Chapter 15

Basic Asynchronous Network Algorithms

In this chapter, we describe a collection of algorithms for solving some basic problems—leader election, constructing an arbitrary spanning tree, broadcast and convergecast, breadth-first search, finding shortest paths, and constructing a minimum spanning tree—in the asynchronous network model with reliable FIFO send/receive channels. The problems are, for the most part, the same ones considered in the synchronous network model in Chapter 4. As before, these problems are motivated by the need to select a process to take charge of a network computation and by the need to build structures suitable for supporting efficient communication. We do not consider faults in this chapter.

All the algorithms in this chapter are constructed by direct programming of the "bare" asynchronous network model. It will not take long for us to see that this model is much more difficult to program than the synchronous network model. This will lead us to seek ways of simplifying and systematizing the programming task. In the four chapters following this one, Chapters 16–19, we introduce four such simplification techniques: *synchronizers*, *simulating shared memory*, *logical time*, and *runtime monitoring.*

15.1 Leader Election in a Ring

We considered the problem of leader election in a synchronous ring in Chapter 3. For the asynchronous version of the problem, the underlying digraph is again a ring of n processes, numbered 1 to n in the clockwise direction. As before, we often count mod n, allowing 0 to be another name for process n, and so on. The ring can be either unidirectional or bidirectional. Figure 15.1 shows

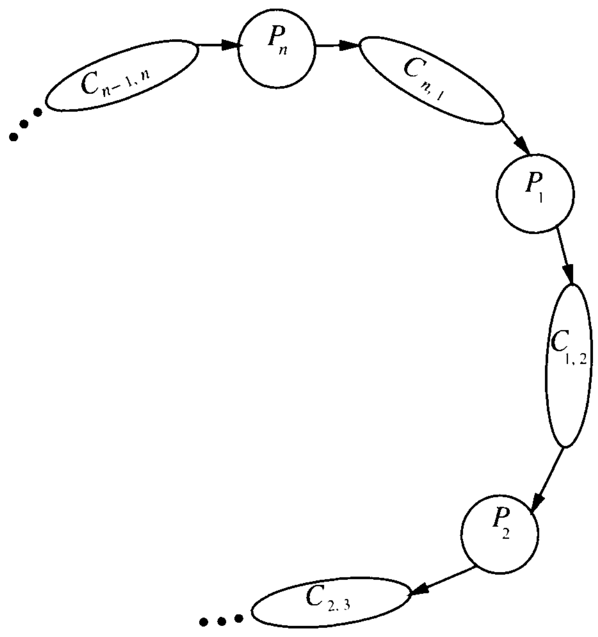

Figure 15.1: Architecture for unidirectional ring network.

the architecture for an asynchronous unidirectional ring network, including both processes and channels.

Now processes and channels are modelled as I/O automata. As in the synchronous setting, processes do not know their indices, nor those of their neighbors, but use local, relative names. This allows arbitrary processes to be arranged into a ring in an arbitrary order. Besides the *send* and *receive* actions by which process automaton P_i interacts with its channels, P_i has a $leader_i$ output action by which it can announce its election as leader. We assume, here and throughout the rest of the chapter, that the channels are reliable FIFO send/receive channels. We also assume here that the processes have UIDs. The problem is for exactly one process eventually to produce a *leader* output.

15.1.1 The *LCR* Algorithm

The *LCR* algorithm described in Section 3.3 can easily be adapted to run in an asynchronous network. Recall that in the *LCR* algorithm, each process sends its identifier around the ring. When a process receives an incoming identifier, it compares that identifier to its own. If the incoming identifier is greater than its own, it keeps passing the identifier; if it is less than its own, it discards the incoming identifier; if it is equal to its own, the process outputs *leader*.

The same idea still works in an asynchronous network; the main difference is that now each process's *send* buffer must be able to hold any number (up to n) of messages instead of just a single one. The reason for the difference is that the asynchrony can cause pileups of UIDs at nodes. We call the modified algorithm *AsynchLCR*.

In the following code, we use $AsynchLCR_i$ as an alternative name for process P_i in the *AsynchLCR* algorithm. When we discuss the algorithm, we use the two names, $AsynchLCR_i$ and P_i, as convenient; we also sometimes denote this process simply as "process i." We use similar conventions elsewhere.

$AsynchLCR_i$ automaton:

```
Signature:
Input:                                 Output:
   $receive(v)_{i-1,i}$, $v$ a UID           $send(v)_{i,i+1}$, $v$ a UID
                                          $leader_i$

States:
$u$, a UID, initially $i$'s UID
$send$, a FIFO queue of UIDs, initially containing only $i$'s UID
$status$, with values in $\{unknown, chosen, reported\}$, initially $unknown$

Transitions:
$send(v)_{i,i+1}$                                    $leader_i$
   Precondition:                                     Precondition:
      $v$ is first on $send$                              $status = chosen$
   Effect:                                           Effect:
      remove first element of $send$                      $status := reported$

$receive(v)_{i-1,i}$
   Effect:
      case
        $v > u$: add $v$ to $send$
        $v = u$: $status := chosen$
        $v < u$: do nothing
      endcase

Tasks:
$\{send(v)_{i,i+1} : v \text{ a UID}\}$
$\{leader_i\}$
```

The transitions should be self-explanatory. Process i is responsible for performing two tasks: sending messages to process $i + 1$ and announcing itself as the leader. Thus, it has two tasks, one for all its *send* actions and one for its *leader* action. The behavior of the *AsynchLCR* is essentially the same as that of *LCR*, but possibly "skewed" in time.

In order to prove that *AsynchLCR* solves the leader-election problem, we use invariant assertions as we did for the synchronous *LCR* algorithm. Invariant assertion proofs work for asynchronous networks just as well as for synchronous networks; the main difference is that now the method must be applied at a finer granularity, to reason about individual events rather than about rounds.

Technically, in order to use invariant assertion proofs, we must know the structure of the state of each channel automaton. Thus, for convenience, we assume that the channels $C_{i,i+1}$ are all *universal* FIFO reliable channels as defined in Section 14.1.2. Then we know that the state of each $C_{i,i+1}$ consists of a single *queue* component, which we refer to as $queue_{i,i+1}$. This assumption does not restrict the generality of the results, because an algorithm that works correctly with universal reliable FIFO channels must also work with arbitrary reliable FIFO channels. We will make the same assumption in all of our correctness proofs for send/receive systems with reliable FIFO channels.

Let $i_{\max}$ denote the index of the process with the maximum UID, and let $u_{\max}$ denote its UID. Here, as in the synchronous case, we must show two things:

1. No process other than $i_{\max}$ ever performs a *leader* output.

2. Process $i_{\max}$ eventually performs a *leader* output.

The first of these two conditions is a safety property while the second is a liveness property.

Lemma 15.1 *No process other than i_{max} ever performs a leader output.*

Proof. We use an invariant similar to Assertion 3.3.3 for the synchronous case. Recall that Assertion 3.3.3 said that no UID v could reach any *send* queue between $i_{\max}$ and v's original home i. Now, because the *AsynchLCR* algorithm includes channel automata, we need a slightly stronger assertion that involves the UIDs in the channel states as well as the UIDs in the process states. As usual, we subscript process state components by the index of the process; we also subscript channel state components by the two indices of the channel.

Assertion 15.1.1 *The following are true in any reachable state:*

1. *If $i \neq i_{max}$ and $j \in [i_{max}, i)$, then u_i does not appear in $send_j$.*

> 2. *If $i \neq i_{max}$ and $j \in [i_{max}, i)$, then u_i does not appear in $queue_{j,j+1}$.*

Assertion 15.1.1 is proved by induction on the number of steps in a finite execution leading to the given state. The proof is generally similar to that of Assertion 3.3.3. This time, we proceed by case analysis based on the individual *send*, *receive*, and *leader* events. The key case is that of a $receive(v)_{j-1,j}$ event where $j = i_{\max}$; for this case, we must argue that if $v = u_i$ where $i \neq i_{\max}$, then v gets discarded.

Assertion 15.1.1 can be used to prove Assertion 15.1.2.

> **Assertion 15.1.2** *The following is true in any reachable state: If $i \neq i_{max}$ then $status_i = unknown$.*

Then it is easy to see that no process other than $i_{\max}$ ever performs a $leader_i$ output, since the precondition of this action is never satisfied. □

Now we turn to the liveness property. Notice that this needs the hypothesis that the execution of *AsynchLCR* is *fair*. This formal notion means that the processes and channels continue to perform their work.

Lemma 15.2 *In any fair execution, process i_{max} eventually performs a leader output.*

Proof. The proof of this property for *AsynchLCR* is quite different from the proof of the corresponding result, Lemma 3.2, for the synchronous *LCR* algorithm. Recall that in the synchronous case, we used a very strong invariant assertion, Assertion 3.3.2, which described exactly where the maximum UID had travelled after any number r of rounds. Now we have no notion of round. Also, it is impossible to characterize precisely what happens in the computation, since the asynchrony introduces so much uncertainty. So we must use a different method.

Our proof is based on establishing intermediate milestones toward the main goal of electing a leader. In particular, we show inductively on r, for $0 \leq r \leq n-1$, that *eventually* $u_{\max}$ appears in the buffer $send_{i_{\max}+r}$. Using this claim for $r = n-1$, we show that eventually $u_{\max}$ is placed in channel $C_{i_{\max}-1,i_{\max}}$, that thereafter eventually $u_{\max}$ is received by process $i_{\max}$, and that thereafter eventually process $i_{\max}$ performs a *leader* output. The fairness properties of the process and channel I/O automata are used to prove all these eventuality claims.

For example, consider a state s in a fair execution α in which any UID v appears at the head of the $send_i$ buffer. We argue that eventually $send(v)_i$ occurs. If not, then examination of the transitions of process $AsynchLCR_i$ shows that v remains at the head of the $send_i$ buffer forever. This implies that the $send_i$

task stays enabled forever, so by fairness, some *send*$_i$ event must subsequently occur. But since v is the message at the head of the *send*$_i$ buffer, this means that *send*$(v)_i$ must eventually occur.

Also, if v appears in the kth position on the *send*$_i$ buffer, for any value of $k \geq 1$, we can show that eventually *send*$(v)_i$ occurs. This follows by induction on k, with the basis case, $k = 1$, given just above. For the inductive step, we note that a UID v in position $k > 1$ eventually reaches position $k - 1$, when the head of the buffer gets removed, and then the inductive hypothesis implies that *send*$(v)_i$ eventually occurs.

Similar arguments can be made for the UIDs in the channels. □

Putting these arguments together, we obtain

Theorem 15.3 *AsynchLCR solves the leader-election problem.*

We next consider the complexity of the *AsynchLCR* algorithm. The number of messages is $O(n^2)$, just as for the synchronous *LCR* algorithm. Recall that the time bound for *LCR* is n rounds. For the time analysis of *AsynchLCR*, we assume an upper bound of ℓ for each task of each process, and an upper bound of d on the time to deliver the oldest message in each channel queue.

A naive analysis gives an $O(n^2(\ell + d))$ time bound, by integrating time bounds into the eventuality argument in the proof of Lemma 15.2. Namely, note that the maximum length of any process *send* buffer or any channel *queue* is n. Therefore, it takes at most $n\ell$ time for a UID in a process *send* buffer to get placed in the adjacent channel, and at most nd time for a UID in a channel *queue* to get received by the next process. The overall time complexity is therefore $O(n^2(\ell + d))$.

However, it is possible to carry out a more refined analysis, yielding an upper bound that is only $O(n(\ell + d))$. The point is that although some *send* buffers and *queues* can reach size n, this cannot happen everywhere. In order for a pileup to form, some UIDs must travel *faster* than the worst-case upper bound in order to overtake others. The overall time turns out to be no worse than if the UIDs had all travelled at the same speed. We show:

Lemma 15.4 *In any fair execution, for any r, $0 \leq r \leq n - 1$, and for any i, the following are true:*

1. *By time $r(\ell + d)$, UID u_i either reaches the send$_{i+r}$ buffer or is deleted.*
2. *By time $r(\ell + d) + \ell$, UID u_i either reaches queue$_{i+r,i+r+1}$ or is deleted.*

Proof. By induction on r.

Basis: $r = 0$. UID u_i starts out in $send_i$, and within time ℓ is placed in $queue_{i,i+1}$, as needed.

Inductive step: Suppose that the claim holds for $r-1$ and prove it for r. Fix any i. For Part 1, suppose that u_i is not deleted by time $r(\ell+d)$. Then the inductive hypothesis implies that by time $t = (r-1)(\ell+d)+\ell$, UID u_i reaches $queue_{i+r-1,i+r}$.

Claim 15.5 *If u_i is not delivered to process $i+r$ by time t, then u_i reaches the head of $queue_{i+r-1,i+r}$ by time t.*

Proof. Suppose for the sake of contradiction that u_i is not delivered to process $i+r$ by time t and also does not reach the head of $queue_{i+r-1,i+r}$ by time t. Then it must be that some other UID, u_j, is ahead of u_i on $queue_{i+r-1,i+r}$ at time t. This is a pileup, where u_i has overtaken u_j; since u_i has not yet travelled distance r around the ring, it follows that u_j has not yet travelled distance $r-1$ around the ring.

However, the inductive hypothesis implies that u_j either reaches $send_{j+r-1}$ (i.e., travels at least distance $r-1$) or is deleted, by time $(r-1)(\ell+d) < t$. This implies that u_j cannot still be in $queue_{i+r-1,i+r}$ at time t, which is a contradiction. □

Thus, either u_i is delivered to process $i+r$ by time t, or else u_i reaches the head of $queue_{i+r-1,i+r}$ by time t. In this latter case, within an additional time d, u_i is delivered to process $i+r$. In either case, u_i is delivered to process $i+r$ by time $t+d = r(\ell+d)$ and placed in the $send_{i+r}$ buffer, as needed.

The proof for Part 2 is similar.

Theorem 15.6 *The time until a leader event occurs in any fair execution of AsynchLCR is at most $n(\ell+d)+\ell$, or $O(n(\ell+d))$.*

Proof. Lemma 15.4 for $r = n-1$ implies that UID $u_{\max}$ reaches $queue_{i_{\max}-1,i_{\max}}$ by time $(n-1)(\ell+d)+\ell$, and the same argument used in the proof of Lemma 15.4 implies that it reaches the first position on that queue by that time. Then within an additional time d, $u_{\max}$ is delivered to process $i_{\max}$, which then performs a *leader* output within an additional time ℓ. The total is $n(\ell+d)+\ell$, as claimed. □

Wakeups. We can modify the input/output conventions for the leader-election problem so that the inputs (here, the UIDs) arrive at the processes in special $wakeup(v)_i$ messages from an external user U, instead of originating in the start states. The correctness conditions would then be modified to assume that exactly one *wakeup*$(v)_i$ occurs for each i. Then the *AsynchLCR* algorithm can easily be modified to satisfy the new correctness conditions: each process P_i simply delays performing any locally controlled actions until after it receives its *wakeup*. If P_i receives any messages before receiving its *wakeup*, then it buffers those messages in a new *receive* buffer and processes them after receiving the *wakeup*.

A similar modification can be made to the other leader-election algorithms later in this section. More generally, any distributed problem that is formulated with inputs in the start states can be reformulated to allow the inputs to arrive in *wakeup* messages. Using the same strategy described above, we can modify any algorithm that solves the original problem so that it satisfies the new correctness conditions.

15.1.2 The *HS* Algorithm

Recall the synchronous *HS* algorithm of Section 3.4, in which each process sends exploratory messages in both directions, for successively doubled distances. It is straightforward to see that this algorithm, suitably rewritten in terms of process I/O automata, still works correctly in the asynchronous network model. Its communication complexity is $O(n \log n)$, as before. We leave the determination of an upper bound on the time complexity for an exercise.

15.1.3 The Peterson Leader-Election Algorithm

The *HS* algorithm (in both its synchronous and asynchronous versions) requires only $O(n \log n)$ messages and uses bidirectional communication. In this subsection, we present the *PetersonLeader algorithm*, which achieves $O(n \log n)$ communication complexity using only unidirectional communication. This algorithm does not rely on knowledge of n, the number of nodes in the ring. It uses comparisons of UIDs only. It elects an arbitrary process as the leader, not necessarily the process with the maximum or minimum UID. The $O(n \log n)$ communication complexity has only a small constant factor (approximately 2).

***PetersonLeader* algorithm (informal):**

While the algorithm is executing, each process is designated as being either in *active* mode or *relay* mode; all processes are initially active. The active processes carry out the "real work" of the algorithm; the relay processes just pass messages along. An execution of the *PetersonLeader* algorithm is divided into (asynchronously determined) *phases*. In each phase, the number of active processes is reduced by a factor of at least 2, so there are at most $\log n$ phases.

> In the first phase of the algorithm, each process i sends its UID two steps clockwise. Then process i compares its own UID to those of its two predecessors in the counterclockwise direction. If the counterclockwise neighbor's UID is the highest of the three, that is, if $u_{i-1} > u_{i-2}$ and $u_{i-1} > u_i$, then process i remains active, adopting the UID u_{i-1} of its counterclockwise neighbor as a new "temporary UID." On the other hand, if one of the other two UIDs is the highest of the three, then process i simply becomes a relay for the remainder of the execution.
>
> Each subsequent phase proceeds in much the same way. Each active process i now sends its temporary UID to the next and second-next active processes in the clockwise direction, and waits to learn the temporary UIDs from its two active predecessors in the counterclockwise direction. Now if the first active predecessor's UID is the largest of the three UIDs, process i remains active, adopting that predecessor's UID as its new temporary UID. On the other hand, if one of the two other UIDs is the largest of the three, then process i becomes a relay.
>
> Also, if at any phase, a process i sees that the temporary UID it receives from its immediate active predecessor is the same as its own temporary UID, then i knows that it is the only active process left. In this case, process i elects itself as the leader.

It should be clear that in any phase in which there is more than one active process, at least one process will discover a combination of UIDs that allows it to remain active at the next phase. Moreover, at most half of the active processes can survive a given phase, since every process that remains active must have an immediate active predecessor that becomes a relay.

***PetersonLeader$_i$* automaton (formal):**

Signature:

Input:	Internal:
$\quad$ *receive*$(v)_{i-1,i}$, v a UID	$\quad$ *get-second-uid*$_i$
Output:	$\quad$ *get-third-uid*$_i$
$\quad$ *send*$(v)_{i,i+1}$, v a UID	$\quad$ *advance-phase*$_i$
$\quad$ *leader*$_i$	$\quad$ *become-relay*$_i$
	$\quad$ *relay*$_i$

States:

mode $\in$ {*active*, *relay*}, initially *active*
status $\in$ {*unknown*, *chosen*, *reported*}, initially *unknown*
uid(j), $j \in \{1, 2, 3\}$, each a UID or *null*; initially *uid*$(1) = i$'s UID, *uid*$(2) =$ *uid*$(3) =$ *null*

send, a FIFO queue of UIDs, initially containing i's UID
receive, a FIFO queue of UIDs, initially empty

Transitions:

```
get-second-uid_i
  Precondition:
    mode = active
    receive is nonempty
    uid(2) = null
  Effect:
    uid(2) := first element of receive
    remove first element of receive
    add uid(2) to send
    if uid(2) = uid(1) then status := chosen

get-third-uid_i
  Precondition:
    mode = active
    receive is nonempty
    uid(2) ≠ null
    uid(3) = null
  Effect:
    uid(3) := first element of receive
    remove first element of receive

advance-phase_i
  Precondition:
    mode = active
    uid(3) ≠ null
    uid(2) > max{uid(1), uid(3)}
  Effect:
    uid(1) := uid(2)
    uid(2) := null
    uid(3) := null
    add uid(1) to send

become-relay_i
  Precondition:
    mode = active
    uid(3) ≠ null
    uid(2) ≤ max{uid(1), uid(3)}
  Effect:
    mode := relay

relay_i
  Precondition:
    mode = relay
    receive is nonempty
  Effect:
    move first element of receive
      to send

leader_i
  Precondition:
    status = chosen
  Effect:
    status := reported

send(v)_i
  Precondition:
    v is first on send
  Effect:
    remove first element of send

receive_i(v)
  Effect:
    add v to receive
```

Tasks:
$\{send(v)_{i,i+1} : v \text{ is a UID}\}$
$\{get\text{-}second\text{-}uid_i, get\text{-}third\text{-}uid_i, advance\text{-}phase_i, become\text{-}relay_i, relay_i\}$
$\{leader_i\}$

Theorem 15.7 *PetersonLeader solves the leader-election problem.*

Now we analyze the complexity. As stated above, the number of active processes is at least halved in each phase, until only one active process remains. This means that the total number of phases until a leader is elected is at most $\lfloor \log n \rfloor + 1$. During each phase, each process (either active or relay) sends at most two messages. Thus, at most $2n(\lfloor \log n \rfloor + 1)$ messages are sent in any execution of the algorithm. This is $O(n \log n)$, with a much better constant factor than in the *HS* algorithm.

For the time complexity, it is not hard to prove a naive upper bound of $O(n \log n(\ell + d))$. This is because there are $O(\log n)$ phases, and we can show that, for any p, the first p phases are completed within time $O(pn(\ell + d))$. (In each phase, each UID travels distance $O(n)$ around the ring. It takes time at most $\ell + d$ for a message to travel from one node to the next, provided that it is not blocked by a pileup. The same method we used in the proof of Lemma 15.4 can be used to argue that pileups cannot hurt the worst-case bound.)

A more refined analysis yields an upper bound of $O(n(\ell + d))$:

Theorem 15.8 *The time until a leader event occurs in any fair execution of PetersonLeader is $O(n(\ell + d))$.*

We only sketch the main ideas here, leaving the proof for a somewhat intricate exercise.

Proof Sketch. First, we can ignore pileups, since arguments such as those for Lemma 15.4 can be used to show that they do not affect the worst-case bound. The following claim is useful for the analysis.

Claim 15.9 *If processes i and j are distinct processes that are both active at phase p, then there must be some process k that is strictly after i and strictly before j in the clockwise direction, and such that process k is active at phase $p - 1$.*

The time complexity is proportional to the length of a certain chain of messages, ending with the message at the final phase p that causes the leader, i_p, to become *chosen*. The UID in that message originates at i_p itself at phase p and so travels a total distance of n at phase p. Process i_p starts this UID on its way when it enters phase p, which is just after i_p receives its *uid*(3) at phase $p - 1$. This *uid*(3) in turn originates at i_p's second predecessor that is active at phase $p - 1$, i_{p-1}, when i_{p-1} enters phase $p - 1$. By Claim 15.9, there is some process other than i_p that reaches phase $p - 1$, which implies that the greatest possible distance this UID can travel at phase $p - 1$ is n.

We continue tracing the chain backward. Process i_{p-1} enters phase $p-1$ when it receives its *uid*(3) at phase $p-2$. This *uid*(3) originates at i_{p-1}'s second predecessor that is active at phase $p-2$, i_{p-2}, when i_{p-2} enters phase $p-2$. Claim 15.9 can be used to show that i_{p-2} is no further back from i_{p-1} than i_{p-1}'s first predecessor that is active at phase $p-1$. Continuing backward, we define $i_{p-3}, \ldots, i_1$, where each i_{q-1} is no further back from i_q than i_q's first predecessor that is active at phase q.

Now, using Claim 15.9 repeatedly, it is possible to show that the total length of the chain from i_{p-1} backward to i_1 is at most n. This implies that the total length of the chain is $3n$, which translates into a time bound of $O(n(\ell + d))$. □

15.1.4 A Lower Bound on Communication Complexity

We have just described two asynchronous network leader-election algorithms, *PetersonLeader* and the asynchronous version of *HS*, that have communication complexity $O(n \log n)$. In this section, we argue that the problem also has a lower bound of $\Omega(n \log n)$. Throughout this section we assume, without loss of generality, that the channels are universal reliable FIFO channels.

Recall that we have already given two $\Omega(n \log n)$ lower bound results for leader election in the synchronous setting, Theorems 3.9 and 3.11. Theorem 3.9 gives a lower bound for algorithms that are *comparison based*; it allows bidirectional communication and allows processes to know the number of nodes in the network. This result can be carried over directly to the asynchronous setting, since the synchronous model can be formulated as a restriction of the asynchronous model.

Theorem 15.10 *Let A be a comparison-based algorithm that elects a leader in asynchronous ring networks of size n, where communication is bidirectional and n is known to the processes. Then there is a fair execution of A in which $\Omega(n \log n)$ messages are sent by the time the leader is elected.*

Theorem 3.11 gives a lower bound for algorithms that can use UIDs in arbitrary ways but that have a fixed time bound and a large space of identifiers. Again, it allows bidirectional communication and allows processes to know the number of nodes. We also carry over a version of this result to the asynchronous setting:

Theorem 15.11 *Let A be any (not necessarily comparison-based) algorithm that elects a leader in asynchronous rings of size n, where the space of UIDs is infinite, communication is bidirectional, and n is known to the processes.*

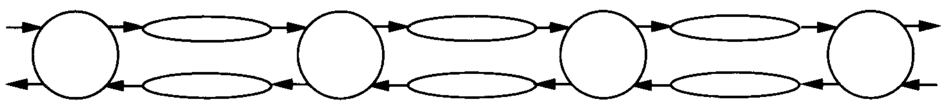

Figure 15.2: A line of automata.

Then there is a fair execution of A in which $\Omega(n \log n)$ messages are sent by the time the leader is elected.

Proof Sketch. If there is any fair execution of A in which more than $n \log n$ messages are sent by the time the leader is elected, then we are done, so assume that this is not the case. We "restrict" A to yield a synchronous algorithm S in which some message is sent at every round. Since at most $n \log n$ messages are sent in any fair execution of A by the time the leader is elected, this means that the number of rounds required for S to elect a leader is at most $n \log n$. Since the UID space is infinite, Theorem 3.11 applies to show that there is an execution of S in which $\Omega(n \log n)$ messages are sent by the time the leader is elected. This can be converted into a fair execution of A in which $\Omega(n \log n)$ messages are sent by the time the leader is elected. □

Since Theorem 3.11 appears in a starred section of this book, we present an alternative, more elementary, lower bound proof for non-comparison-based algorithms. This proof is quite different from those of Theorems 3.9 and 3.11 in that it is based on asynchrony and on the assumption that the processes do not know the size of the ring.

Theorem 15.12 *Let A be any (not necessarily comparison-based) algorithm that elects a leader in rings of arbitrary size, where the space of UIDs is infinite, communication is bidirectional, and the ring size is unknown to the processes. Then there is a fair execution of A in which $\Omega(n \log n)$ messages are sent.*

The proof requires a few preliminary definitions. Assume that we have a universal infinite set $\mathcal{P}$ of process automata. All processes in $\mathcal{P}$ are assumed to be identical except for UIDs; also, they are assumed to know their neighbors only by local names, say "right" and "left."

Our main interest is in seeing how a collection of process automata from $\mathcal{P}$ behave when they are arranged in a ring; however, it is also useful to see how they behave when arranged in a straight line, as depicted in Figure 15.2. We define a *line* to be a linear composition (using I/O automaton composition) of distinct process automata from $\mathcal{P}$, with intervening reliable FIFO send/receive channels in both directions.

We say that two lines are *disjoint* if they contain no common process automaton, that is, no common UID. If L and M are two disjoint lines of automata, we define *join*(L, M) to be the line consisting of L concatenated with M, with new reliable FIFO send/receive channels inserted between the rightmost process of L and the leftmost process of M. The *join* operator is associative, so we can extend it to apply to any number of lines. If L is any line of automata, we define *ring*(L) to be the ring consisting of L wrapped around, with new reliable FIFO send/receive channels inserted in both directions between the rightmost and leftmost processes of L. Each process's right neighbor in the line becomes its clockwise neighbor in the ring. The *ring* and *join* operations are depicted in Figure 15.3. (We now represent the channels as just arrows rather than ovals.)

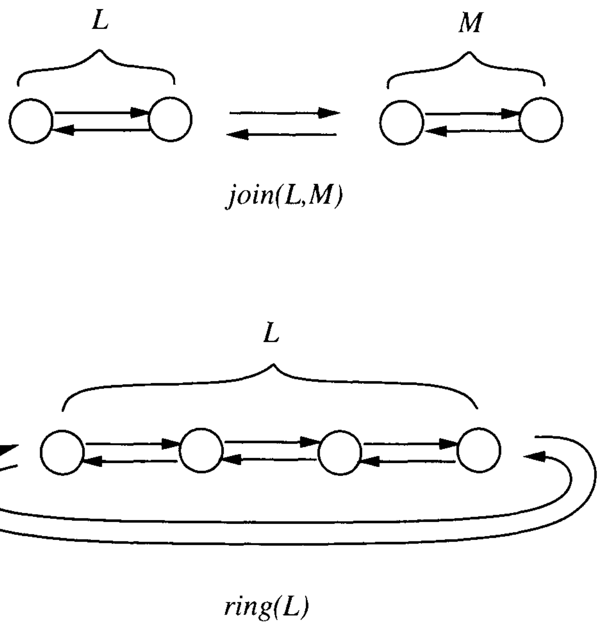

Figure 15.3: The *join* and *ring* operations.

If α is an execution of a line or ring, we define $C(\alpha)$ to be the number of messages sent in α. If R is a ring, we define $C(R)$ to be *sup*$\{C(\alpha) : \alpha$ is an execution of $R\}$, that is, the supremum of the number of messages that are sent in any execution of R. For a line, we consider the number of messages that can be sent when the line operates "in isolation," with no messages arriving at the end processes from the line's environment. Thus, if L is a line, we define $C(L)$ to be *sup*$\{C(\alpha) : \alpha$ is an input-free execution of $L\}$, that is, the supremum of the number of messages that are sent in any execution of L without any messages arriving at its endpoints from outside the line.

We say that a state s of a ring is *silent* if there is no execution fragment starting from s in which any new message is sent. We say that a state s of a line is *silent* if there is no input-free execution fragment starting from s in which any new message is sent. Note that if a ring or line is in a silent state, it does *not* mean that no further activity is possible—it just means that no further message-sending events can occur. It is still possible for processes to receive messages and perform internal steps and *leader* outputs.

We begin with a preliminary lemma.

Lemma 15.13 *There is an infinite set of process automata in $\mathcal{P}$, each of which can send at least one message without first receiving any message.*

Proof. We show something stronger: that all except possibly one process automaton in $\mathcal{P}$ can send at least one message without first receiving any message.

Suppose, to obtain a contradiction, that there are two processes in $\mathcal{P}$, say processes i and j, such that neither can send a message without first receiving one. Then consider the three rings R_1, R_2, and R_3 shown in Figure 15.4. (Now, for simplicity, we do not depict the channel automata at all.)

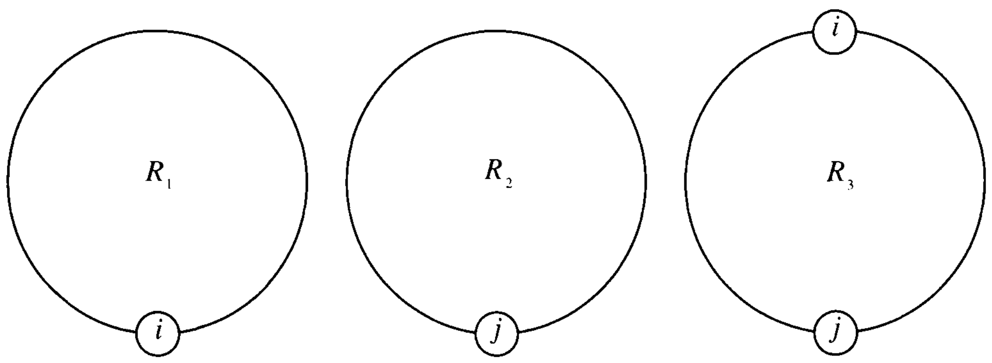

Figure 15.4: Rings R_1, R_2, and R_3 in the proof of Lemma 15.13.

Since neither i nor j can send a message unless it first receives one, no messages are ever sent in any execution of any of the three rings. Thus, the processes i and j proceed independently, performing local computation and *leader* actions, but never any communication actions. Since R_1 solves the leader-election problem, i must eventually perform a *leader* output in any fair execution of R_1. Likewise, since R_2 solves the leader-election problem, j must eventually perform a *leader* output in any fair execution of R_2. Now consider any fair execution α

of R_3. Because there is no communication, α is indistinguishable by process i from some fair execution of R_1 (using the formal notion of "indistinguishability" defined in Section 8.7), so i eventually performs a *leader* output in α. Likewise, α is indistinguishable by process j from some fair execution of R_2, so j eventually performs a *leader* output in α. But this causes two leaders to be elected in R_3, a contradiction.

We have shown that there cannot be two processes, i and j, in $\mathcal{P}$, neither of which can send a message without first receiving one. That is, there is at most one process in $\mathcal{P}$ that cannot send a message before receiving one. Since $\mathcal{P}$ is an infinite set, removing one process leaves an infinite set of processes, each of which can send a message without first receiving one. □

The proof of Theorem 15.12 uses the following key lemma.

Lemma 15.14 *For every $r \geq 0$, there is an infinite collection of pairwise-disjoint lines, $\mathcal{L}_r$, such that for every $L \in \mathcal{L}_r$, it is the case that $|L| = 2^r$ and $C(L) \geq r2^{r-2}$.*

Proof. By induction on r.

Basis: $r = 0$. Let $\mathcal{L}_0$ be the set of all single-node lines corresponding to all the processes in $\mathcal{P}$. The claim is trivial.

Basis: $r = 1$. Let $\mathcal{L}_1$ be any infinite collection of disjoint two-node lines composed of processes each of which can send a message without first receiving one. The existence of this collection is implied by Lemma 15.13. Then if L is any line from $\mathcal{L}_1$, there must be an input-free execution of L in which at least one message is sent: simply let one of the two processes send a message without first receiving one. This suffices.

Inductive step: Assume that $r \geq 2$ and the lemma is true for $r - 1$, that is, that there is an infinite collection of pairwise-disjoint lines, $\mathcal{L}_{r-1}$, such that for every $L \in \mathcal{L}_{r-1}$, it is the case that $|L| = 2^{r-1}$ and $C(L) \geq (r-1)2^{r-3}$. Let $n = 2^r$.

Let L, M, and N be any three lines from $\mathcal{L}_{r-1}$. We consider the six possible joins of two of these three lines: $join(L, M)$, $join(M, L)$, $join(L, N)$, $join(N, L)$, $join(M, N)$, and $join(N, M)$. We show the following claim.

Claim 15.15 *At least one of these six lines has an input-free execution in which at least $\frac{n}{4} \log n = r2^{r-2}$ messages are sent.*

The lemma then follows from Claim 15.15, because infinitely many sets of three lines can be chosen from $\mathcal{L}_{r-1}$ without reusing any processes.

Proof (of Claim 15.15). Assume the contrary, that none of these six lines can be made to send as many as $\frac{n}{4}\log n$ messages. By the inductive hypothesis, there is a finite input-free execution α_L of L for which $C(\alpha_L) \geq (r-1)2^{r-3} = \frac{n}{8}\log\frac{n}{2}$. We can assume without loss of generality that the final state of α_L is silent, since otherwise α_L could be extended to a longer finite execution in which more messages are generated. (This extension cannot go on indefinitely, since we know that L alone cannot send as many as $\frac{n}{4}\log n$ messages.) Similarly, we obtain finite input-free executions α_M of M and α_N of N with the same properties.

Now we construct a finite execution $\alpha_{L,M}$ of the line *join*(L, M). Execution $\alpha_{L,M}$ starts by running α_L on L and α_M on M, delaying all messages sent on the channels connecting the lines L and M. In this prefix of $\alpha_{L,M}$, at least $2(\frac{n}{8})\log\frac{n}{2} = \frac{n}{4}(\log n - 1)$ messages are sent.

Next, $\alpha_{L,M}$ continues to a silent state. Note that, however this happens, the number of additional messages that are sent in the extension must be strictly less than $\frac{n}{4}$, because otherwise the total number of messages in $\alpha_{L,M}$ would be at least $\frac{n}{4}\log n$, contradicting our assumption.

The particular way in which we make this extension is to allow only the $\frac{n}{4}-1$ processes of L and the $\frac{n}{4}-1$ processes of M that are closest to the junction of L and M to take steps after α_L and α_M, until the system reaches a state from which none of these processes can send any more messages. We claim that the resulting state of *join*(L, M) must be silent. For if not, then a series of at least $\frac{n}{4}-1$ messages must have been sent after the initial α_L and α_M, conveying information about the junction to a process at distance $\frac{n}{4}$ from the junction and enabling that process to send yet another message. (Convince yourself of this.) But this is a total of at least $\frac{n}{4}$ additional messages in the extension of α_L and α_M, which is impossible. So the indicated state of *join*(L, M) must be silent.

Informally speaking, after $\alpha_{L,M}$, information about the junction of lines L and M has not reached either the midpoint of L or the midpoint of M. Only the $\frac{n}{4}$ processes on either side of the junction can know about the junction, and the two processes at distance exactly $\frac{n}{4}$ from the junction cannot send any new messages as a result of this knowledge. Figure 15.5 depicts the junction of L and M, for the case where $n = 16$.

In a similar way, we define finite executions $\alpha_{M,L}$, $\alpha_{L,N}$, and so on.

Now we combine the lines L, M, and N into several different rings to obtain a contradiction. First define R_1 to be *ring*(*join*(L, M, N)), as depicted in Figure 15.6. Define a fair execution α_1 of R_1, as follows. Execution α_1 begins with α_L, α_M, and α_N, thus making the three separate lines L, M, and N silent. Then α_1 continues as in $\alpha_{L,M}$, $\alpha_{M,N}$, and $\alpha_{N,L}$. Since the processes that learn about each junction extend at most halfway into each of the adjacent lines, there

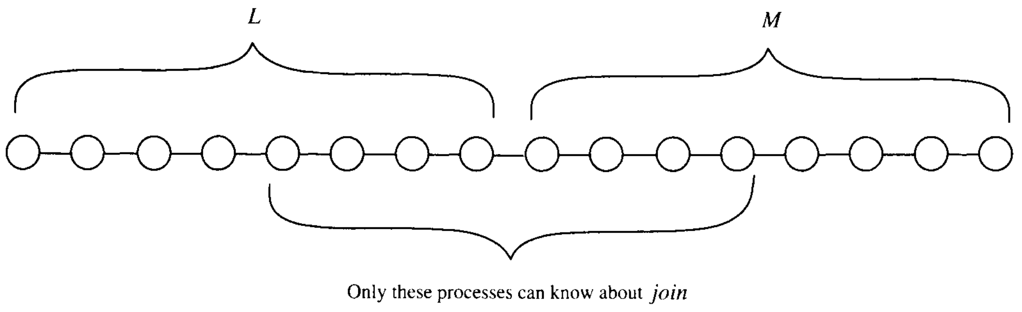

Figure 15.5: $\alpha_{L,M}$

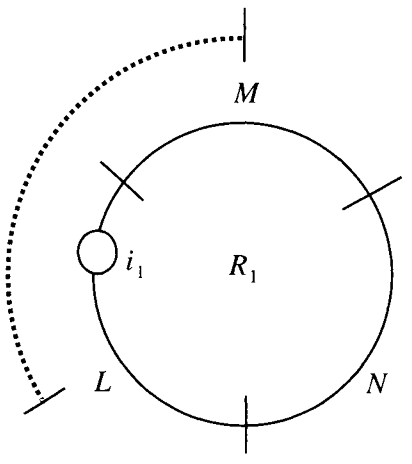

Figure 15.6: $R_1 = ring(join(L, M, N))$.

is no interference among these three extensions. Furthermore, after these three extensions, the entire ring is silent. Then α_1 continues in any fair manner. The correctness conditions imply that some leader, say i_1, is elected in α_1. We may assume without loss of generality that process i_1 is between the midpoint of L and the midpoint of M, as depicted in Figure 15.6.

Next we define $R_2 = ring(join(L, N, M))$, and define a fair execution α_2 of R_2 analogous to α_1 (this time using α_L, α_M, α_N, $\alpha_{L,N}$, $\alpha_{N,M}$, and $\alpha_{M,L}$). Then some leader, say i_2, is elected in α_2 (see Figure 15.7).

Next define $R_3 = ring(join(M, N))$, and define a fair execution α_3 of R_3 (using α_M, α_N, $\alpha_{M,N}$, and $\alpha_{N,M}$). Again, some leader, say i_3, must be elected

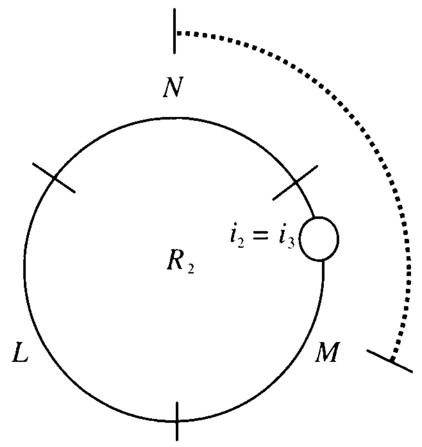

Figure 15.7: $R_2 = ring(join(L, N, M))$.

in α_3 (see Figure 15.8). We claim that i_3 must be in the lower half of R_3 as it is drawn in Figure 15.8, that is, somewhere between the midpoint of N and the midpoint of M, moving clockwise. For if i_3 were in the upper half of R_3, then α_1 and α_3 would be indistinguishable to process i_3, so i_3 would also be elected in α_1. But then two distinct processes, i_1 and i_3, would be elected in α_1, a contradiction. (Processes i_1 and i_3 are distinct because i_1 is between the midpoints of L and M, while i_3 is between the midpoints of M and N.)

Since i_3 is in the lower half of R_3, α_2 and α_3 are indistinguishable to i_3; hence, i_3 is also elected in R_2. Note that i_3 is between the midpoint of N and the midpoint of M in R_2. Since only one leader can be elected in α_2, we have $i_2 = i_3$. See Figure 15.7.

Finally, we define $R_4 = ring(join(L, N))$ and define a fair execution α_4 of R_4 (using α_L, α_N, $\alpha_{L,N}$, and $\alpha_{N,L}$). See Figure 15.9. We claim that no leader can be elected in α_4. For if a leader were elected from the top half of R_4, then that leader would also be elected in α_2, yielding two leaders in α_2. And if a leader were elected from the bottom half of R_4, then that leader would also be elected in α_1, yielding two leaders in α_1. Either way is a contradiction.

But the fact that no leader is elected in α_4 violates the problem requirements, which yields the contradiction needed to prove the claim. □

The lemma now follows immediately from Claim 15.15, as described just before the proof of the claim.

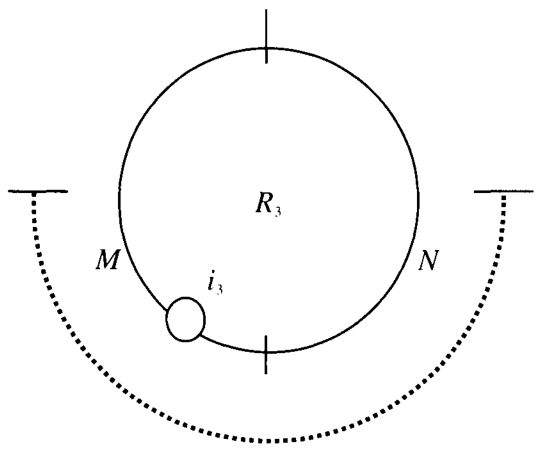

Figure 15.8: $R_3 = ring(join(M, N))$.

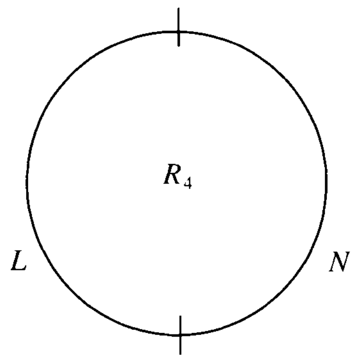

Figure 15.9: $R_4 = ring(join(L, N))$.

Now using Lemma 15.14, it is easy to complete the proof of Theorem 15.12.

Proof (of Theorem 15.12). First suppose that n is a power of 2, say $n = 2^r$. Let L be any line in $\mathcal{L}_r$. Lemma 15.14 implies that $|L| = n$ and $C(L) \geq \frac{n}{4} \log n$. Let α be an input-free execution of L such that $C(\alpha) \geq \frac{n}{4} \log n$. Define $R = ring(L)$, that is, paste L into a ring. Define an execution α' of R that behaves exactly like α on L, delaying all messages across the junction between the endpoints of L until after at least $\frac{n}{4} \log n$ messages have been sent. Then $C(\alpha') \geq \frac{n}{4} \log n$, which proves that $C(R) \geq \frac{n}{4} \log n$.

We leave the argument for values of n that are non-powers of 2 for an exercise.

□

Note the crucial parts played in the proof of Theorem 15.12 by the asynchrony and the unknown ring size.

15.2 Leader Election in an Arbitrary Network

So far in this chapter, we have considered algorithms for electing a leader in an asynchronous ring network. Now we will consider the leader-election problem in networks based on more general graphs. We assume in this section that the underlying graph is undirected, that is, that there is bidirectional communication on all the edges, and that it is connected. Processes are assumed to be identical except for UIDs.

Recall the *FloodMax* algorithm for synchronous networks from Section 4.1.2. It requires that processes know *diam*, the diameter of the network. In that algorithm, every process maintains a record of the maximum UID it has seen so far, initially its own. At each synchronous round, the process sends this maximum on all its channels. The algorithm terminates after *diam* rounds; the unique process that has its own UID as its known maximum then announces itself as the leader.

The *FloodMax* algorithm does not extend directly to the asynchronous setting, because there are no rounds in the asynchronous model. However, it is possible to simulate the rounds asynchronously. We simply require each process that sends a round r message to tag that message with its round number r. The recipient waits to receive round r messages from all its neighbors before performing its round r transition. By simulating *diam* rounds, the algorithm can terminate correctly.

In the synchronous setting, we described an optimization of *FloodMax* called *OptFloodMax*, in which each process only sends messages when it has new information, that is, when its maximum UID has just changed. It is not clear how to simulate this optimized version in an asynchronous network. If we simply tag messages with round numbers as for *FloodMax*, then a process that does not hear from all its neighbors at a round r cannot determine when it has received all its incoming messages for round r, so it cannot tell when it can perform its round r transition. We can, of course, add dummy messages between pairs of neighbors that do not otherwise communicate, but that destroys the optimization.

Alternatively, we can simulate *OptFloodMax* purely asynchronously—whenever a process obtains a new maximum UID, it sends that UID to its neighbors at some later time. This strategy will indeed eventually propagate the maximum to