3

Leader Election in Rings

In this chapter, we consider systems in which the topology of the message passing system is a ring. Rings are a convenient structure for message-passing systems and correspond to physical communication systems, for example, token rings. We investigate the *leader election* problem, in which a group of processors must choose one among them to be the leader. The existence of a leader can simplify coordination among processors and is helpful in achieving fault tolerance and saving resources—recall how the existence of the special processor p_r made possible a simple solution to the broadcast problem in Chapter 2. Furthermore, the leader election problem represents a general class of symmetry-breaking problems. For example, when a deadlock is created, because of processors waiting in a cycle for each other, the deadlock can be broken by electing one of the processors as a leader and removing it from the cycle.

3.1 THE LEADER ELECTION PROBLEM

The leader election problem has several variants, and we define the most general one below. Informally, the problem is for each processor eventually to decide that either it is the leader or it is not the leader, subject to the constraint that exactly one processor decides that it is the leader. In terms of our formal model, an algorithm is said to solve the leader election problem if it satisfies the following conditions:

- The terminated states are partitioned into *elected* and *not-elected* states. Once a processor enters an elected (respectively, not-elected) state, its transition

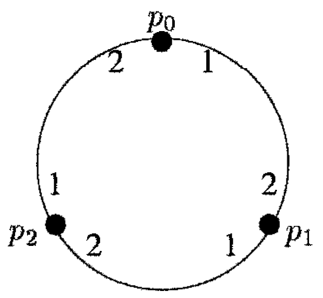

Fig. 3.1 A simple oriented ring.

function will only move it to another (or the same) elected (respectively, not-elected) state.

- In every admissible execution, exactly one processor (the *leader*) enters an elected state and all the remaining processors enter a not-elected state.

We restrict our attention to the situation in which the topology of the system is a ring. In particular, we assume that the edges in the topology graph go between p_i and p_{i+1}, for all i, $0 \leq i < n$, where addition is mod n. Furthermore, we assume that processors have a consistent notion of left and right, resulting in an *oriented* ring. Formally, this assumption is modeled by requiring that, for every i, $0 \leq i < n$, p_i's channel to p_{i+1} is labeled 1, also known as *left* or *clockwise*, and p_i's channel to p_{i-1} is labeled 2, also known as *right* or *counterclockwise* (as usual, addition and subtraction are mod n). Figure 3.1 contains a simple example of a three-node ring. (See the chapter notes for more on orientation.)

3.2 ANONYMOUS RINGS

A leader election algorithm for a ring system is *anonymous* if processors do not have unique identifiers that can be used by the algorithm. More formally, every processor in the system has the same state machine. In describing anonymous algorithms, recipients of messages can be specified only in terms of channel labels, for example, left and right neighbors.

A potentially useful piece of information for an algorithm is n, the number of processors. If n is not known to the algorithm, that is, n is not hardcoded in advance, the algorithm is said to be "uniform," because the algorithm looks the same for every value of n. Formally, in an anonymous *uniform* algorithm, there is only one state machine for all processors, no matter what the ring size. In an anonymous *nonuniform* algorithm, for each value of n, the ring size, there is a single state machine, but there can be different state machines for different ring sizes, that is, n can be explicitly present in the code.

We show that there is no anonymous leader election algorithm for ring systems.

For generality and simplicity, we prove the result for nonuniform algorithms and synchronous rings. Impossibility for synchronous rings immediately implies the same result for asynchronous rings (see Exercise 3.1). Similarly, impossibility for nonuniform algorithms, that is, algorithms in which n, the number of processors, is known, implies impossibility for algorithms when n is unknown (see Exercise 3.2).

Recall that in a synchronous system, an algorithm proceeds in rounds, where in each round all pending messages are delivered, following which every processor takes one computation step. The initial state of a processor includes in the *outbuf* variables any messages that are to be delivered to the processor's right and left neighbors in the first round.

The idea behind the impossibility result is that in an anonymous ring, the symmetry between the processors can always be maintained; that is, without some initial asymmetry, such as provided by unique identifiers, symmetry cannot be broken. Specifically, all processors in the anonymous ring algorithm start in the same state. Because they are identical and execute the same program (i.e., they have the same state machine), in every round each of them sends exactly the same messages; thus they all receive the same messages in each round and change state identically. Consequently, if one of the processors is elected, then so are all the processors. Hence, it is impossible to have an algorithm that elects a single leader in the ring.

To formalize this intuition, consider a ring R of size $n > 1$ and assume, by way of contradiction, that there exists an anonymous algorithm, A, for electing a leader in this ring. Because the ring is synchronous and there is only one initial configuration, there is a unique admissible execution of A on R.

Lemma 3.1 *For every round k of the admissible execution of A in R, the states of all the processors at the end of round k are the same.*

Proof. The proof is by induction on k. The base case, $k = 0$ (before the first round), is straightforward because the processors begin in the same initial state.

For the inductive step, assume the lemma holds for round $k - 1$. Because the processors are in the same state in round $k - 1$, they all send the same message m_r to the right and the same message m_ℓ to the left. In round k, every processor receives the message m_ℓ on its right edge and the message m_r on its left edge. Thus all processors receive exactly the same messages in round k; because they execute the same program, they are in the same state at the end of round k. □

The above lemma implies that if at the end of some round some processor announces itself as a leader, by entering an elected state, so do all other processors. This contradicts the assumption that A is a leader election algorithm and proves:

Theorem 3.2 *There is no nonuniform anonymous algorithm for leader election in synchronous rings.*

3.3 ASYNCHRONOUS RINGS

This section presents upper and lower bounds on the message complexity for the leader election problem in asynchronous rings. Because Theorem 3.2 just showed that there is no anonymous leader election algorithm for rings, we assume in the remainder of this chapter that processors have unique identifiers.

We assume that each processor in a ring has a unique identifier. Every natural number is a possible identifier. When a state machine (local program) is associated with each processor p_i, there is a distinguished state component id_i that is initialized to the value of that identifier.

We will specify a ring by listing the processors' identifiers in clockwise order, beginning with the smallest identifier. Thus each processor p_i, $0 \leq i < n$, is assigned an identifier id_i. Note that two identifier assignments, one of which is a cyclic shift of the other, result in the same ring by this definition, because the indices of the underlying processors (e.g., the 97 of processor p_{97}) are not available.

The notions of uniform and nonuniform algorithms are slightly different when unique identifiers are available.

A (non-anonymous) algorithm is said to be *uniform* if, for every identifier, there is a state machine and, regardless of the size of the ring, the algorithm is correct when processors are assigned the unique state machine for their identifier. That is, there is only one local program for a processor with a given identifier, no matter what size ring the processor is a part of.

A (non-anonymous) algorithm is said to be *nonuniform* if, for every n and every identifier, there is a state machine. For every n, given any ring of size n, the algorithm in which every processor has the state machine for its identifier *and for ring size* n must be correct.

We start with a very simple leader election algorithm for asynchronous rings that requires $O(n^2)$ messages. This algorithm motivates a more efficient algorithm that requires $O(n \log n)$ messages. We show that this algorithm has optimal message complexity by proving a lower bound of $\Omega(n \log n)$ on the number of messages required for electing a leader.

3.3.1 An $O(n^2)$ Algorithm

In this algorithm, each processor sends a message with its identifier to its left neighbor and then waits for messages from its right neighbor. When it receives such a message, it checks the identifier in this message. If the identifier is greater than its own identifier, it forwards the message to the left; otherwise, it "swallows" the message and does not forward it. If a processor receives a message with its own identifier, it declares itself a leader by sending a termination message to its left neighbor and terminating as a leader. A processor that receives a termination message forwards it to the left and terminates as a non-leader. Note that the algorithm does not depend on the size of the ring, that is, it is uniform.

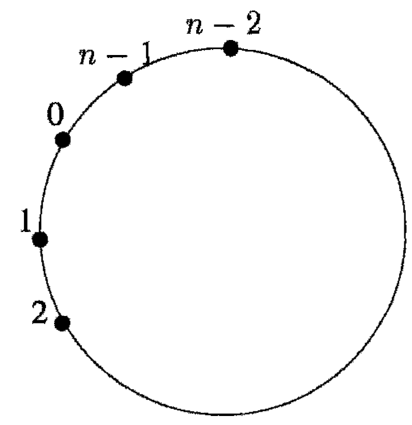

Fig. 3.2 Ring with $\Theta(n^2)$ messages.

Note that, in any admissible execution, only the message of the processor with the maximal identifier is never swallowed. Therefore, only the processor with the maximal identifier receives a message with its own identifier and will declare itself as a leader. All the other processors receive termination messages and are not chosen as leaders. This implies the correctness of the algorithm.

Clearly, the algorithm never sends more than $O(n^2)$ messages in any admissible execution. Moreover, there is an admissible execution in which the algorithm sends $\Theta(n^2)$ messages: Consider the ring where the identifiers of the processors are $0, \ldots, n-1$ and they are ordered as in Figure 3.2. In this configuration, the message of processor with identifier i is sent exactly $i+1$ times. Thus the total number of messages, including the n termination messages, is $n + \sum_{i=0}^{n-1}(i+1) = \Theta(n^2)$.

3.3.2 An $O(n \log n)$ Algorithm

A more efficient algorithm is based on the same idea as the algorithm we have just seen. Again, a processor sends its identifier around the ring and the algorithm guarantees that only the message of the processor with the maximal identifier traverses the whole ring and returns. However, the algorithm employs a more clever method for forwarding identifiers, thus reducing the worst-case number of messages from $O(n^2)$ to $O(n \log n)$.

To describe the algorithm, we first define the *k-neighborhood* of a processor p_i in the ring to be the set of processors that are at distance at most k from p_i in the ring (either to the left or to the right). Note that the k-neighborhood of a processor includes exactly $2k+1$ processors.

The algorithm operates in phases; it is convenient to start numbering the phases with 0. In the kth phase a processor tries to become a *winner* for that phase; to be a winner, it must have the largest id in its 2^k-neighborhood. Only processors that are winners in the kth phase continue to compete in the $(k+1)$-st phase. Thus fewer processors proceed to higher phases, until at the end, only one processor is a winner and it is elected as the leader of the whole ring.

In more detail, in phase 0, each processor attempts to become a phase 0 winner and sends a ⟨probe⟩ message containing its identifier to its 1-neighborhood, that is, to each of its two neighbors. If the identifier of the neighbor receiving the probe is greater than the identifier in the probe, it swallows the probe; otherwise, it sends back a ⟨reply⟩ message. If a processor receives a reply from both its neighbors, then the processor becomes a phase 0 winner and continues to phase 1.

In general, in phase k, a processor p_i that is a phase $k-1$ winner sends ⟨probe⟩ messages with its identifier to its 2^k-neighborhood (one in each direction). Each such message traverses 2^k processors one by one. A probe is swallowed by a processor if it contains an identifier that is smaller than its own identifier. If the probe arrives at the last processor in the neighborhood without being swallowed, then that last processor sends back a ⟨reply⟩ message to p_i. If p_i receives replies from both directions, it becomes a phase k winner, and it continues to phase $k+1$. A processor that receives its own ⟨probe⟩ message terminates the algorithm as the leader and sends a termination message around the ring.

Note that in order to implement the algorithm, the last processor in a 2^k-neighborhood must return a reply rather than forward a ⟨probe⟩ message. Thus we have three fields in each ⟨probe⟩ message: the identifier, the phase number, and a hop counter. The hop counter is initialized to 0, and is incremented by 1 whenever a processor forwards the message. If a processor receives a phase k message with a hop counter 2^k, then it is the last processor in the 2^k-neighborhood.

The pseudocode appears in Algorithm 5. Phase k for a processor corresponds to the period between its sending of a ⟨probe⟩ message in line 4 or 15 with third parameter k and its sending of a ⟨probe⟩ message in line 4 or 15 with third parameter $k+1$. The details of sending the termination message around the ring have been left out in the code, and only the leader terminates.

The correctness of the algorithm follows in the same manner as in the simple algorithm, because they have the same swallowing rules. It is clear that the probes of the processor with the maximal identifier are never swallowed; therefore, this processor will terminate the algorithm as a leader. On the other hand, it is also clear that no other ⟨probe⟩ can traverse the whole ring without being swallowed. Therefore, the processor with the maximal identifier is the only leader elected by the algorithm.

To analyze the worst-case number of messages that is sent during any admissible execution of the algorithm, we first note that the probe distance in phase k is 2^k, and thus the number of messages sent on behalf of a particular competing processor in phase k is $4 \cdot 2^k$. How many processors compete in phase k, in the worst case? For $k = 0$, the number is n, because all processors could begin the algorithm. For $k \geq 1$, every processor that is a phase $k-1$ winner competes in phase k. The next lemma gives an upper bound on the number of winners in each phase.

Lemma 3.3 *For every $k \geq 1$, the number of processors that are phase k winners is at most $\frac{n}{2^k+1}$.*

Proof. If a processor p_i is a phase k winner, then every processor in p_i's 2^k-neighborhood must have an id smaller than p_i's id. The closest together that two

Algorithm 5 Asynchronous leader election: code for processor p_i, $0 \leq i < n$.

```
Initially, asleep = true

upon receiving no message:
    if asleep then
        asleep := false
        send ⟨probe,id,0,1⟩ to left and right

upon receiving ⟨probe,j,k,d⟩ from left (resp., right):
    if $j = id$ then terminate as the leader
    if $j > id$ and $d < 2^k$ then                      // forward the message
        send ⟨probe,j,k,d + 1⟩ to right (resp., left)    // increment hop counter
    if $j > id$ and $d \geq 2^k$ then                   // reply to the message
        send ⟨reply,j,k⟩ to left (resp., right)
                                            // if $j < id$, message is swallowed

upon receiving ⟨reply,j,k⟩ from left (resp., right):
    if $j \neq id$ then send ⟨reply,j,k⟩ to right (resp., left)   // forward the reply
    else                                                 // reply is for own probe
        if already received ⟨reply,j,k⟩ from right (resp., left) then
            send ⟨probe,id,k + 1,1⟩                          // phase k winner
```

phase k winners, p_i and p_j, can be is if the left side of p_i's 2^k-neighborhood is exactly the right side of p_j's 2^k-neighborhood. That is, there are 2^k processors in between p_i and p_j. The maximum number of phase k winners is achieved when this dense packing continues around the ring. The number of winners in this case is $\frac{n}{2^k+1}$. □

By the previous lemma, there is only one winner once the phase number is at least $\log(n-1)$. In the next phase, the winner elects itself as leader. The total number of messages then, including the $4n$ phase 0 messages and n termination messages, is at most:

$$5n + \sum_{k=1}^{\lceil \log(n-1) \rceil + 1} 4 \cdot 2^k \cdot \frac{n}{2^{k-1}+1} < 8n(\log n + 2) + 5n$$

To conclude, we have the following theorem:

Theorem 3.4 *There is an asynchronous leader election algorithm whose message complexity is $O(n \log n)$.*

Note that, in contrast to the simple algorithm of Section 3.3.1, this algorithm uses bidirectional communication on the ring. The message complexity of this algorithm is not optimal with regard to the constant factor, 8; the chapter notes discuss papers that achieve smaller constant factors.

3.3.3 An $\Omega(n \log n)$ Lower Bound

In this section, we show that the leader election algorithm of Section 3.3.2 is asymptotically optimal. That is, we show that any algorithm for electing a leader in an asynchronous ring sends at least $\Omega(n \log n)$ messages. The lower bound we prove is for uniform algorithms, namely, algorithms that do not know the size of the ring.

We prove the lower bound for a special variant of the leader election problem, where the elected leader must be the processor with the maximum identifier in the ring; in addition, all the processors must know the identifier of the elected leader. That is, before terminating each processor writes to a special variable the identity of the elected leader. The proof of the lower bound for the more general definition of the leader election problem follows by reduction and is left as Exercise 3.5.

Assume we are given a uniform algorithm A that solves the above variant of the leader election problem. We will show that there exists an admissible execution of A in which $\Omega(n \log n)$ messages are sent. Intuitively, this is done by building a "wasteful" execution of the algorithm for rings of size $n/2$, in which many messages are sent. Then we "paste together" two different rings of size $n/2$ to form a ring of size n, in such a way that we can combine the wasteful executions of the smaller rings and force $\Theta(n)$ additional messages to be received.

Although the preceding discussion referred to pasting together executions, we will actually work with schedules. The reason is that executions include configurations, which pin down the number of processors in the ring. We will want to apply the same sequence of events to different rings, with different numbers of processors. Before presenting the details of the lower bound proof, we first define schedules that can be "pasted together."

Definition 3.1 *A schedule σ of A for a particular ring is* open *if there exists an edge e of the ring such that in σ no message is delivered over the edge e in either direction; e is an* open *edge of σ.*

Note that an open schedule need not be admissible; in particular, it can be finite, and processors may not have terminated yet.

Intuitively, because the processors do not know the size of the ring, we can paste together two open schedules of two small rings to form an open schedule of a larger ring. Note that this argument relies on the fact that the algorithm is uniform and works in the same manner for every ring size.

We now give the details. For clarity of presentation, we assume that n is an integral power of 2 for the rest of the proof. (Exercise 3.6 asks you to prove the lower bound for other values of n.)

Theorem 3.5 *For every n and every set of n identifiers, there is a ring using those identifiers that has an open schedule of A in which at least $M(n)$ messages are received, where $M(2) = 1$ and $M(n) = 2M(\frac{n}{2}) + \frac{1}{2}(\frac{n}{2} - 1)$ for $n > 2$.*

Since $M(n) = \Theta(n \log n)$, this theorem implies the desired lower bound. The proof of the theorem is by induction. Lemma 3.6 is the base case ($n = 2^1$) and

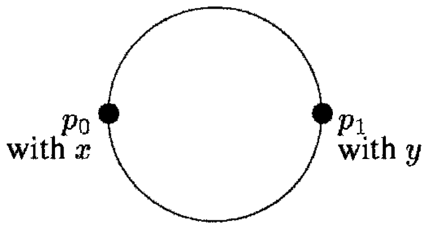

Fig. 3.3 Illustration for Lemma 3.6.

Lemma 3.7 is the inductive step ($n = 2^i, i > 1$). For the base case ring consisting of two processors, we assume that there are actually two distinct links connecting the processors.

Lemma 3.6 *For every set consisting of two identifiers, there is a ring R using those two identifiers that has an open schedule of A in which at least one message is received.*

Proof. Assume R contains processors p_0 and p_1 and the identifier of p_0 (say, x) is larger than the identifier of p_1 (say, y) (see Fig. 3.3).

Let α be an admissible execution of A on the ring. Since A is correct, eventually p_1 must write p_0's identifier x in α. Note that at least one message must be received in α; otherwise, if p_1 does not get a message from p_0 it cannot discover that the identifier of p_0 is x. Let σ be the shortest prefix of the schedule of α that includes the first event in which a message is received. Note that the edge other than the one over which the first message is received is open. Since exactly one message is received in σ and one edge is open, σ is clearly an open schedule that satisfies the requirements of the lemma. □

The next lemma provides the inductive step of the pasting procedure. As mentioned above, the general approach is to take two open schedules on smaller rings in which many messages are received and to paste them together at the open edges into an open schedule on the bigger ring in which the same messages plus extra messages are received. Intuitively, one can see that two open schedules can be pasted together and still behave the same (this will be proved formally below). The key step, however, is forcing the additional messages to be received. After the two smaller rings are pasted together, the processors in the half that does not contain the eventual leader must somehow learn the id of the eventual leader, and this can only occur through message exchanges. We unblock the messages delayed on the connecting open edges and continue the schedule, arguing that many messages must be received. Our main problem is how to do this in a way that will yield an open schedule on the bigger ring so that the lemma can be applied inductively. The difficulty is that if we pick in advance which of the two edges connecting the two parts to unblock, then the algorithm can choose to wait for information on the other edge. To avoid this problem, we first create a 'test' schedule, learning which of the two edges, when

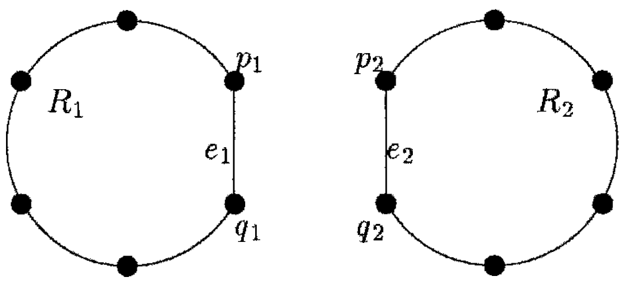

Fig. 3.4 R_1 and R_2.

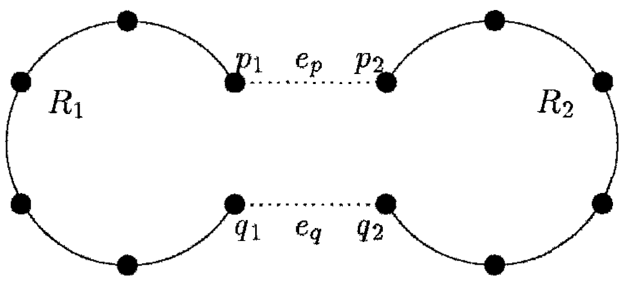

Fig. 3.5 Pasting together R_1 and R_2 into R.

unblocked, causes the larger number of messages to be received. We then go back to our original pasted schedule and only unblock that edge.

Lemma 3.7 *Choose $n > 2$. Assume that for every set of $\frac{n}{2}$ identifiers, there is a ring using those identifiers that has an open schedule of A in which at least $M(\frac{n}{2})$ messages are received. Then for every set of n identifiers, there is a ring using those identifiers that has an open schedule of A in which at least $2M(\frac{n}{2}) + \frac{1}{2}(\frac{n}{2} - 1)$ messages are received.*

Proof. Let S be a set of n identifiers. Partition S into two sets S_1 and S_2, each of size $\frac{n}{2}$. By assumption, there exists a ring R_1 using the identifiers in S_1 that has an open schedule σ_1 of A in which at least $M(\frac{n}{2})$ messages are received. Similarly, there exists ring R_2 using the identifiers in S_2 that has an open schedule σ_2 of A in which at least $M(\frac{n}{2})$ messages are received. Let e_1 and e_2 be the open edges of σ_1 and σ_2, respectively. Denote the processors adjacent to e_1 by p_1 and q_1 and the processors adjacent to e_2 by p_2 and q_2. Paste R_1 and R_2 together by deleting edges e_1 and e_2 and connecting p_1 to p_2 with edge e_p and q_1 to q_2 with edge e_q; denote the resulting ring by R. (This is illustrated in Figs. 3.4 and 3.5.)

We now show how to construct an open schedule σ of A on R in which $2M(\frac{n}{2}) + \frac{1}{2}(\frac{n}{2} - 1)$ messages are received. The idea is to first let each of the smaller rings execute its wasteful open schedule separately.

We now explain why σ_1 followed by σ_2 constitutes a schedule for A in the ring R. Consider the occurrence of the event sequence σ_1 starting in the initial configuration for ring R. Since the processors in R_1 cannot distinguish during these events whether R_1 is an independent ring or a sub-ring of R, they execute σ_1 exactly as though R_1

was independent. Consider the subsequent occurrence of the event sequence σ_2 in the ring R. Again, since no messages are delivered on the edges that connect R_1 and R_2, processors in R_2 cannot distinguish during these events whether R_2 is an independent ring or a sub-ring of R. Note the crucial dependence on the uniformity assumption.

Thus $\sigma_1\sigma_2$ is a schedule for R in which at least $2M(\frac{n}{2})$ messages are received.

We now show how to force the algorithm into receiving $\frac{1}{2}(\frac{n}{2}-1)$ additional messages by unblocking either e_p or e_q, but not both.

Consider every finite schedule of the form $\sigma_1\sigma_2\sigma_3$ in which e_p and e_q both remain open. If there is a schedule in which at least $\frac{1}{2}(\frac{n}{2}-1)$ messages are received in σ_3, then the lemma is proved.

Suppose there is no such schedule. Then there exists some schedule $\sigma_1\sigma_2\sigma_3$ that results in a "quiescent" configuration in the corresponding execution. A processor state is said to be *quiescent* if there is no sequence of computation events from that state in which a message is sent. That is, the processor will not send another message until it receives a message. A configuration is said to be *quiescent* (with respect to e_p and e_q) if no messages are in transit except on the open edges e_p and e_q and every processor is in a quiescent state.

Assume now, without loss of generality, that the processor with the maximal identifier in R is in the sub-ring R_1. Since no message is delivered from R_1 to R_2, processors in R_2 do not know the identifier of the leader, and therefore no processor in R_2 can terminate at the end of $\sigma_1\sigma_2\sigma_3$ (as in the proof of Lemma 3.6).

We claim that in every admissible schedule extending $\sigma_1\sigma_2\sigma_3$, every processor in the sub-ring R_2 must receive at least one additional message before terminating. This holds because a processor in R_2 can learn the identifier of the leader only through messages that arrive from R_1. Since in $\sigma_1\sigma_2\sigma_3$ no message is delivered between R_1 and R_2, such a processor will have to receive another message before it can terminate. This argument depends on the assumption that all processors must learn the id of the leader.

The above argument clearly implies that an additional $\Omega(\frac{n}{2})$ messages must be received on R. However, we cannot conclude our proof here because the above claim assumes that both e_p and e_q are unblocked (becasue the schedule must be admissible), and thus the resulting schedule is not open. We cannot a priori claim that many messages will be received if e_p alone is unblocked, because the algorithm might decide to wait for messages on e_q. However, we can prove that it suffices to unblock only one of e_p or e_q and still force the algorithm to receive $\Omega(\frac{n}{2})$ messages. This is done in the next claim.

Claim 3.8 *There exists a finite schedule segment σ_4 in which $\frac{1}{2}(\frac{n}{2}-1)$ messages are received, such that $\sigma_1\sigma_2\sigma_3\sigma_4$ is an open schedule in which either e_p or e_q is open.*

Proof. Let σ_4'' be such that $\sigma_1\sigma_2\sigma_3\sigma_4''$ is an admissible schedule. Thus all messages are delivered on e_p and e_q and all processors terminate. As we argued above, since each of the processors in R_2 must receive a message before termination, at least $\frac{n}{2}$ messages are received in σ_4'' before A terminates. Let σ_4' be the shortest prefix of σ_4'' in which $\frac{n}{2}-1$ messages are received. Consider all the processors in R that received

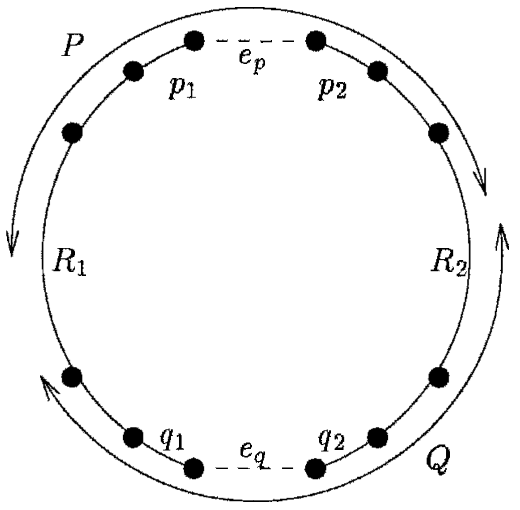

Fig. 3.6 Illustration for Claim 3.8.

messages in σ'_4. Since α'_4 starts in a quiescent configuration in which messages are in transit only on e_p and e_q, these processors form two consecutive sets of processors P and Q. P contains the processors that are awakened because of the unblocking of e_p and thus contains at least one of p_1 and p_2. Similarly, Q contains the processors that are awakened because of the unblocking of e_q and thus contains at least one of q_1 and q_2 (see Fig. 3.6).

Since at most $\frac{n}{2} - 1$ processors are included in these sets and the sets are consecutive, it follows that the two sets are disjoint. Furthermore, the number of messages received by processors in one of the sets is at least $\frac{1}{2}(\frac{n}{2} - 1)$. Without loss of generality, assume this set is P, that is, the one containing p_1 or p_2. Let σ_4 be the subsequence of σ'_4 that contains only the events on processors in P. Since in σ'_4 there is no communication between processors in P and processors in Q, $\sigma_1\sigma_2\sigma_3\sigma_4$ is a schedule. By assumption, at least $\frac{1}{2}(\frac{n}{2} - 1)$ messages are received in σ_4. Furthermore, by construction, no message is delivered on e_q. Thus $\sigma_1\sigma_2\sigma_3\sigma_4$ is the desired open schedule. □

To summarize, we started with two separate schedules on R_1 and R_2, in which $2M(\frac{n}{2})$ messages were received. We then forced the ring into a quiescent configuration. Finally, we forced $\frac{1}{2}(\frac{n}{2} - 1)$ additional messages to be received from the quiescent configuration, while keeping either e_p or e_q open. Thus we have constructed an open schedule in which at least $2M(\frac{n}{2}) + \frac{1}{2}(\frac{n}{2} - 1)$ messages are received.

□

3.4 SYNCHRONOUS RINGS

We now turn to the problem of electing a leader in a synchronous ring. Again, we present both upper and lower bounds. For the upper bound, two leader election algorithms that require $O(n)$ messages are presented. Obviously, the message complexity of these algorithms is optimal. However, the running time is not bounded by any function (solely) of the ring size, and the algorithms use processor identifiers in an unusual way. For the lower bound, we show that any algorithm that is restricted to use only comparisons of identifiers, or is restricted to be time bounded (that is, to terminate in a number of rounds that depends only on the ring size), requires at least $\Omega(n \log n)$ messages.

3.4.1 An $O(n)$ Upper Bound

The proof of the $\Omega(n \log n)$ lower bound for leader election in an asynchronous ring presented in Section 3.3.3, heavily relied on delaying messages for arbitrarily long periods. It is natural to wonder whether better results can be achieved in the synchronous model, where message delay is fixed. As we shall see, in the synchronous model information can be obtained not only by receiving a message but also by *not* receiving a message in a certain round.

In this section, two algorithms for electing a leader in a synchronous ring are presented. Both algorithms require $O(n)$ messages. The algorithms are presented for a unidirectional ring, where communication is in the clockwise direction. Of course, the same algorithms can be used for bidirectional rings. The first algorithm is nonuniform, and requires all processors in the ring to start at the same round, as is provided for in the synchronous model. The second algorithm is uniform, and processors may start in different rounds, that is, the algorithm works in a model that is slightly weaker than the standard synchronous model.

3.4.1.1 The Nonuniform Algorithm The nonuniform algorithm elects the processor with the minimal identifier to be the leader. It works in phases, each consisting of n rounds. In phase i ($i \geq 0$), if there is a processor with identifier i, it is elected as the leader, and the algorithm terminates. Therefore, the processor with the minimal identifier is elected.

In more detail, phase i includes rounds $n \cdot i + 1, n \cdot i + 2, \ldots, n \cdot i + n$. At the beginning of phase i, if a processor's identifier is i, and it has not terminated yet, the processor sends a message around the ring and terminates as a leader. If the processor's identifier is not i and it receives a message in phase i, it forwards the message and terminates the algorithm as a non-leader.

Because identifiers are distinct, it is clear that the unique processor with the minimal identifier terminates as a leader. Moreover, exactly n messages are sent in the algorithm; these messages are sent in the phase in which the winner is found. The number of rounds, however, depends on the minimal identifier in the ring. More precisely, if m is the minimal identifier, then the algorithm takes $n \cdot (m + 1)$ rounds.

Note that the algorithm depends on the requirements mentioned—knowledge of n and synchronized start. The next algorithm overcomes these restrictions.

3.4.1.2 The Uniform Algorithm The next leader election algorithm does not require knowledge of the ring size. In addition, the algorithm works in a slightly weakened version of the standard synchronous model, in which the processors do not necessarily start the algorithm simultaneously. More precisely, a processor either wakes up spontaneously in an arbitrary round or wakes up upon receiving a message from another processor (see Exercise 3.7).

The uniform algorithm uses two new ideas. First, messages that originate at different processors are forwarded at different rates. More precisely, a message that originates at a processor with identifier i is delayed $2^i - 1$ rounds at each processor that receives it, before it is forwarded clockwise to the next processor. Second, to overcome the unsynchronized starts, a preliminary wake-up phase is added. In this phase, each processor that wakes up spontaneously sends a "wake-up" message around the ring; this message is forwarded without delay. A processor that receives a wake-up message before starting the algorithm does not participate in the algorithm and will only act as a *relay*, forwarding or swallowing messages. After the preliminary phase the leader is elected among the set of participating processors.

The wake-up message sent by a processor contains the processor's identifier. This message travels at a regular rate of one edge per round and eliminates all the processors that are not awake when they receive the message. When a message from a processor with identifier i reaches a participating processor, the message starts to travel at a rate of 2^i; to accomplish this slowdown, each processor that receives such a message delays it for $2^i - 1$ rounds before forwarding it. Note that after a message reaches an awake processor, all processors it will reach are awake. A message is in the *first phase* until it is received by a participating processor; after reaching a participating processor, a message is in the *second phase*, and it is forwarded at a rate of 2^i.

Throughout the algorithm, processors forward messages. However, as in previous leader election algorithms we have seen, processors sometimes swallow messages without forwarding them. In this algorithm, messages are swallowed according to the following rules:

1. A participating processor swallows a message if the identifier in the message is larger than the minimal identifier it has seen so far, including its own identifier.

2. A relay processor swallows a message if the identifier in the message is larger than the minimal identifier it has seen so far, not including its own id.

The pseudocode appears in Algorithm 6.

As we prove below, n rounds after the first processor wakes up, only second-phase messages are left, and the leader is elected among the participating processors. The swallowing rules guarantee that only the participating processor with the smallest identifier receives its message back and terminates as a leader. This is proved in Lemma 3.9.

For each i, $0 \leq i < n$, let id_i be the identifier of processor p_i and $\langle id_i \rangle$ be the message originated by p_i.

Lemma 3.9 *Only the processor with the smallest identifier among the participating processors receives its own message back.*

Algorithm 6 Synchronous leader election: code for processor p_i, $0 \leq i < n$.

```
Initially waiting is empty and status is asleep

1:  let $R$ be the set of messages received in this computation event
2:  $S := \emptyset$                                   // the messages to be sent

3:  if status = asleep then
4:      if $R$ is empty then                          // woke up spontaneously
5:          status := participating
6:          min := id
7:          add $\langle id, 1 \rangle$ to $S$                         // first phase message
8:      else
9:          status := relay
10:         min := $\infty$

9:  for each $\langle m,h \rangle$ in $R$ do
10:     if $m <$ min then
11:         become not elected
12:         min := $m$
13:         if (status = relay) and ($h = 1$) then           // $m$ stays first phase
14:             add $\langle m,h \rangle$ to $S$
15:         else                                   // $m$ is/becomes second phase
16:             add $\langle m,2 \rangle$ to waiting tagged with current round number
17:     elseif $m =$ id then become elected
                                        // if $m >$ min then message is swallowed

18: for each $\langle m,2 \rangle$ in waiting do
19:     if $\langle m,2 \rangle$ was received $2^m - 1$ rounds ago then
20:         remove $\langle m \rangle$ from waiting and add to $S$

21: send $S$ to left
```

Proof. Let p_i be the participating processor with the smallest identifier. (Note that at least one processor must participate in the algorithm.) Clearly, no processor, participating or not, can swallow $\langle id_i \rangle$.

Furthermore, since $\langle id_i \rangle$ is delayed at most 2^{id_i} rounds at each processor, p_i eventually receives its message back.

Assume, by way of contradiction, that some other processor p_j, $j \neq i$, also receives back its message $\langle id_j \rangle$. Thus, $\langle id_j \rangle$ must pass through all the processors in the ring, including p_i. But $id_i < id_j$, and since p_i is a participating processor, it does not forward $\langle id_j \rangle$, a contradiction. □

The above lemma implies that exactly one processor receives its message back. Thus this processor will be the only one to declare itself a leader, implying the correctness of the algorithm. We now analyze the number of messages sent during an admissible execution of the algorithm.

To calculate the number of messages sent during an admissible execution of the algorithm we divide them into three categories:

1. First-phase messages
2. Second-phase messages sent before the message of the eventual leader enters its second phase
3. Second-phase messages sent after the message of the eventual leader enters its second phase

Lemma 3.10 *The total number of messages in the first category is at most n.*

Proof. We show that at most one first-phase message is forwarded by each processor, which implies the lemma.

Assume, by way of contradiction, that some processor p_i forwards two messages in their first phase, $\langle id_j \rangle$ from p_j and $\langle id_k \rangle$ from p_k. Assume, without loss of generality, that p_j is closer to p_i than p_k is to p_i, in terms of clockwise distance. Thus, $\langle id_k \rangle$ must pass through p_j before it arrives at p_i. If $\langle id_k \rangle$ arrives at p_j after p_j woke up and sent $\langle id_j \rangle$, $\langle id_k \rangle$ continues as a second-phase message, at a rate of 2^{id_k}; otherwise, p_j does not participate and $\langle id_j \rangle$ is not sent. Thus either $\langle id_k \rangle$ arrives at p_i as a second phase message or $\langle id_j \rangle$ is not sent, a contradiction. □

Let r be the first round in which some processor starts executing the algorithm, and let p_i be one of these processors. To bound the number of messages in the second category, we first show that n rounds after the first processor starts executing the algorithm, all messages are in their second phase.

Lemma 3.11 *If p_j is at (clockwise) distance k from p_i, then a first-phase message is received by p_j no later than round $r + k$.*

Proof. The proof is by induction on k. The base case, $k = 1$, is obvious because p_i's neighbor receives p_i's message in round $r + 1$. For the inductive step, assume that the processor at (clockwise) distance $k - 1$ from p_i receives a first-phase message no later than round $r + k - 1$. If this processor is already awake when it receives the first-phase message, it has already sent a first-phase message to its neighbor p_j; otherwise, it forwards the first-phase message to p_j in round $r + k$. □

Lemma 3.12 *The total number of messages in the second category is at most n.*

Proof. As shown in the proof of Lemma 3.10, at most one first-phase message is sent on each edge. Since by round $r + n$ one first-phase message was sent on every edge, it follows that after round $r + n$ no first-phase messages are sent. By Lemma 3.11, the message of the eventual leader enters its second phase at most n rounds after the first message of the algorithm is sent. Thus messages from the second category are sent only in the n rounds following the round in which the first processor woke up.

Message $\langle i \rangle$ in its second phase is delayed $2^i - 1$ rounds before being forwarded. Thus $\langle i \rangle$ is sent at most $\frac{n}{2^i}$ times in this category. Since messages containing smaller identifiers are forwarded more often, the maximum number of messages is obtained when all the processors participate, and when the identifiers are as small as possible, that is, $0, 1, \ldots, n-1$. Note that second-phase messages of the eventual leader (in our case, 0) are not counted in this category. Thus the number of messages in the second category is at most $\sum_{i=1}^{n-1} \frac{n}{2^i} \leq n$. □

Let p_i be the processor with the minimal identifier; no processor forwards a message after it forwards $\langle id_i \rangle$. Once $\langle id_i \rangle$ returns to p_i, all the processors in the ring have already forwarded it, and therefore we have the following lemma:

Lemma 3.13 *No message is forwarded after $\langle id_i \rangle$ returns to p_i.*

Lemma 3.14 *The total number of messages in the third category is at most $2n$.*

Proof. Let p_i be the eventual leader, and let p_j be some other participating processor. By Lemma 3.9, $id_i < id_j$. By Lemma 3.13, there are no messages in the ring after p_i receives its message back. Since $\langle id_i \rangle$ is delayed at most 2^{id_i} rounds at each processor, at most $n \cdot 2^{id_i}$ rounds are needed for $\langle id_i \rangle$ to return to p_i. Therefore, messages in the third category are sent only during $n \cdot 2^{id_i}$ rounds. During these rounds, $\langle id_j \rangle$ is forwarded at most

$$\frac{1}{2^{id_j}} \cdot n \cdot 2^{id_i} = n \cdot 2^{id_i - id_j}$$

times. Hence, the total number of messages transmitted in this category is at most

$$\sum_{j=0}^{n-1} \frac{n}{2^{id_j - id_i}}$$

By the same argument as in the proof of Lemma 3.12, this is less than or equal to

$$\sum_{k=0}^{n-1} \frac{n}{2^k} \leq 2n$$

□

Lemmas 3.10, 3.12, and 3.14 imply:

Theorem 3.15 *There is a synchronous leader election algorithm whose message complexity is at most $4n$.*

Now consider the time complexity of the algorithm. By Lemma 3.13, the computation ends when the elected leader receives its message back. This happens within $O(n2^i)$ rounds since the first processor starts executing the algorithm, where i is the identifier of the elected leader.

3.4.2 An $\Omega(n \log n)$ Lower Bound for Restricted Algorithms

In Section 3.4.1, we presented two algorithms for electing a leader in synchronous rings whose worst-case message complexity is $O(n)$. Both algorithms have two undesirable properties. First, they use the identifiers in a nonstandard manner (to decide how long a message should be delayed). Second, and more importantly, the number of rounds in each admissible execution depends on the identifiers of processors. The reason this is undesirable is that the identifiers of the processors can be huge relative to n.

In this section, we show that both of these properties are inherent for any message efficient algorithm. Specifically, we show that if an algorithm uses the identifiers only for comparisons it requires $\Omega(n \log n)$ messages. Then we show, by reduction, that if an algorithm is restricted to use a bounded number of rounds, independent of the identifiers, then it also requires $\Omega(n \log n)$ messages.

The synchronous lower bounds cannot be derived from the asynchronous lower bound (of Theorem 3.5), because the algorithms presented in Section 3.4.1 indicate that additional assumptions are necessary for the synchronous lower bound to hold. The synchronous lower bound holds even for nonuniform algorithms, whereas the asynchronous lower bound holds only for uniform algorithms. Interestingly, the converse derivation, of the asynchronous result from the synchronous, is correct and provides an asynchronous lower bound for nonuniform algorithms, as explored in Exercise 3.11.

3.4.2.1 Comparison-Based Algorithms In this section, we formally define the concept of comparison-based algorithms.

For the purpose of the lower bound, we assume that all processors begin executing at the same round.

Recall that a ring is specified by listing the processors' identifiers in clockwise order, beginning with the smallest identifier. Note that in the synchronous model an admissible execution of the algorithm is completely defined by the initial configuration, because there is no choice of message delay or relative order of processor steps. The initial configuration of the system, in turn, is completely defined by the ring, that is, by the listing of processors' identifiers according to the above rule. When the choice of algorithm is clear from context, we will denote the admissible execution determined by ring R as *exec*(R).

Two processors, p_i in ring R_1 and p_j in ring R_2, are *matching* if they both have the same position in the respective ring specification. Note that matching processors are at the same distance from the processor with the smallest identifier in the respective rings.

Intuitively, an algorithm is comparison based if it behaves the same on rings that have the same order pattern of the identifiers. Formally, two rings, $x_0, \ldots, x_{n-1}$ and $y_0, \ldots, y_{n-1}$, are *order equivalent* if for every i and j, $x_i < x_j$ if and only if $y_i < y_j$. Recall that the *k-neighborhood* of a processor p_i in a ring is the sequence of $2k + 1$ identifiers of processors $p_{i-k}, \ldots, p_{i-1}, p_i, p_{i+1}, \ldots, p_{i+k}$ (all indices are calculated modulo n). We extend the notion of order equivalence to k-neighborhoods in the obvious manner.

We now define what it means to "behave the same." Intuitively, we would like to claim that in the admissible executions on order equivalent rings R_1 and R_2, the same messages are sent and the same decisions are made. In general, however, messages sent by the algorithm contain identifiers of processors; thus messages sent on R_1 will be different from messages sent on R_2. For our purpose, however, we concentrate on the message pattern, that is, when and where messages are sent, rather than their content, and on the decisions. Specifically, consider two executions α_1 and α_2 and two processors p_i and p_j. We say that the behavior of p_i in α_1 is *similar* in round k to the behavior of p_j in α_2 if the following conditions are satisfied:

1. p_i sends a message to its left (right) neighbor in round k in α_1 if and only if p_j sends a message to its left (right) neighbor in round k in α_2

2. p_i terminates as a leader in round k of α_1 if and only if p_j terminates as a leader in round k of α_2

We say that that the behaviors of p_i in α_1 and p_j in α_2 are *similar* if they are similar in all rounds $k \geq 0$. We can now formally define comparison-based algorithms.

Definition 3.2 *An algorithm is* comparison based *if for every pair of order-equivalent rings R_1 and R_2, every pair of matching processors have similar behaviors in exec(R_1) and exec(R_2).*

3.4.2.2 Lower Bound for Comparison-Based Algorithms Let A be a comparison-based leader election algorithm. The proof considers a ring that is highly symmetric in its order patterns, that is, a ring in which there are many order-equivalent neighborhoods. Intuitively, as long as two processors have order-equivalent neighborhoods they behave the same under A. We derive the lower bound by executing A on a highly symmetric ring and arguing that if a processor sends a message in a certain round, then all processors with order-equivalent neighborhoods also send a message in that round.

A crucial point in the proof is to distinguish rounds in which information is obtained by processors from rounds in which no information is obtained. Recall that in a synchronous ring it is possible for a processor to obtain information even without receiving a message. For example, in the nonuniform algorithm of Section 3.4.1, the fact that no message is received in rounds 1 through n implies that no processor in the ring has the identifier 0. The key to the proof that follows is the observation that the nonexistence of a message in a certain round r is useful to processor p_i only if a message could have been received in this round in a different, but order-equivalent, ring. For example, in the nonuniform algorithm, if some processor in the ring had the identifier 0, a message would have been received in rounds $1, \ldots, n$. Thus a round in which no message is sent in any order-equivalent ring is not useful. Such useful rounds are called active, as defined below:

Definition 3.3 *A round r is* active *in an execution on a ring R if some processor sends a message in round r of the execution. When R is understood from context, we denote by r_k the index of the kth active round.*[1]

Recall that, by definition, a comparison-based algorithm generates similar behaviors on order-equivalent rings. This implies that, for order equivalent rings R_1 and R_2, a round is active in $exec(R_1)$ if and only if it is active in $exec(R_2)$.

Because information in messages can travel only k processors around the ring in k rounds, the state of a processor after round k depends only on its k-neighborhood. We have, however, a stronger property that the state of a processor after the kth *active* round depends only on its k-neighborhood. This captures the above intuition that information is obtained only in active rounds and is formally proved in Lemma 3.16. Note that the lemma does not require that the processors be matching (otherwise the claim follows immediately from the definition) but does require that their neighborhoods be identical. This lemma requires the hypothesis that the two rings be order equivalent. The reason is to ensure that the two executions under consideration have the same set of active rounds; thus r_k is well-defined.

Lemma 3.16 *Let R_1 and R_2 be order-equivalent rings, and let p_i in R_1 and p_j in R_2 be two processors with identical k-neighborhoods. Then the sequence of transitions that p_i experiences in rounds 1 through r_k of $exec(R_1)$ is the same as the sequence of transitions that p_j experiences in rounds 1 through r_k of $exec(R_2)$.*

Proof. Informally, the proof shows that after k active rounds, a processor may learn only about processors that are at most k away from itself.

The formal proof follows by induction on k. For the base case $k = 0$, note that two processors with identical 0-neighborhoods have the same identifiers, and thus they are in the same state.

For the inductive step, assume that every two processors with identical $(k-1)$-neighborhoods are in the same state after the $(k-1)$-st active round. Since p_i and p_j have identical k-neighborhoods, they also have identical $(k-1)$-neighborhoods; therefore, by the inductive hypothesis, p_i and p_j are in the same state after the $(k-1)$-st active round. Furthermore, their respective neighbors have identical $(k-1)$-neighborhoods. Therefore, by the inductive hypothesis, their respective neighbors are in the same state after the $(k-1)$-st active round.

In the rounds between the $(k-1)$-st active round and the kth active round (if there are any), no processor receives any message and thus p_i and p_j remain in the same state as each other, and so do their respective neighbors. (Note that p_i might change its state during the nonactive rounds, but since p_j has the same transition function, it makes the same state transition.) In the kth active round, if both p_i and p_j do not receive messages they are in the same states at the end of the round. If p_i receives a message from its right neighbor, p_j also receives an identical message from its

[1]Recall that once the ring is fixed, the whole admissible execution is determined because the system is synchronous.

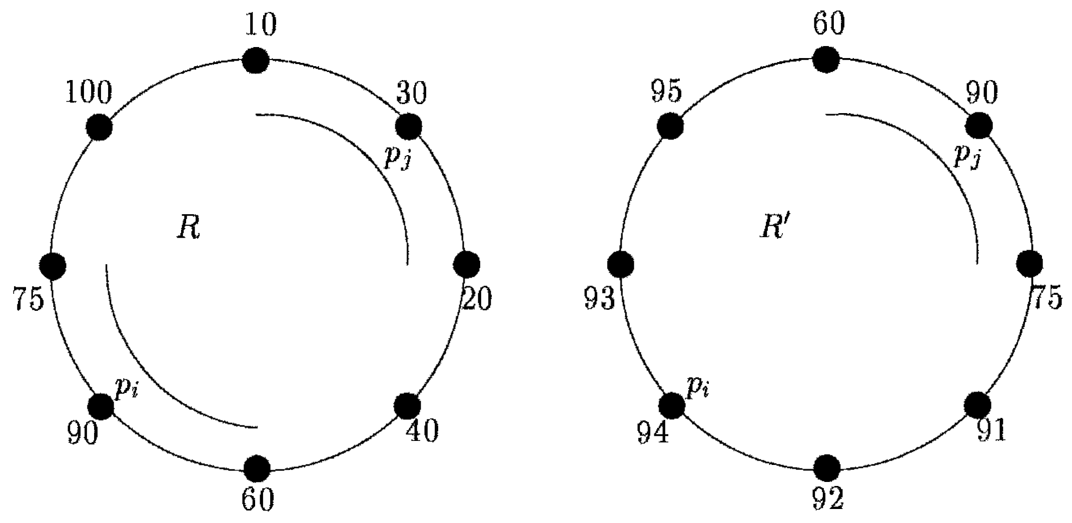

Fig. 3.7 Example for the proof of Lemma 3.17; $k = 1$ and $n = 8$.

right neighbor, because the neighbors are in the same state, and similarly for the left neighbor. Hence, p_i and p_j are in the same state at the end of the kth active round, as needed. □

Lemma 3.17 extends the above claim from processors with identical k-neighborhoods to processors with order-equivalent k-neighborhoods. It relies on the fact that A is comparison based. Furthermore, it requires the ring R to be spaced, which intuitively means that for every two identifiers in R, there are n unused identifiers between them, where n is the size of the ring. Formally, a ring of size n is *spaced* if for every identifier x in the ring, the identifiers $x - 1$ through $x - n$ are not in the ring.

Lemma 3.17 *Let R be a spaced ring and let p_i and p_j be two processors with order-equivalent k-neighborhoods in R. Then p_i and p_j have similar behaviors in rounds 1 through r_k of exec(R).*

Proof. We construct another ring R' that satisfies the following:

- p_j's k-neighborhood is the same as p_i's k-neighborhood from R
- the identifiers in R' are unique
- R' is order equivalent to R with p_j in R' matching p_j in R

R' can be constructed because R is spaced (see an example in Fig. 3.7).

By Lemma 3.16, the sequence of transitions that p_i experiences in rounds 1 through r_k of *exec*(R) is the same as the sequence of transitions that p_j experiences in rounds 1 through r_k of *exec*(R'). Thus p_i's behavior in rounds 1 through r_k of *exec*(R) is similar to p_j's behavior in rounds 1 through r_k of *exec*(R'). Since the algorithm is comparison based and p_j in R' is matching to p_j in R, p_j's behavior in rounds 1 through r_k of *exec*(R') is similar to p_j's behavior in rounds 1 through r_k of *exec*(R).

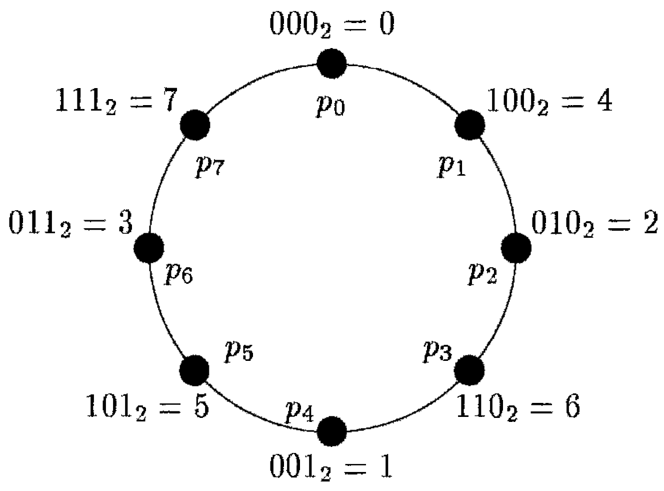

Fig. 3.8 The ring R_8^{rev}.

Thus p_i's behavior and p_j's behavior in rounds 1 through r_k of *exec*(R) are similar. □

We can now prove the main theorem:

Theorem 3.18 *For every $n \geq 8$ that is a power of 2, there exists a ring S_n of size n such that for every synchronous comparison-based leader election algorithm A, $\Omega(n \log n)$ messages are sent in the admissible execution of A on S_n.*

Proof. Fix any such algorithm A. The key to this proof is the construction of S_n, a highly symmetric ring, in which many processors have many order equivalent neighborhoods. S_n is constructed in two steps.

First, define the n-processor ring R_n^{rev} as follows. For each i, $0 \leq i < n$, let p_i's identifier be rev(i), where rev(i) is the integer whose binary representation using $\log n$ bits is the reverse of the binary representation of i. (See the special case $n = 8$ in Fig. 3.8.) Consider any partitioning of R_n^{rev} into consecutive segments of length j, where j is a power of 2. It can be shown that all these segments are order equivalent (see Exercise 3.9).

S_n is a spaced version of R_n^{rev}, obtained by multiplying each identifier in R_n^{rev} by $n + 1$ and then adding n to it. These changes do not alter the order equivalence of segments.

Lemma 3.19 quantifies how many order-equivalent neighborhoods of a given size there are in S_n. This result is then used to show, in Lemma 3.20, a lower bound on the number of active rounds of the algorithms and to show, in Lemma 3.21, a lower bound on the number of messages sent in each active round. The desired bound of $\Omega(n \log n)$ is obtained by combining the latter two bounds.

Lemma 3.19 *For all $k < n/8$, and for all k-neighborhoods N of S_n, there are more than $\frac{n}{2(2k+1)}$ k-neighborhoods of S_n that are order equivalent to N (including N itself).*

Proof. N consists of a sequence of $2k+1$ identifiers. Let j be the smallest power of 2 that is bigger than $2k+1$. Partition S_n into $\frac{n}{j}$ consecutive segments such that one segment totally encompasses N. By the construction of S_n, all of these segments are order equivalent. Thus there are at least $\frac{n}{j}$ neighborhoods that are order equivalent to N. Since $j < 2(2k+1)$, the number of neighborhoods order equivalent to N is more than $\frac{n}{2(2k+1)}$. □

Lemma 3.20 *The number of active rounds in exec(S_n) is at least $n/8$.*

Proof. Let T be the number of active rounds. Suppose in contradiction $T < n/8$. Let p_i be the processor that is elected the leader in *exec*(S_n). By Lemma 3.19, there are more than $\frac{n}{2(2T+1)}$ T-neighborhoods that are order equivalent to p_i's T-neighborhood. By assumption on T,

$$\frac{n}{2(2T+1)} > \frac{n}{2(2n/8+1)} = \frac{2n}{n+4}$$

and, since $n \geq 8$, $\frac{2n}{n+4} > 1$. Thus there exists some processor p_j other than p_i whose T-neighborhood is order equivalent to p_i's T-neighborhood. By Lemma 3.17, p_j is also elected, contradicting the assumed correctness of A. □

Lemma 3.21 *At least $\frac{n}{2(2k+1)}$ messages are sent in the kth active round of exec(S_n), for each k, $1 \leq k \leq n/8$.*

Proof. Consider the kth active round. Since it is active, at least one processor sends a message, say p_i. By Lemma 3.19, there are more than $\frac{n}{2(2k+1)}$ processors whose k-neighborhoods are order equivalent to p_i's k-neighborhood. By Lemma 3.17, each of them also sends a message in the kth active round. □

We now finish the proof of the main theorem. By Lemma 3.20 and Lemma 3.21, the total number of messages sent in *exec*(S_n) is at least

$$\sum_{k=1}^{n/8} \frac{n}{2(2k+1)} \geq \frac{n}{6}\sum_{k=1}^{n/8}\frac{1}{k} > \frac{n}{6}\ln\frac{n}{8}$$

which is $\Omega(n \log n)$. □

Note that in order for this theorem to hold, the algorithm need not be comparison based for *every* set of identifiers drawn from the natural numbers, but only for identifiers drawn from the set $\{0, 1, \ldots, n^2+2n-1\}$. The reason is that the largest identifier in S_n is $n^2+n-1 = (n+1)\cdot \text{rev}(n-1)+n$ (recall that n is a power of 2 and thus the binary representation of $n-1$ is a sequence of 1s). We require the algorithm to be comparison based on *all* identifiers between 0 and n^2+2n-1, and not just on identifiers that occur in S_n, because the proof of Lemma 3.17 uses the fact that the algorithm is comparison based on all identifiers that range from n less than the smallest in S_n to n greater than the largest in S_n.

3.4.2.3 Lower Bound for Time-Bounded Algorithms

The next definition disallows the running time of an algorithm from depending on the identifiers: It requires the running time for each ring size to be bounded, even though the possible identifiers are not bounded, because they come from the set of natural numbers.

Definition 3.4 *A synchronous algorithm A is* time-bounded *if, for each n, the worst-case running time of A over all rings of size n, with identifiers drawn from the natural numbers, is bounded.*

We now prove the lower bound for time-bounded algorithms, by reduction to comparison-based algorithms. We first show how to map from time-bounded algorithms to comparison-based algorithms. Then we use the lower bound of $\Omega(n \log n)$ messages for comparison-based algorithms to obtain a lower bound on the number of messages sent by time-bounded algorithms. Because the comparison-based lower bound as stated is only for values of n that are powers of 2, the same is true here, although the lower bound holds for all values of n (see chapter notes).

To map from time-bounded to comparison-based algorithms, we require definitions describing the behavior of an algorithm during a bounded amount of time.

Definition 3.5 *A synchronous algorithm A is* t-comparison based over identifier set S for ring size n *if, for every two order equivalent rings, R_1 and R_2, of size n, every pair of matching processors have similar behaviors in rounds 1 through t of exec(R_1) and exec(R_2).*

Intuitively, an r-comparison based algorithm over S is an algorithm that behaves as a comparison-based algorithm in the first r rounds, as long as identifiers are chosen from S. If the algorithm terminates within r rounds, then this is the same as being comparison based over S for all rounds.

The first step is to show that every time-bounded algorithm behaves as a comparison-based algorithm over a subset of its inputs, provided that the input set is sufficiently large. To do this we use the finite version of Ramsey's theorem. Informally, the theorem states that if we take a large set of elements and we color each subset of size k with one of t colors, then we can find some subset of size ℓ such that all its subsets of size k have the same color. If we think of the coloring as partitioning into equivalence classes (two subsets of size k belong to the same equivalence class if they have the same color), the theorem says that there is a set of size ℓ such that all its subsets of size k are in the same equivalence class. Later, we shall color rings with the same color if the behavior of matching processors is similar in them.

For completeness, we repeat Ramsey's theorem:

Ramsey's Theorem (finite version) *For all integers k, ℓ, and t, there exists an integer $f(k,\ell,t)$ such that for every set S of size at least $f(k,\ell,t)$, and every t-coloring of the k-subsets of S, some ℓ-subset of S has all its k-subsets with the same color.*

In Lemma 3.22, we use Ramsey's theorem to map any time-bounded algorithm to a comparison-based algorithm.

Lemma 3.22 *Let A be a synchronous time-bounded algorithm with running time $r(n)$. Then, for every n, there exists a set C_n of $n^2 + 2n$ identifiers such that A is $r(n)$-comparison based over C_n for ring size n.*

Proof. Fix n. Let Y and Z be any two n-subsets of N (the natural numbers). We say that Y and Z are *equivalent subsets* if, for every pair of order equivalent rings, R_1 with identifiers from Y and R_2 with identifiers from Z, matching processors have similar behaviors in rounds 1 through $t(n)$ of *exec*(R_1) and *exec*(R_2). This definition partitions the n-subsets of N into finitely many equivalence classes, since the term 'similar behavior' only refers to the presence or absence of messages and terminated states. We color the n-subsets of N such that two n-subsets have the same color if and only if they are in the same equivalence class.

By Ramsey's theorem, if we take t to be the number of equivalence classes (colors), ℓ to be $n^2 + 2n$, and k to be n, then, since N is infinite, there exists a subset C_n of N of cardinality $n^2 + 2n$ such that all n-subsets of C_n belong to the same equivalence class.

We claim that A is an $r(n)$-comparison based algorithm over C_n for ring size n. Consider two order-equivalent rings, R_1 and R_2, of size n with identifiers from C_n. Let Y be the set of identifiers in R_1 and Z be the set of identifiers in R_2. Y and Z are n-subsets of C_n; therefore, they belong to the same equivalence class. Thus matching processors have similar behaviors in rounds 1 through $r(n)$ of *exec*(R_1) and *exec*(R_2). Therefore, A is an $r(n)$-comparison based algorithm over C_n for ring size n. $\square$

Theorem 3.18 implies that every comparison-based algorithm has worst-case message complexity $\Omega(n \log n)$. We cannot immediately apply this theorem now, because we have only shown that a time-bounded algorithm A is comparison based on a specific set of ids, not on all ids. However, we will use A to design another algorithm A', with the same message complexity as A, that is comparison based on rings of size n with ids from the set $\{0, 1, \ldots, n^2 + 2n - 1\}$. As was discussed just after the proof of Theorem 3.18, this will be sufficient to show that the message complexity of A' is $\Omega(n \log n)$. Thus the message complexity of A is $\Omega(n \log n)$.

Theorem 3.23 *For every synchronous time-bounded leader election algorithm A and every $n \geq 8$ that is a power of 2, there exists a ring R of size n such that $\Omega(n \log n)$ messages are sent in the admissible execution of A on R.*

Proof. Fix an algorithm A satisfying the hypotheses of the theorem with running time $r(n)$. Fix n; let C_n be the set of identifiers guaranteed by Lemma 3.22, and let $c_0, c_1, \ldots, c_{n^2+2n-1}$ be the elements of C_n in increasing order.

We define an algorithm A' that is comparison based on rings of size n with identifiers from the set $\{0, 1, \ldots, n^2+2n-1\}$ and that has the same time and message complexity as A. In algorithm A', a processor with identifier i executes algorithm A as if though had the identifier c_i. Since A is $r(n)$-comparison based over C_n for ring size n and since A terminates within $r(n)$ rounds, it follows that A' is comparison based on rings of size n with identifiers from the set $\{0, 1, \ldots, n^2 + 2n - 1\}$.

By Theorem 3.18, there is a ring of size n with identifiers from $\{0, 1, \ldots, n^2 + 2n - 1\}$ in which A' sends $\Omega(n \log n)$ messages. By the way A' was constructed, there is an execution of A in a ring of size n with identifiers from C_n in which the same messages are sent, which proves the theorem. □

Exercises

3.1 Prove that there is no anonymous leader election algorithm for asynchronous ring systems.

3.2 Prove that there is no anonymous leader election algorithm for synchronous ring systems that is uniform.

3.3 Is leader election possible in a synchronous ring in which all but one processor have the same identifier? Either give an algorithm or prove an impossibility result.

3.4 Consider the following algorithm for leader election in an asynchronous ring: Each processor sends its identifier to its right neighbor; every processor forwards a message (to its right neighbor) only if it includes an identifier larger than its own.

Prove that the average number of messages sent by this algorithm is $O(n \log n)$, assuming that identifiers are uniformly distributed integers.

3.5 In Section 3.3.3, we have seen a lower bound of $\Omega(n \log n)$ on the number of messages required for electing a leader in an asynchronous ring. The proof of the lower bound relies on two additional properties: (a) the processor with the maximal identifier is elected, and (b) all processors must know the identifier of the elected leader.

Prove that the lower bound holds also when these two requirements are omitted.

3.6 Extend Theorem 3.5 to the case in which n is not an integral power of 2.

Hint: Consider the largest $n' < n$ that is an integral power of 2, and prove the theorem for n'.

3.7 Modify the formal model of synchronous message passing systems to describe the non-synchronized start model of Section 3.4.1. That is, state the conditions that executions and admissible executions must satisfy.

3.8 Prove that the order-equivalent ring R' in proof of Lemma 3.17 can always be constructed.

3.9 Recall the ring R_n^{rev} from the proof of Theorem 3.18. For every partition of R_n^{rev} into $\frac{n}{j}$ consecutive segments, where j is a power of 2, prove that all of these segments are order equivalent.