensus. In this chapter, we single out two specific coordination algorithms and explain their construction. Our first example addresses leader election, where one among a designated set of processes is chosen as leader and assigned special responsibilities. The second example addresses a problem of model transformation. Recall that asynchrony is hard to deal with in real-life applications due to the lack of temporal guarantees — it is simpler to write algorithms on the synchronous process model (where processes execute actions in lock-step synchrony) and easier to prove their correctness. This motivates the design of synchronizers that transform an asynchronous model into a synchronous one. In this chapter, we discuss several algorithms for leader election and synchronizer construction.

11.2 LEADER ELECTION

Many distributed applications rely on the existence of a leader process. The leader is invariably the focus of control, entrusted with the responsibility of systemwide management. Consider the client–server model of resource management. Here, the server can be viewed as a leader. Client processes send requests for resources to the server, and based on the information maintained by the server, a request may be granted or deferred or denied. As another example, consider a centralized database manager: The shared data can be accessed or updated by client processes in the system. This manager maintains a queue of pending reads and writes, and processes these requests in an appropriate order. Such centralization of control is not essential, but it offers a simple solution with manageable complexity.

When the leader (i.e., coordinator) fails or becomes unreachable, a new leader is elected from among nonfaulty processes. Failures affect the topology of the system — even the partitioning of the system into a number of disjoint subsystems is not ruled out. For a partitioned system, some applications elect a leader from each connected component. When the partitions merge, a single leader remains and others drop out.

It is tempting to compare leader election with the problem of mutual exclusion and to use a mutual exclusion algorithm to elect a leader. They are not exactly equivalent. The similarity is that whichever process enters the critical section becomes the leader; however, there are three major differences between the two paradigms:

1. Failure is not an inherent part of mutual exclusion algorithms. In fact, failure within the critical section is typically ruled out.
2. Starvation is an irrelevant issue in leader election. Processes need not take turns to be the leader. The system can happily function for an indefinite period with its original leader, as long as there is no failure.

3. If leader election is viewed from the perspective of mutual exclusion, then exit from the critical section is unnecessary. On the other hand, the leader needs to inform every active process about its identity, which is not a issue in mutual exclusion.

A formal specification of leader election follows: Assume that each process has a unique identifier from a totally ordered set **V**. Also, let each process **i** have a variable **L** that represents the identifier of its leader. Then the following condition must eventually hold:

$$\forall\ \mathbf{i}, \mathbf{j} \in \mathbf{V} : \mathbf{i}, \mathbf{j}\ \textbf{are nonfaulty} :: \mathbf{L(i)} \in \mathbf{V} \wedge \mathbf{L(i)} = \mathbf{L(j)} \wedge \mathbf{L(i)}\ \textbf{is nonfaulty}$$

11.2.1 The Bully Algorithm

The bully algorithm is due to Garcia-Molina [G82] and works on a completely connected network of processes. It assumes that (i) communication links are fault-free, (ii) processes can fail only by stopping, and (iii) failures can be detected using timeout. Once a failure of the current leader is detected, the bully algorithm allows the nonfaulty process with largest id eventually to elect itself as the leader.

The algorithm uses three different types of messages: *election*, *reply*, and *leader*. A process initiates the election by sending an *election* message to every other process with a higher id. By sending this message, a process effectively asks, "Can I be the new leader?" A *reply* message is a response to the election message. To a receiving process, a reply implies, "No, you cannot be the leader." Finally, a process sends a *leader* message when it believes that it is the leader. The algorithm can be outlined as follows:

Step 1. Any process, after detecting a failure of the leader, bids to become the new leader by sending an election message to every process with a higher identifier.

Step 2. If any process with a higher id responds with a reply message, then the requesting process gives up its bid to become the leader. Subsequently, it waits to receive a leader message ("I am the leader") from some process with a higher identifier.

Step 3. If no higher-numbered process responds to the election message sent by node **i**, then node **i** elects itself the leader and sends a leader message to every process in the system.

Step 4. If no leader message is received by process **i** within a timeout period after receiving a reply to its election message, then process **i** suspects that the winner of the election failed in the meantime and **i** reinitiates the election.

```
program     bully; {program for process i}
define      failed: boolean {true if the current leader fails}
            L: process {identifies the leader}
            m: message {election |leader| reply}
            state: idle |wait for reply| wait for leader
initially   state = idle (for every process)
do          failed                          →  ∀j:j>i :: send election to j;
                                                state := wait for reply;
                                                failed := false
□ (state = idle) ∧ (m = election)           →  send reply to sender;
                                                failed := true
□ (state = wait for reply)
    ∧ (m = reply)                           →  state := wait for leader
□ (state = wait for reply)
    ∧ timeout                               →  L(i) := i;
                                               ∀j:j > i :: send leader to j;
                                               state := idle
```

```
□ (state = wait for leader)
    ∧ (m = leader)                  →  L(i) := sender; state := idle
□ (state = wait for leader)
    ∧ timeout                       →  failed := true; state := idle
od
```

Theorem 11.1 Every nonfaulty process eventually elects a unique leader.

Proof. A process **i** sending out an election message may or may not receive a reply.

Case 1. If **i** receives a reply, then **i** does not send the leader message. In that case, $\forall$**j** : **j is nonfaulty :: L(j)** $\neq$ **i**. However, at least one process, which is a nonfaulty process with the highest id, will never receive a reply.
Case 2. If **i** does not receive a reply, then **i** must be unique in as much as there is no other nonfaulty process **j** such that **j > i**. In this case **i** elects itself the leader, and after the leader message is sent out by **i**, $\forall$**j** : **j is nonfaulty :: L(j) = i holds**.

If Case 1 holds, but no leader message is subsequently received before timeout, then the would-be leader itself must have failed in the meantime. This sets **failed** to true for every process that was waiting for the leader message. As a result, a new election is initiated and one of the above two cases must eventually hold, unless every process has failed. ■

Message complexity. Each process **i** sends an election message to **n − i** other processes. Also, each process **j** receiving an election message can potentially send the reply message to **j − 1** processes. If the would-be leader does not fail in the meantime, then it sends out **n − 1** leader messages; otherwise a timeout occurs, and every process repeats the first two steps. Since out of the **n** processes at most **n − 1** can fail, the first two steps can be repeated at most **n − 1** times. Therefore, the worst-case message complexity of the bully algorithm can be $\mathbf{O}(\mathbf{n}^3)$.

11.2.2 Maxima Finding on a Ring

Once we disregard fault detection, the task of leader election reduces to finding a node with a unique (maximum or minimum) id. (We deal with failures in the next few chapters.) While the bully algorithm works on a completely connected graph, there are several algorithms for maxima finding that work on a sparsely connected graph such as a ring. These solutions are conceptually simple but differ from one another in message complexity. We discuss three algorithms in this section.

11.2.2.1 Chang–Roberts Algorithm

Chang and Roberts [CR79] presented a leader election algorithm for a unidirectional ring. This is an improvement over the first such algorithm, proposed by LeLann [Le77].

Assume that a process can have one of two colors: *red* or *black*. Initially every process is red, which implies that every process is a potential candidate for becoming the leader. A red process initiates the election by sending a token that means "I want to be the leader." Any number of red processes can initiate the election. If, however, a process receives a token before initiating the algorithm, then it knows that there are other processes running for leadership — so it decides to quit, and turns black. A black process never turns red.

```
program   Chang-Roberts (for an initiator i)
define    token : process id
initially all processes are red, and i sends a token <i>
           to its neighbor;
```

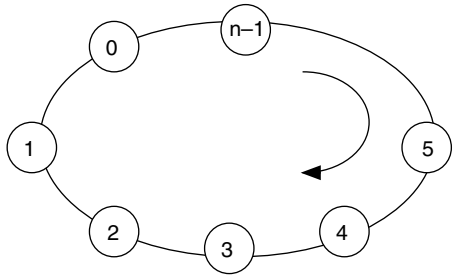

FIGURE 11.1 An execution of the Chang–Roberts election algorithm.

```
do  token <j> ∧ j < i → skip {j's token is removed and j
     will not be elected}
 □  token <j> ∧ j > i → send <j>; color := black {i resigns}
 □  token <j> ∧ j = i → L(i) := i {i becomes the leader}
od
{for a noninitiator process}
do  token <j> received → color := black; send <j>     od
```

Let us examine why the program works. A token initiated by a red process **j** will be removed when it is received by a process **i > j**. So ultimately the token from the process with the largest id will prevail and will return to the initiator. Thus, the process with the largest id elects itself the leader. It will require another round of leader messages to notify the identity of the leader to every other process.

To analyze the complexity of the algorithm, consider Figure 11.1. Assume that all processes are initiators, and their tokens are sent in the clockwise direction around the ring. Before the token from process $(\mathbf{n-1})$ reaches the next process $(\mathbf{n-2})$, it is possible for the tokens from every process 0, 1, 2, … , n − 2 (in that order) to reach node $(\mathbf{n-1})$ and be removed. The token from node **k** thus makes a maximum number of $(\mathbf{k+1})$ hops. The worst-case message complexity is therefore $\mathbf{1+2+3+\cdots+n=n(n+1)/2}$.

11.2.2.2 Franklin's Algorithm

Franklin's election algorithm works on a ring that allows bidirectional communication. Compared with Chang–Robert's algorithm, it has a lower message complexity. Processes with unique identifiers are arranged in an arbitrary order in the ring. There are two possible colors for each process: red or black. Initially each process is red.

The algorithm is synchronous and works in rounds. In each round, to bid for leadership, each red process sends a token containing its unique id to both neighbors and then examines the tokens received from other processes. If process **i** receives a token from a process **j** and **j** > **i**, **i** quits the race and turns black. A black process remains passive and does not assume any role other than forwarding the tokens to the appropriate neighbors.

Since tokens are sent in both directions, whenever two adjacent red processes exchange tokens, one of them must turn black. In each round, a fraction of the existing red processes turn black. The algorithm terminates when there is only one red process in the entire system. This is the leader. The program for a red process **i** is as follows:

```
{program for a red process i in round r,  r ≥ 0}
send token <i> to both neighbors;
receive tokens from both neighbors;
if ∃ token <j> : j > i → color := black
```

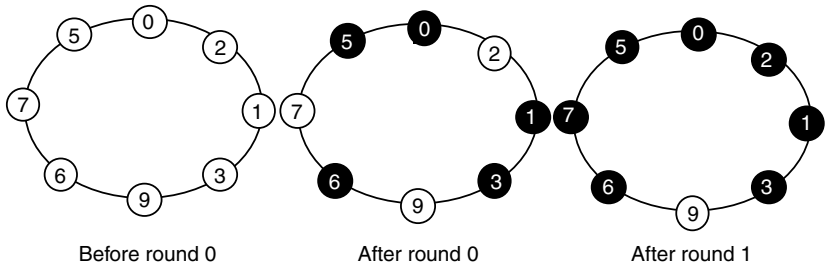

FIGURE 11.2 An execution of Franklin's algorithm. The shaded processes are black. After two rounds, process 9 is identified as the maximum.

```
□ ∀ token <j> : j < i → r := r + 1;
     execute program for the next round
□ ∀ token <j> : j = i → L(i) := i
fi
```

Theorem 11.2 Franklin's algorithm elects a unique leader.

Proof. For a red process **i**, in each of the two directions, define a red neighbor to be the red process that is closest to **i** in that direction. Thus, in Figure 11.2, after round 0, processes 7 and 9 are the two red neighbors of process 2.

After each round, every process **i** that has at least one red neighbor **j** such that $\mathbf{j > i}$ turns black. Therefore, in a ring with $\mathbf{k}(\mathbf{k > 1})$ red processes, at least $\mathbf{k/2}$ turn black. Initially $\mathbf{k = n}$. Therefore, after at most **log n** rounds, the number of red processes is reduced to one. In the next round, it becomes the leader. ■

The algorithm terminates in at most **log n** rounds, and in each round, every process sends (or forwards) a message in both directions. Therefore, the worst-case message complexity of Franklin's algorithm is **O(n log n)**.

11.2.2.3 Peterson's Algorithm

Similar to Franklin's algorithm, Peterson's algorithm works in synchronous rounds. Interestingly, it elects a leader using **O(n log n)** messages even though it runs on a unidirectional ring. Compared with Franklin's algorithm, there are two distinct differences:

1. A process communicates using an alias that changes during the progress of the computation.
2. A unique leader is eventually elected, but it is not necessarily the process with the largest identifier in the system.

As before, we assume that processes have one of two colors: *red* or *black*. Initially every process is red. A red process turns black when it quits the race to become a leader. A black process is passive — it acts only as a router and forwards incoming messages to its neighbor.

Assume that the ring is oriented in the clockwise direction. Any process will designate its anticlockwise neighbor as its predecessor, and its clockwise neighbor as its successor. Designate the red predecessor (the closest red process in the anticlockwise direction) of **i** by **N(i)**, and the red predecessor of **N(i)** by **NN(i)**. Until a leader is elected, in each round, every red process **i** receives two messages: one from **N(i)** and the other from **NN(i)**. These messages contain aliases of the senders.

The channels are FIFO. Depending on the relative values of the aliases, a red process either decides to continue with a new alias or quits the race by turning black.

Denote the alias of a process **i** by **alias(i)**. Initially, **alias(i)** = **i**. The idea is comparable to that in Franklin's algorithm, but unlike Franklin's algorithm, a process cannot receive a message from both neighbors. So, each process determines the local maximum by comparing **alias(N)** with its own alias and with **alias(NN)**. If **alias(N)** happens to be larger than the other two, then the process continues its run for leadership by assuming **alias(N)** as its new alias. Otherwise, it turns black and quits.

In each round, every red process executes the following program:

```
    program   Peterson
    define    alias : process id
             color : black or red
    initially ∀ i : color(i) = red, alias(i) = i
    {program for a red process in each round}
1   send alias;
2   receive alias (N);
3   if alias = alias (N) → I am the leader
4      alias ≠ alias (N) → send alias(N);
5              receive alias(NN);
6              if alias(N) > max (alias, alias (NN)) →
7                           alias := alias (N)
8                           alias(N) < max (alias, alias (NN)) →
9                                       color := black
                            fi
     fi
```

Figure 11.3 illustrates the execution of one round of Peterson's algorithm.

Proof of Correctness. Let **i** be a red process before round **r**(**r ≥ 0**) begins. Then, after round **r**, process **i** remains red if and only if the following condition holds:

[LocalMax] **(alias (N(i)) > alias (i)) ∧ (alias(N(i)) > alias (NN(i)))**

We show that in each round, at least one of the two red processes **i** and **N(i)** will turn black.

Theorem 11.3 Let **i** be a red process, and **j = N(i)** when a round begins. Then at the end of the round, either **i** or **j** must turn black.

Proof (by contradiction). Assume that the statement is false. Then both **i** and **j** will remain red after that round. From **LocalMax**, it follows that if **i** remains red, then **(alias**

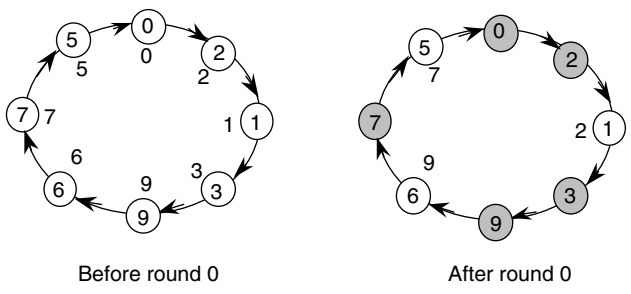

FIGURE 11.3 One round of execution of Peterson's algorithm. For each process, its id appears inside the circle and its alias appears outside the circle. Shaded circles represent black processes.

(j) > alias (i)) ∧ (alias(j) > alias(N(j))) must hold. Again if **j** remains red after that round, then **(alias(N(j))> alias (j)) ∧ (alias(N(j)) > alias(NN(j)))** must also hold. Since both of these cannot be true at the same time, the statement holds. ■

It is not impossible for two neighboring red processes to turn black in the same round (see Figure 11.3). In fact this helps our case, and accelerates the convergence.

It follows from Lemma 11.3 that after every round, at least half of the existing red processes turn black. Finally, only one red process **i** remains, and the condition **alias(i) = alias(N(i))** holds in the final round — so process **i** elects itself the leader.

Message Complexity. Since there are at most **log n** rounds, and in each round every process sends out two messages, the number of messages required to elect a leader is bounded from above by **2n · log n**. Despite the fact that the communication is unidirectional, the message complexity is not inferior to that found in Franklin's algorithm for a bidirectional ring.

11.2.3 Election in Arbitrary Networks

For general networks, if a ring is embedded on the given topology, then a ring algorithm can be used for leader election. (An embedded ring is a cyclic path that includes each vertex at least once.) The orientation of the embedded ring helps messages propagate in a predefined manner.

As an alternative, one can use flooding to construct a leader election algorithm that will run in rounds. Initially, **∀i : L(i) = i**. In each round, every node sends the id of its leader to all its neighbors. A process **i** picks the largest id from the set {**L(i) ∪ the set of all ids received**}, assigns it to **L(i)**, and sends **L(i)** out to its neighbors. The algorithm terminates after **D** rounds, where **D** is the diameter of the graph. Here is an outline:

```
program general network;
define r : integer {round number},
       L : process id {identifies the leader}
initially r = 0, L(i) = i
{program for process i}
do r < D →
      send L(i) to each neighbor;
      receive all messages from the neighbors;
      L(i) := max  {L(i) ∪ set of leader ids received
        from neighbors}
      r := r + 1
od
```

The algorithm requires processes to know the diameter (or at least the total number of processes) of the network. The message complexity is **O(δ · D)**, where **δ** is the maximum degree of a node.

11.2.4 Election in Anonymous Networks

In anonymous networks, processes do not have identifiers. Therefore the task of leader election becomes an exercise in symmetry breaking. Randomization is a widely used tool for breaking symmetry. Here we present a randomized algorithm that uses only one bit **b** per process to elect a leader. This solution works on completely connected networks only.

The state of a process can be active or passive. Initially, every process is active, and each active process is a contender for leadership. In every round, each active process **i** executes the following program, until a leader is elected. The operation **random{S}** randomly picks an element from the set **S**.

```
program randomized leader election;
initially b(i) = random{0,1};
send b to every active neighbor;
receive b from every active neighbor;
if b(i)=1 ∧ ∀j ≠ i : b(j)=0 → i is the leader
   (b(i)=1 ∧ ∃j ≠ i : b(j)=1) ∨ (∀k : b(k) = 0) →
                                 b(i) = random{0,1}
                                 go to next round
   b(i) = 0 ∧ ∃j : b(j) = 1   → become passive
fi
```

The description uses ids for the purpose of identification only. As long as **random{0,1}** generates the same bit for every active process, no progress is achieved. However, since the bits are randomly chosen, this cannot continue for ever, and some bit(s) inevitably become different from others. A process **j** with $\mathbf{b(j) = 0}$ quits if there is another process **i** with $\mathbf{b(i) = 1}$.

Note that after each round, half of the processes are expected to find $\mathbf{b = 0}$, and quit the race for leadership. So in a system of **n** processes a leader is elected after an expected number of **log n** rounds using **O(n log n)** messages.

11.3 SYNCHRONIZERS

Distributed algorithms are easier to design on a synchronous network in which processes operate in lock-step synchrony. The computation progresses in discrete steps known as rounds or ticks. In each tick, a process can

- Receive messages from its neighbors
- Perform a local computation
- Send out messages to its neighbors

Messages sent out in tick i reach their destinations in the same tick, so that the receiving process can read it in the next tick $(i + 1)$. Global state transition occurs after all processes complete their actions in the current tick.

A synchronizer is a protocol that transforms an asynchronous model into a synchronous process model (i.e., processes operate in lock-step synchrony). It does so by simulating the ticks on an asynchronous network. Actions of the synchronous algorithm are scheduled at the appropriate ticks, and the system is ready to run the synchronous version of distributed algorithms. This results in a two-layered design. Synchronizers provide an alternative technique for designing asynchronous distributed algorithms. Regardless of how the ticks are simulated, a synchronizer must satisfy the following condition:

(**Synch**) Every message sent in tick **k** must be received in tick **k**.

Despite apprehension about the complexity of the two-layered algorithm, the complexity figures of algorithms using synchronizers are quite encouraging. In this section, we present the design of a few basic types of synchronizers.

11.3.1 THE ABD SYNCHRONIZER

An ABD (asynchronous bounded delay) synchronizer [CCGZ90] [TKZ94] can be implemented on a network where every process has a physical clock, and the message propagation delays have a known upper bound δ. In real life, all physical clocks tend to drift. However, to keep our discussion simple, we assume that once initialized, the difference between a pair of physical clocks does not change during the life of the computation.

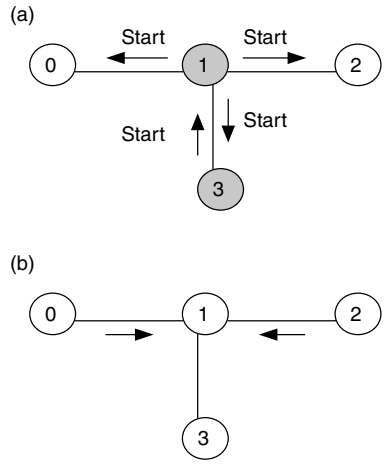

FIGURE 11.4 Two phases of an ABD synchronizer. In (a) 1 and 3 spontaneously initiate the synchronizer operation, initialize themselves, and send start signals to the noninitiators 0 and 2. In (b) 0 and 2 wake up and complete the initialization. This completes the action of tick **0**.

Let **c** denote the physical clock of process. One or more processes spontaneously initiate the synchronizer by assigning $\mathbf{c := 0}$, executing the actions for **tick** 0, and sending a **start** signal to its neighbors (Fig. 11.4). By assumption, actions take zero time. Each non-initiating neighbor **j** wakes up when it receives the **start** signal from a neighbor, initializes its clock **c** to 0, and executes the actions for **tick** 0. This completes the initialization.

Before the actions of **tick** $(i+1)$ are simulated, every process must send and receive all messages corresponding to **tick** i. If **p** sends the **start** message to **q**, **q** wakes up, and sends a message that is a part of initialization actions at **tick** 0, then **p** will receive it *before* time 2δ. Therefore process **p** will start the simulation of the next step (tick 1) at time 2δ. Eventually, process **p** will simulate *tick k* of the synchronous algorithm when its local clock reads $2k\delta$. The permission to start the simulation of a tick thus entirely depends on the clock value, and nothing else.

11.3.2 Awerbuch's Synchronizer

When the physical clocks are not synchronized and the upper bound of the message propagation delays is not known, the ABD synchronizer does not work. Awerbuch [Aw85] addressed the design of synchronizers for such weaker models and proposed three different kinds of synchronizers with varying message and time complexities.

The key idea behind Awerbuch's synchronizers is the determination of when a process is safe (for a given tick). By definition, a process is safe for a given tick when it has received every message sent to it, and also received an acknowledgment for every message sent by it during that tick. Each process has a counter **tick** to preserve the validity of **Synch**, a process increments **tick** (and begins the simulation for the next tick) when it determines that all neighbors are safe for the current tick. Observe that a violation of this policy could cause a process simulating the actions of tick **j** to receive a message for tick $(\mathbf{j+1})$ from a neighbor. Here we describe three synchronizers. Each has a different strategy for using the topological information and detecting safe configurations.

11.3.2.1 The α-Synchronizer

Before incrementing **tick**, each node needs to ensure that it is safe to do so. The $\boldsymbol{\alpha}$-synchronizer implements this by asking each process to send a **safe** message whenever it has received all messages

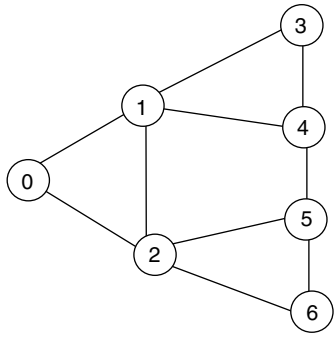

FIGURE 11.5 A network used to illustrate the operation of the α-synchronizer.

for the current tick. Each process executes the following three steps in each tick:

1. Send and receive messages for the current tick.
2. Send **ack** for each incoming message, and receive **ack** for each outgoing message for the current tick.
3. Send **safe** to each neighbor after exchanging all **ack** messages.

A process schedules its actions for the next tick when it receives a **safe** message for the current tick from every neighbor. Here is a partial trace of the execution of the synchronizer for a given tick **j** in the network of Figure 11.5.

Action	Comment
0 sends **m[j]** to 1, 2	*Simulation of tick **j** begins*
1, 2 send **ack[j]** to 0	*0 knows that 1,2 have received **m***
1 sends **m[j]** to 0, 2, 3, 4	
0, 2, 3, 4 send **ack[j]** to 1	*these may be sent in an arbitrary order*
2 sends **m[j]** to 0, 1, 5, 6	
0, 1, 4, 5 send **ack[j]** to 2	
0 sends **safe [j]** to 1, 2	
1 sends **safe [j]** to 0, 2, 3, 4	*0 knows that 1 is safe for the current tick*
2 sends **safe [j]** to 0, 1, 4, 5	*0 knows that 2 is safe for the current tick*

After process **0** receives the **safe** messages from processes **1** and **2**, it increments **tick**.

Complexity Issues. For the α-synchronizer, the message complexity $\mathbf{M}(\alpha)$ is the number of messages passed around the entire network for the simulation of each tick. It is easy to observe that $\mathbf{M}(\alpha) = \mathbf{O}(|\mathbf{E}|)$. Similarly, the time complexity $\mathbf{T}(\alpha)$ is the maximum number of asynchronous rounds required by the synchronizer to simulate each tick across the entire network. Since each process exchanges three messages **m**, **ack**, and **safe** for each tick, $\mathbf{T}(\alpha) = \mathbf{3}$.

If $\mathbf{M_S}$ and $\mathbf{T_S}$ are the message and time complexities of a synchronous algorithm, then the asynchronous version of the same algorithm can be implemented on top of a synchronizer with a message complexity $\mathbf{M}_A$ and a time complexity $\mathbf{T}_A$, where

$$\mathbf{M_A} = \mathbf{M_S} + \mathbf{T_S} \cdot \mathbf{M}(\alpha)$$
$$\mathbf{T_A} = \mathbf{T_S} \cdot \mathbf{T}(\alpha)$$

During a continuous run, after one process initiates the simulation of tick **i**, every other process is guaranteed to initiate the simulation of the same tick within at most **D** time units, where **D** is the diameter of the graph.

11.3.2.2 The β-Synchronizer

The β-synchronizer starts with an initialization phase that involves constructing a spanning tree of the network. The initiator is the root of this spanning tree. This initiator starts the simulation by sending out a message for tick **0** to each child. Thereafter, the operations are similar to those in the α-synchronizer, with the exception that control messages (i.e., **safe** and **ack**) are sent along the tree edges only. A process sends a **safe** message for tick **i** to its parent to indicate that the entire subtree under it is safe. If the root receives a **safe** message from each child, then it is confident that every node in the spanning tree is safe for tick **i** — so it starts the simulation of the next tick **(i + 1)**.

The message complexity $\mathbf{M}(\beta)$ can be estimated as follows. Each process exchanges the following four messages:

1. A message (for the actual computation) to the neighbors for the current tick.
2. An **ack** from each child to its parent for the current tick.
3. A **safe** message from each child via the tree edges, indicating that each child is safe for the current tick. If the process itself is safe, and is not the root, then it forwards the **safe** message to its parent.
4. When the root receives the **safe** message, it knows that the entire tree is safe, and it sends out a **next** message via the tree edges to the nodes of the network. After receiving the **next** message, a node resumes the simulation of the next tick.

In a spanning tree of a graph with **N** nodes, there are $\mathbf{N-1}$ edges. Since the above three control messages **(ack, safe, next)** flow through each of the $\mathbf{N-1}$ edges, the additional message complexity $\mathbf{M}(\beta) = \mathbf{3}(\mathbf{N-1})$. The time complexity $\mathbf{T}(\beta)$ is proportional to the height of the tree, which is at most $\mathbf{N-1}$, and is often much smaller when the tree is balanced.

The method of computing the complexity of an asynchronous algorithm using a β-synchronizer is similar to that using the α-synchronizer, except that there is the overhead for the construction of the spanning tree. This is a one-time overhead and its complexity depends on the algorithm chosen for it.

11.3.2.3 The γ-Synchronizer

Comparing the α-synchronizer with the β-synchronizer, we find that the β-synchronizer has a lower message complexity, but the α-synchronizer has a lower time complexity. The γ-synchronizer combines the best features of both the α- and β-synchronizers.

In a γ-synchronizer, there is an initialization phase, during which the network is divided into clusters of processes. Each cluster contains a few processes. The β-synchronizer protocol is used to synchronize the processes within each cluster, whereas to synchronize processes between clusters, the α-synchronizer is used. Each cluster identifies a leader that acts as the root of a spanning tree for that cluster. Neighboring clusters communicate with one another through designated edges known as intercluster edges (see Figure 11.6).

Using the β-synchronizer algorithm, when the leader of a cluster finds that each process in the cluster is safe, it broadcasts to all processes in the cluster that the cluster is safe. Processes incident on the intercluster edges forward this information to the neighboring clusters.

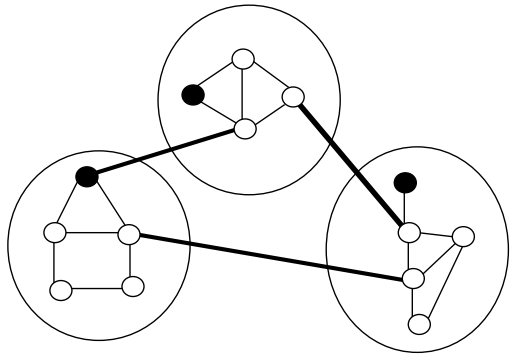

FIGURE 11.6 The clusters in a γ-synchronizer: The shaded nodes are the leaders in the clusters, and the thick lines are the intercluster edges between clusters.

These nodes in the neighboring clusters convey to their leaders that the neighboring clusters are safe. The sending of a **safe** message from the leader of one cluster to the leader of a neighboring cluster simulates the sending of a **safe** message from one node to its neighbor in the α-synchronizer.

We now compute the message and time complexities of a γ-synchronizer. Decompose the graph into $\mathbf{p}$ clusters $\mathbf{0}, \mathbf{1}, \mathbf{2}, \ldots, \mathbf{p-1}$. In addition to all messages that belong to the application, each process sends the following control messages:

1. An **ack** is sent to the sender of a message in the cluster.
2. A **safe** message is sent to the parent after a **safe** message is received from all children in the cluster. When the leader in a cluster receives the **safe** messages from every child, it knows that the entire cluster is safe. To pass on this information, the leader broadcasts a **cluster.safe** message to every node in the cluster via the spanning tree.
3. When a node incident on the intercluster edge receives the **cluster.safe** message from its parent, it forwards that message to the neighboring cluster(s) through the intercluster edges to indicate that this cluster is safe.
4. The **cluster.safe** message is forwarded towards the leader. When a node receives the **cluster.safe** message from every child (the leaves that are not incident on any intercluster edge spontaneously send such messages) it forwards it to its parent in the cluster tree. The leader of a cluster eventually learns about the receipt of the **cluster.safe** messages from every neighboring cluster. At this time, the leader sends the **next** message to the nodes down the cluster tree, and the simulation of the next tick begins.

Table 11.1 shows the chronology of the various control signals. It assumes that the simulation of the previous tick has been completed at time $\mathbf{T}$, and $\mathbf{h}$ is the maximum height of the spanning tree in each cluster.

It follows from Table 11.1 that the time complexity is $\mathbf{4h+3}$, where $\mathbf{h}$ is the maximum height of a spanning tree within the clusters. To compute the message complexity, note that through each tree edge within a cluster, the messages **ack, safe, cluster.safe**, and **next** are sent out exactly once. In addition, through each intercluster edge, the message **cluster.safe** is sent twice, once in each direction. Therefore, if $\mathbf{e}$ denotes the set of tree edges and intercluster edges per cluster, then the message complexity is $\mathbf{O(|e|)}$ per cluster. If there are $\mathbf{p}$ clusters, then the message complexity of the γ-synchronizer is $\mathbf{O(|e| \cdot p)}$. To compute the overall complexity, one should also take into account the initial one-time overhead of identifying these clusters and computing the spanning trees within each cluster.

Note that if there is a single cluster, then the γ-synchronizer reduces to a β-synchronizer. On the other hand, if each node is treated as a cluster, then the γ-synchronizer reduces to an α-synchronizer.

TABLE 11.1
A Timetable of the Control Signals in a γ-Synchronizer

Time	Control action
$T + 1$	Send and receive **ack** for the messages sent at time **T**
$T + h + 1$	Leader receives **safe** indicating that its cluster is safe
$T + 2h + 1$	All processes in a cluster receive a **cluster.safe** message from the leader
$T + 2h + 2$	**Cluster.safe** received from (and sent to) neighboring clusters
$T + 3h + 2$	The leader receives the **cluster.safe** from its children
$T + 4h + 2$	All processes receive the **next** message
$T + 4h + 3$	Simulation of the next tick begins

The overall complexity of the γ-synchronizer depends on the number of clusters. Details can be found in Awerbuch's original paper.

11.3.2.4 Performance of Synchronizers

The use of a synchronizer to run synchronous algorithms on asynchronous systems does not necessarily incur a significant performance penalty. To demonstrate this, consider a synchronous BFS (Breadth First Search) algorithm, and transform it to an asynchronous version using a synchronizer. The synchronous algorithm works as follows:

1. A designated root starts the algorithm by sending a probe to each neighbor in tick **0**.
2. All nodes at distance **d** (**d** > 0) receive the first probe in tick **d** − 1, and forward that probe to all neighbors (other than the sender) in tick **d**.
3. The algorithm terminates when every node has received a probe and the BFS tree consists of all edges through which nodes received their first probes.

The BFS tree is computed in **D** rounds (where **D** is the diameter of the graph), and it requires $\mathbf{O}(|\mathbf{E}|)$ messages.

Now, consider running this algorithm on an asynchronous system using an α-synchronizer. The synchronizer will simulate the clock ticks, and the action of the synchronous algorithm will be scheduled at the appropriate ticks. Since the time complexity of simulating each clock tick is **3**, the time complexity of the overall algorithm is **3D** rounds. If there are **N** nodes, then the message complexity is $\mathbf{3N \cdot D}$, since in each round every process sends out three messages. Furthermore, the message complexity of the composite algorithm will be $\mathbf{O}(|\mathbf{E}|) + \mathbf{O}(|\mathbf{E}|) \cdot \mathbf{D}$, that is, $\mathbf{O}(|\mathbf{E}| \cdot \mathbf{D})$. Compare these figures with the complexities of a solution to the same problem without using the synchronizer. A straightforward algorithm for the asynchronous model starts in the same way as the synchronous version, but the additional complexity is caused by the arbitrary propagation speeds of the probes. It is possible that a node receives a probe from some node with distance **d**, and assigns to itself a distance (**d** + 1), but later receives another probe from a node with a distance less than **d**, thus revoking the earlier decision. Such an algorithm requires $\mathbf{O(N^2)}$ messages and the time complexity is also $\mathbf{O(N^2)}$ steps. So a solution using synchronizers is not bad! A more precise calculation of the complexity will reveal the exact differences.

11.4 CONCLUDING REMARKS

Numerous types of coordination problems are relevant to the design of distributed applications. In this chapter, we singled out two such problems with different flavors. A separate chapter addresses the consensus problem, which the most widely used type of coordination.

Extrema (i.e., maxima or minima) finding is a simple abstraction of the problem of leader election since it disregards failures. The problem becomes more difficult when failures or mobility affect the network topology. For example, if a mobile ad hoc network failure is partitioned into two disjoint components, then separate leaders need to be defined for each connected component to sustain a reduced level of performance. Upon merger, one of the two leaders will relinquish leadership. Algorithms addressing such issues differ in message complexities and depend on topological assumptions.

A synchronizer is an example of simulation. Such simulations preserve the constraints of the original system on a per node (i.e., local) basis but may not provide any global guarantee. An example of a local constraint is that no node in a given tick will accept a message for a different tick from another node. A global constraint for a truly synchronous behavior is that when one node executes the actions of tick $\mathbf{k}$, then every other node will execute the actions of the same tick. As we have seen, the $\boldsymbol{\alpha}$-synchronizer does not satisfy this requirement, since a node at distance $\mathbf{d}$ can simulate an action of tick $\mathbf{k} + \mathbf{d} - \mathbf{1}$.

11.5 BIBLIOGRAPHIC NOTES

LeLann [Le77] presented the first solution to the maxima finding problem: Every initiator generates a token that it sends to its neighbors, and a neighbor forwards a token when its own id is lower than that of the initiator; otherwise it forwards a token with its own id. This leads to a message complexity of $\mathbf{O(n^2)}$. Chang and Roberts's algorithm [CR79] is an improvement over LeLann's algorithm since its worst-case message complexity is $\mathbf{O(n^2)}$, but its average-case complexity is **O(n log n)**. The bully algorithm was proposed by Garcia-Molina [G82]. The first algorithm for leader election on a bidirectional ring with a message complexity of **O(n log n)** was described by Hirschberg and Sinclair [HS80], the constant factor being approximately **8**. Franklin's algorithm [F82] had an identical complexity measure, but the constant was reduced to **2**. Peterson's algorithm [P82] was the first algorithm on a unidirectional ring with a message complexity of **O(n log n)** and a constant factor of **2**. Subsequent modifications lowered this constant factor. The randomized algorithm for leader election is a simplification of the stabilizing version presented in [DIM91].

Chou et al. [CCGZ90], and subsequently Tel et al. [TKZ94], studied the design of synchronization on the asynchronous bounded delay model of networks. The $\boldsymbol{\alpha}$-, $\boldsymbol{\beta}$-, and $\boldsymbol{\gamma}$-synchronizers were proposed and investigated by Awerbuch [Aw85].

11.6 EXERCISES

1. In a network of 100 processes, specify an initial configuration of Franklin's algorithm so that the leader is elected in the second round.
2. Consider Peterson's algorithm for leader election on a unidirectional ring of 16 processes **0** through **15**. Describe an initial configuration of the ring so that a leader is elected in the third round.
3. Show that the Chang–Roberts algorithm has an average message complexity of **O(n log n)**.
4. Election is an exercise in symmetry breaking; initially all processes are equal, but at the end, one process stands out as the leader. Assume that instead of a single leader, we want to elect $\mathbf{k}$ leaders $(\mathbf{k} \geq \mathbf{1})$ on a unidirectional ring, Generalize the Chang–Roberts algorithm to elect $\mathbf{k}$ leaders. (Do not consider the obvious solution in which first a single leader gets elected and later this leader picks $(\mathbf{k} - \mathbf{1})$ other processes as leaders. Is there a solution to the $\mathbf{k}$-leader election problem that needs fewer messages that the single leader algorithm?)
5. In a hypercube of $\mathbf{N}$ nodes, suggest an algorithm for leader election with a message complexity of **O(N log N)**.

6. Design an election algorithm for a tree of anonymous processes. (Of course, the tree is not a rooted tree; otherwise the problem would be trivial.) Think of orienting the edges of the tree so that (i) eventually there is exactly one process (which is the leader) with all incident edges directed towards itself, (ii) every leaf process has outgoing edges only, and (iii) every nonleaf process has one or more incoming and one outgoing edge.
7. Consider a connected graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ where each node $\mathbf{v} \in \mathbf{V}$ represents a store. Each node wants to find out the lowest price of an item. Solve the problem on the mobile agent model. Assume that a designated process is the home of the mobile agent. What is the message complexity of your algorithm? Show your calculations.
8. The problem of leader election has some similarities to the mutual exclusion problem. Chapter 7 describes Maekawa's distributed mutual exclusion algorithm with **O(sqrt N)** message complexity. Can we use similar ideas to design a leader election with sublinear message complexity? Explore this possibility and report your findings.
9. A simple synchronizer for an asynchronous distributed system works as follows:
   i. Each process has a variable **tick** initialized to **0**.
   ii. A process with **tick = j** exchanges messages for that tick with its neighbors.
   iii. When a process has sent and received all messages for the current tick, it increments **tick** (the detection of the safe configuration is missing from this proposal).

   Clearly, a process with **tick = j** can receive messages for both **tick = j** and **tick = j + 1** from its neighbors. To satisfy the requirements of a synchronizer, the process will buffer the messages for **tick = j + 1** for later processing until the process has exchanged all messages for **tick = j** and incremented its tick.

   Comment on the correctness of the simple synchronizer, and calculate its time and message complexities.
10. In the ideal ABD synchronizer, the physical clocks do not drift. As a result, after the initial synchronization, no messages are required to maintain the synchronization.

    Assume now that the physical clocks drift, so the difference between a pair of clocks grows at a maximum rate of **1** in **R time** units, and each process simulates a clock tick every 2δ time units, where δ is the upper bound of the message propagation delay along any link. Calculate the maximum time interval after which the predicate **synch** will fall apart.
11. Consider an array of processes $\mathbf{0}, \mathbf{1}, \mathbf{2}, \ldots, \mathbf{2N-1}$ that has a different type of synchrony requirement; we will call it interleaved synchrony. Interleaved synchrony is specified as follows: (i) Neighboring processes should not execute actions simultaneously, and (ii) between two consecutive actions by any process, all its neighbors must execute an action. When $\mathbf{N = 4}$, two sample schedules are as follows:

Schedule 1	Schedule 2
0, 2, 4, 6: tick 0	0, 2: tick 0
1, 3, 5, 7: tick 0	1, 3, 7: tick 0
0, 2, 4, 6: tick 1	0, 2: tick 1
1, 3, 5, 7: tick 1	4, 6: tick 0
	5: tick 0

The computation is an infinite execution of such a schedule. Design an algorithm to implement the interleaved synchrony of Schedule 1. In designing your solution, consider two different cases: (a) an ABD system and (b) a system without physical clocks.