Distributed Computing: Fundamentals, Simulations and Advanced Topics, Second Edition
Hagit Attiya and Jennifer Welch
 ISBN: 0-471-45324-2

10

Fault-Tolerant Simulations of Read/Write Objects

In preceding chapters, we have seen several types of shared objects, differing both in their sharing patterns (i.e., how many processors can access an object) and in their semantics (i.e., what type of operations can be applied to the object). A natural question concerns the relative power of these objects, that is, do different types of shared objects admit solutions to different problems?

One way to compare the power of shared object types is to simulate a shared object of one type, the high-level type, using shared objects of another type, the low-level type. Such a simulation implies that any algorithm that uses objects of the high-level type will also work using objects of the low-level type, allowing one to design algorithms assuming the more convenient high-level objects, but to run it in a system that provides only the low-level objects, which might be a more realistic assumption. The existence of such a simulation indicates that, at least theoretically, the low-level type allows solutions to the same problems that the high-level type does.

A traditional method of simulating shared objects using low-level objects is to use critical sections (cf. Chapter 4). In this method, access to the low-level objects in the simulation of the high-level object is guarded by a critical section and the objects are updated in mutual exclusion. Although simple, this solution is very sensitive to processor failures and slowdowns. A failed or slow processor that is inside the critical section can block or delay the progress of all processors and prohibit them from accessing the simulated shared object. Therefore, we are interested in simulations that are *wait-free*, that is, such that each processor can complete an access to the high-level object using a finite number of accesses to the low-level objects, without depending on other processors. (A more precise definition appears below.)

This chapter is the first of two chapters that discuss the relationships between various types of shared objects. In this chapter, we restrict our attention to a few kinds of objects that can be wait-free simulated from read/write registers. Chapter 15 addresses arbitrary data types.

Throughout this chapter we are concerned solely with linearizable objects.

10.1 FAULT-TOLERANT SHARED MEMORY SIMULATIONS

In this chapter, we study how to simulate shared memory systems both on top of other kinds of shared memory and on top of an asynchronous message-passing system, in the presence of crash failures. We consider two ways to formalize such simulations. In the first, we define a failure-prone version of linearizable shared memory and require the existence of a global simulation as defined in Chapter 7. In the second, we place an additional constraint on the definition of the simulation but keep the original (failure free) definition of linearizable shared memory.

The definition of an (asynchronous) shared memory system that is subject to crash failures differs from the failure-free definition in Chapter 9 (Section 9.1) in two ways. First, the liveness condition is weakened to require responses only for invocations by nonfaulty processors. Second, the linearizability condition must be modified. Consider the situation in which a processor has done all the work to implement a high-level operation but fails just before it completes the operation. Subsequent reads will observe the result of the operation, even though it is not complete. Thus the definition of linearizability must require that there is a permutation of all the completed operations *and some subset of the pending operations* satisfying the same properties.

Here is the definition of an *f-resilient shared memory system.* Inputs are invocations on the shared object, and outputs are responses from the shared object. For a sequence of events σ to be in the allowable set, there must be a partitioning of the processor indices into "faulty" and "nonfaulty" such that there are at most f faulty processors and the following properties are satisfied:

Correct Interaction: For each processor p_i, $\sigma|i$ consists of alternating invocations and matching responses, beginning with an invocation. This condition imposes constraints on the inputs.

Nonfaulty Liveness: Every invocation by a nonfaulty processor has a matching response.

Extended Linearizability: There exists a permutation π of all the completed operations in σ and some subset of the pending operations such that

1. For each object O, $\pi|O$ is legal, that is, it is in the sequential specification of O; and
2. If the response of operation o_1 occurs in σ before the invocation of operation o_2, then o_1 appears before o_2 in π

We will be studying whether communication system $\mathcal{C}_1$ can simulate communication system $\mathcal{C}_2$, where $\mathcal{C}_2$ is a shared memory system consisting of certain types of objects and subject to f processor crashes and $\mathcal{C}_1$ is a communication system (either shared memory or message passing) that is also subject to f processor crashes.

Throughout most of this chapter, we will be studying the wait-free case, that is, when $f = n - 1$. For this case, there is an alternative definition of a simulation of an f-resilient shared memory system. (Exercise 10.2 asks you to show that the two definitions are equivalent.)

Intuitively, we want the simulation not to delay faster processors on account of slower ones. Thus we define a *wait-free simulation* to be a (global) simulation of one (failure free) shared memory system by another (failure free) shared memory system with the following additional property:

- Let α be any admissible execution of the simulation program. Let α' be any finite prefix of α in which some processor p_i has a pending high-level operation, that is, there is an invocation from the environment at p_i without a matching response. Then there must exist an extension of α' consisting solely of events by p_i in which the pending operation is completed.

Another way of stating the wait-free property is that every operation is able to complete on its own, from any point, without assistance from other processors.

Usually we will use the wait-free simulation definition when $f = n - 1$.

By the definition of linearizability, there exists a single point, called the *linearization point*, somewhere between the invocation and response of the operation at which the operation appears to take effect. One strategy for showing that an execution is linearizable is to explicitly assign a candidate for the linearization point of every simulated operation, somewhere between its invocation and response. Clearly, the order implied by the sequence of linearization points preserves the order of non-overlapping operations. If we show, in addition, that every read operation returns the value of the write operation with the latest linearization point that precedes the linearization point of the read operation, then we have shown the desired linearization of the execution. Linearization points correspond to our intuition that each operation "appears" to execute atomically at some point during its execution interval.

10.2 SIMPLE READ/WRITE REGISTER SIMULATIONS

In Chapter 4, we mentioned different types of read/write registers, depending on the manner in which processors access them: single-reader or multi-reader, single-writer or multi-writer. Registers can be further classified depending on the number of values that can be written to them; they may be *binary*, with only two possible values, or *multi-valued*, with any finite number of possible values.

In this section, we show that registers that may seem more complicated, namely, multi-writer multi-reader multi-valued registers, have a wait-free simulation using simpler registers, that is, single-writer single-reader binary registers. This simulation is incremental, going through several stages in which the versatility of accesses

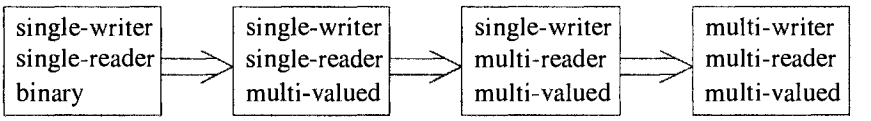

Fig. 10.1 Chain of constructions presented in this chapter.

increases. Figure 10.1 depicts the stages of this simulation. In this figure, an arrow stands for a simulation of a register type from a simpler (more restricted) register type. Actually, the binary to multi-valued construction requires that the multi-valued register being simulated take on only a bounded number of different values. In contrast, the other constructions depend on using building block registers that can take on unbounded values. However, it is possible to transform these unbounded value simulation algorithms into algorithms that only use bounded values (see the chapter notes).

10.2.1 Multi-Valued from Binary

The basic object we consider is a single-writer, single-reader binary register. For each register, there is a single processor (the writer) that can write to it and a single processor (the reader) that can read from it. Only binary values can be written to a register of this type, that is, its value is either 0 or 1.

We describe how to simulate a K-valued single-writer single-reader register from a binary single-writer single-reader register for $K > 2$. For simplicity, we talk about a single register R, and two well-defined processors, a (single) reader and a (single) writer. Such simulations can be combined to simulate multiple registers; see Exercise 9.2.

We consider a simple approach in which values are represented in unary; that is, to simulate the K-valued single-writer single-reader register R, we use an array of K binary single-writer single-reader registers, $B[0 \ldots K-1]$. The value i is represented by a 1 in the ith entry and 0 in all other entries. Thus the possible values of R are $\{0, 1, \ldots, K-1\}$.

When read and write operations do not overlap, it is simple to perform operations: A read operation scans the array beginning with index 0 until it finds a 1 in some entry and returns the index of this entry. A write operation writes the value v, by writing the value 1 in the entry whose index is v and clearing (setting to 0) the entry corresponding to the previous value, if different from v.

This simple algorithm is correct if we are guaranteed that read and write operations do not overlap, but it returns incorrect responses when read and write operations may overlap. Consider, for example, the scenario depicted in Figure 10.2 in which R initially holds the value 3, and thus $B[3] = 1$ while $B[2] = B[1] = B[0] = 0$. In response to a request to read the multi-valued register, the reader reads $B[0]$ and observes 0, then reads $B[1]$ and observes 0, and then reads $B[2]$. While the reader

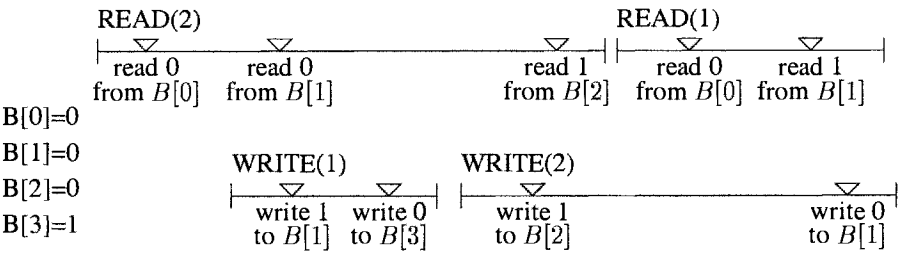

Fig. 10.2 A counterexample to simple multi-valued algorithm; the triangular markers indicate the linearization points of the low-level operations.

is waiting to get the response from $B[2]$, the writer performs a high-level write of 1, during which it sets $B[1]$ to 1 and clears $B[3]$. Then the writer begins a high-level write of 2 and begins writing 1 to $B[2]$. Finally the reader gets the response from $B[2]$, with value 1. Now the reader returns 2 as the value of the multi-valued register. Subsequently, the reader begins a second read on the multi-valued register; it reads $B[0]$ and observes 0, then reads $B[1]$ and observes 1. Thus the reader returns 1 as the value of the multi-valued register. Finally the writer receives the ack for its write to $B[2]$, and clears $B[1]$.[1]

That is, the operations by the reader are

$$\text{read}(R, 2), \text{read}(R, 1)$$

in this order, whereas the operations by the writer are

$$\text{write}(R, 1), \text{write}(R, 2)$$

Any linearization of these operations must preserve the order between the read operations as well as the order between the write operations. However, to preserve the semantics of the register R, write$(R, 2)$ must appear before read$(R, 2)$, while write$(R, 1)$ must appear before read$(R, 1)$ and after read$(R, 2)$. Thus, either the order between the read operations or the order between write operations should be reversed.

To avoid this problem of "new-old" inversion of values, two changes are made: (1) a write operation clears *only* the entries whose indices are smaller than the value it is writing, and (2) a read operation does not stop when it finds the first 1 but makes sure there are still zeroes in all lower indices. Specifically, the reader scans from the low values toward the high values until it finds the first 1; then it reverses direction and scans back down to the beginning, keeping track of the smallest index observed

[1] This scenario suggests another problem: if no write follows the first write, then the reader will run off the end of the array in the first read, as it will observe zeroes in all the binary registers, causing the high-level read to be undefined.

Algorithm 26 Simulating a multi-valued register R from binary registers: code for reader and writer.

Initially the shared registers $B[0]$ through $B[K-1]$ are all 0, except $B[i] = 1$, where i is the initial value of R

```
1:  when read(R) occurs:                          // the reader reads from register R
2:      i := 0
3:      while B[i] = 0 do i := i + 1              // upward scan
4:      up, v := i
5:      for i = up − 1 downto 0 do                // downward scan
6:          if B[i] = 1 then v := i
7:      return(R, v)

8:  when write(R, v) occurs:          // the writer writes the value v to register R
9:      B[v] := 1
10:     for i := v − 1 downto 0 do B[i] := 0
11:     ack(R)
```

to contain a 1 during the downward scan. This is the value returned. The pseudocode appears as Algorithm 26.

Clearly, the algorithm is wait-free: Each write executes at most K low-level write operations, whereas each read executes at most $2K - 1$ low-level read operations. To show that the algorithm is correct, we need to show that each of its admissible executions is linearizable, that is, there is permutation of the high-level operations that preserves the order of non-overlapping operations, and in which every read returns the value of the latest preceding write. The proof technique is to explicitly describe an ordering of the operations that satisfies the semantics and then show that it respects the order of operations; a similar approach was used in the proof of Theorem 9.5 in Chapter 9.

Consider any admissible execution α of the algorithm and fix a linearization of the operations on the low-level binary registers. Such a linearization exists because α is admissible.

In the rest of this section, high-level operations are capitalized (e.g., Read), whereas low-level operations are not (e.g., read). We say that a (low-level) read r of any $B[v]$ in α *reads from* a (low-level) write w to $B[v]$ if w is the latest write to $B[v]$ that precedes r in the linearization of the operations on $B[v]$. We say that a (high-level) Read R in α *reads from* a (high-level) Write W if R returns v and W contains the write to $B[v]$ that R's last read of $B[v]$ reads from. Note that W writes the value v, by writing a 1 in $B[v]$, and thus a Read that returns v reads from a Write that writes v.

Construct a permutation π of the high-level operations in α as follows. First, place all the Writes in the order in which they occur in α; because there is only one writer, this order is well-defined.

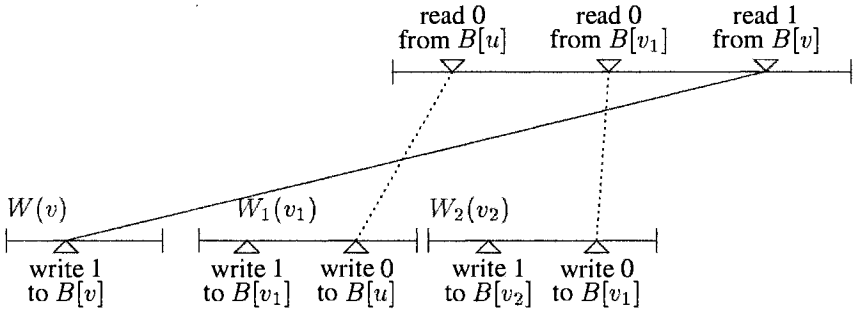

Fig. 10.3 Scenario for proof of Lemma 10.1; dotted lines indicate low-level reads-from relationships, and solid line indicates high-level reads-from relationship.

Consider the Reads in the order in which they occur in α; because there is only one reader, this order is well-defined. For each Read R, let W be the Write that R reads from. Place R in π immediately before the Write in π just following W. The purpose is to place R after W and after all previous Reads that also read from W (which have already been placed after W).

We must show that π is a linearization of α. First note that π satisfies the sequential specification of a read write register by construction. Now we show that the real-time ordering of operations in α is preserved in π. For any pair of Writes, this is preserved by construction. If a Read R finishes in α before Write W begins, then clearly R precedes W in π, because R cannot read from a Write that starts after R ends.

To argue the two other cases, we use Lemma 10.1, which constrains the Writes that a Read can read from in certain situations.

Lemma 10.1 *Consider two values u and v with $u < v$. If Read R returns v and R's read of $B[u]$ during its upward scan reads from a write contained in Write W_1, then R does not read from any Write that precedes W_1.*

Proof. Suppose in contradiction that R reads from a Write W that precedes W_1 (see Fig. 10.3). Let v_1 be the value written by W_1. Since W_1 writes a 1 to $B[v_1]$ and then does a downward scan, $v_1 > u$. Also, $v_1 < v$, since otherwise W_1 would overwrite W's write to v, contradicting the fact that R reads from W.

Thus in R's upward scan, it reads $B[v_1]$ after $B[u]$ and before $B[v]$. R must read a 0 in $B[v_1]$ since otherwise it would not return v. Consequently there must be another write W_2, after W_1, that writes 0 in $B[v_1]$ before R's read of $B[v_1]$. Let v_2 be the value written by W_2. Note that v_2 must be less than v_1 for the same reason that v_1 is less than v.

Similarly, W_2 must be followed by a Write W_3 that writes v_3, with $v_2 < v_3 < v$, and so on.

Thus there exists an infinite increasing sequence of integers $v_1, v_2, v_3, \ldots$, all of which are less than v. This is a contradiction. □

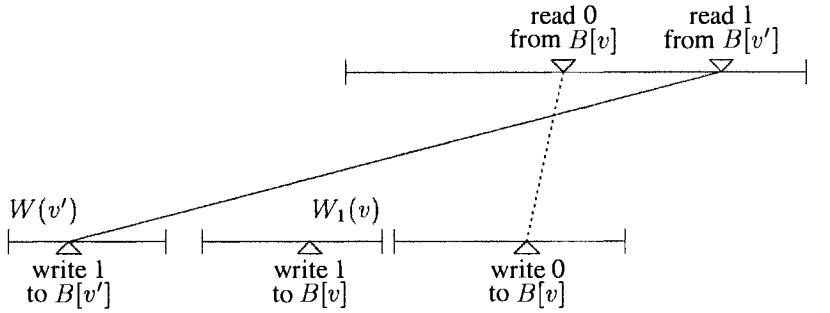

Fig. 10.4 Scenario for the Write-before-Read case.

We now return to verifying that π, the permutation defined above, respects the ordering of non-overlapping high-level operations in the execution α.

Case 1: Write before Read: Suppose in contradiction that Write W finishes in α before Read R begins but in π, R is placed before W. Then R reads from some Write W' that precedes W. Let W write v and W' write v'. Thus the value of R is v'.

First, assume $v' \leq v$. Then W overwrites the write to $B[v']$ by W' before R begins, and R cannot read from W'.

Now consider the case $v' > v$ (see Fig. 10.4). W writes a 1 in $B[v]$. Since R does not see this 1 and stop at $B[v]$ during its upward scan, there must be a Write after W containing a write of 0 to $B[v]$ that R's read of $B[v]$ reads from. Then, by Lemma 10.1, R cannot read from W'.

Case 2: Read before Read: Suppose in contradiction that Read R_1 precedes Read R_2 in α but follows R_2 in π. This inversion implies that R_1 reads from a Write $W_1(v_1)$ that follows the Write $W_2(v_2)$ that R_2 reads from.

First, consider the case where $v_1 = v_2$. Then when W_1 writes 1 to $B[v_1]$, it overwrites the 1 that W_2 wrote to $B[v_2]$ earlier. To be consistent with real time, the operations on $B[v_1]$ must be linearized in the order: W_2's write, W_1's write, R_1's last read, R_2's last read. This contradicts the assumption that R_2 reads from W_2.

Since v_1 and v_2 must therefore be distinct, we next consider the case where $v_1 > v_2$ (see Fig. 10.5). Since R_2 reads from W_2, no write to $B[v_2]$ is linearized between W_2's write of 1 to $B[v_2]$ and R_2's last read of $B[v_2]$. Since R_1 reads from W_1, W_1's write of 1 to $B[v_1]$ precedes R_1's last read of $B[v_1]$. So $B[v_2]$ has the value 1 starting before R_1 does its last read of $B[v_1]$ and ending after R_2 does its last read of $B[v_2]$. But then R_1's read of $B[v_2]$ during its downward scan would return 1, not 0, a contradiction since R_1 returns v_1, which is larger than v_2.

Finally, we consider the case where $v_1 < v_2$ (see Figure 10.6). Since R_1 reads from W_1, W_1's write of 1 to $B[v_1]$ precedes R_1's last read of $B[v_1]$. Since R_2 returns $v_2 > v_1$, R_2's first read of $B[v_1]$ must return 0. So there must be another Write after W_1 containing a write of 0 to $B[v_1]$ that R_2's read of $B[v_1]$ reads from. Then, by Lemma 10.1, R cannot read from W'.

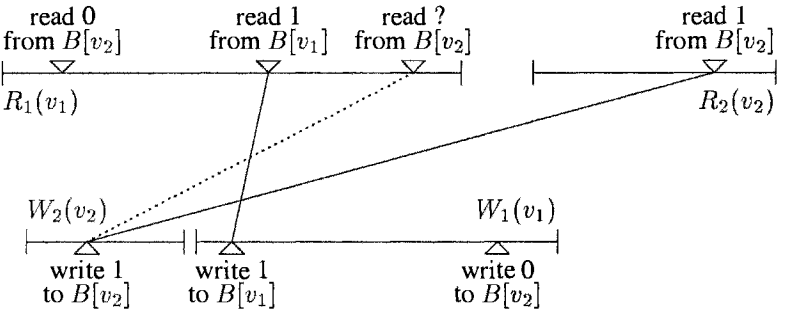

Fig. 10.5 Scenario for the Read-before-Read case when $v_1 > v_2$.

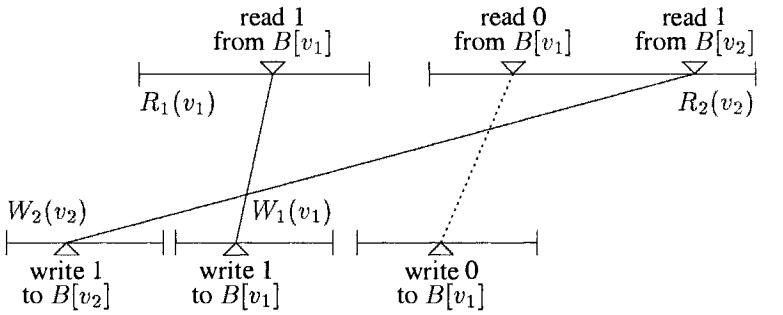

Fig. 10.6 Scenario for the Read-before-Read case when $v_1 < v_2$.

Thus π is a linearization of α and we have:

Theorem 10.2 *There exists a wait-free simulation of a K-valued register using K binary registers in which each high-level operation performs $O(K)$ low-level operations.*

10.2.2 Multi-Reader from Single-Reader

The next step we take in the simulation of general read/write registers is to allow several processors to read from the same register; we still allow only one processor to write the register. That is, we use single-writer single-reader registers to build a wait-free simulation of a single-writer *multi*-reader register.

Let n be the number of reading processors to be supported by the multi-reader register. A simple idea for this simulation is to use a collection of n shared single-writer single-reader registers, $Val[i]$, $i = 1, \ldots, n$, one for each reader. In a write operation, the writer stores the new value in the registers, one after the other. In a read operation, the read returns the value in its register. The simulation is clearly wait-free. However, it is incorrect, that is, it has executions that are not linearizable.

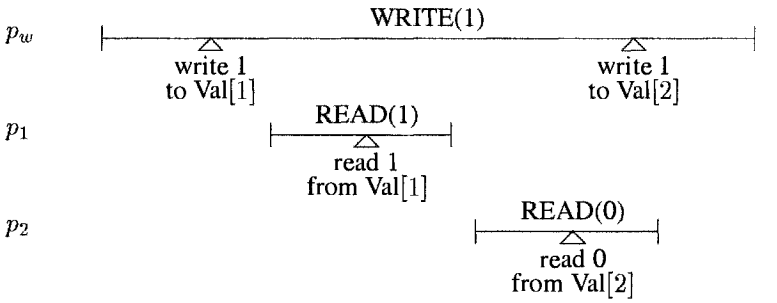

Fig. 10.7 A counterexample to the multi-reader algorithm.

Consider for example the following execution in which a new-old inversion of values happens, that is, a later read operation returns the value of an earlier write.

Assume that the initial value of the register is 0 and that the writer, p_w, wishes to write 1 to the register. Consider the following sequence of operations (see Fig. 10.7):

- p_w starts the write operation and begins writing 1 to *Val*[1].
- p_1 reads the value 1 from *Val*[1] and returns 1.
- p_2 reads the value 0 from *Val*[2] and returns 0.
- p_w finally receives the ack for its write to *Val*[1], writes 1 to *Val*[2], and returns.

This execution cannot be linearized because the read of p_1 should be ordered after the write of p_w and the read of p_2 should be ordered before the write of p_w; therefore, the read of p_1 should be ordered after the read of p_2; however, the read of p_1 occurs strictly before the read of p_2.

One might be tempted to require the readers to read again and again, to use more registers, or to require the writer to write more values. None of these fixes is correct (see Exercise 10.3 for an example), as shown by Theorem 10.3.

Theorem 10.3 *In any wait-free simulation of a single-writer multi-reader register from any number of single-writer single-reader registers, at least one reader must write.*

Proof. Suppose in contradiction there exists such a simulation for a register R in which the readers do not write. Let p_w be the writer and p_1 and p_2 the readers of R. Suppose the initial value of R is 0.

Since the readers do not write, the execution of the writer is unaffected by concurrent reading.

Since the registers used by the simulation are single-reader, they can be partitioned into two sets, S_1 and S_2, such that only the reader p_1 reads the registers in S_1 and only the reader p_2 reads the registers in S_2.

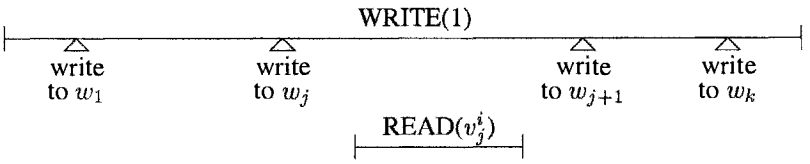

Fig. 10.8 Illustration of α_j^i in the proof of Theorem 10.3.

Consider the execution α of a high-level write of 1 to R, starting in the initial configuration. The writer p_w performs a series of low-level writes, $w_1, \ldots, w_k$, on the single-reader registers. Each w_i involves a register in either S_1 or S_2.

For each $i = 1, 2$, and each $j = 0, \ldots, k$, define an alternative execution α_j^i obtained from α by interposing a high-level read operation by p_i after the linearization point of w_j (if this write exists) and before the linearization point of w_{j+1} (if this write exists). Let v_j^i be the value returned by this high-level read (see Fig. 10.8). That is, we check to see what value would be returned by each reader after each low-level write.

Since the simulation guarantees linearizability, for each $i = 1, 2$, there exists j_i between 1 and k such that $v_j^i = 0$ for all $j < j_i$ and $v_j^i = 1$ for all $j \geq j_i$. That is, there is a single low-level write operation that causes p_i to observe the simulated register as having changed its value from 0 to 1.

It is crucial to realize that j_1 cannot equal j_2. The reason is that w_{j_1} writes some register in S_1, the set of registers that p_1 can read. (See Exercise 10.4.) Similarly, w_{j_2} writes some register in S_2, the set of registers that p_2 can read. Since S_1 and S_2 are disjoint, j_1 cannot equal j_2.

Without loss of generality, assume $j_1 < j_2$. Let α' be an execution obtained from α by inserting a read by p_1 followed by a read by p_2 after w_{j_1} and before w_{j_1+1}.

By definition of j_1 and j_2 and the assumption that $j_1 < j_2$, p_1's read returns 1 and p_2's read returns 0 in α'. But this contradicts the assumption that the simulation guarantees linearizability, since p_1's read precedes p_2's, yet sees the newer value. □

To avoid the ordering problem described above, the readers write to each other (through additional registers), creating an ordering among them. Before a reader returns from a read operation, it announces the value it has decided to return. A reader reads not only the value written for it by the writer, but also the values announced by the other readers. It then chooses the most recent value among the values it has read.

Crucial to this algorithm is the ability to compare different values and choose the most recent one among them. We assume that registers can take on an unbounded number of values. Therefore, the writer chooses a nonnegative integer to associate as a *sequence number*, or *timestamp*, with each value it writes. The writer increments the sequence number each time it wishes to write a new value. Clearly, the value associated with the largest sequence number among a set of values is the one written by the most recent write.

Algorithm 27 Simulating a multi-reader register R from single-reader registers: code for readers and writer.

Initially $Report[i, j] = Val[i] = (v_0, 0), 1 \leq i, j \leq n$,
where v_0 is the initial value of R

```
1:  when read_r(R) occurs:                    // reader p_r reads from register R
2:      (v[0], s[0]) := Val[r]                // most recent value reported to p_r by writer
3:      for i := 1 to n do
        (v[i], s[i]) := Report[i, r]  // most recent value reported to p_r by reader p_i
4:      let j be such that s[j] = max{s[0], s[1], ..., s[n]}
5:      for i := 1 to n do Report[r, i] := (v[j], s[j]) // p_r reports to each reader p_i
6:      return_r(R, v[j])

7:  when write(R, v) occurs:                  // the writer writes v to register R
8:      seq := seq + 1                        // local variable seq = 0 initially
9:      for i := 1 to n do Val[i] := (v, seq)
10:     ack(R)
```

In the rest of this section, we refer to a *value* as a pair containing a value of the high-level register and a sequence number.

The pseudocode for a register supporting one writer p_w and n readers $p_1, \ldots, p_n$ appears in Algorithm 27; it uses the following shared arrays of single-writer single-reader read/write registers:

***Val*[i]:** The value written by p_w for each reader p_i, $1 \leq i \leq n$.

***Report*[i, j]:** The value returned by the most recent read operation performed by p_i; written by p_i and read by p_j, $1 \leq i, j \leq n$.

Each processor has two local arrays, v and s, each of which is an $(n + 1)$-element array, that hold the values and corresponding sequence numbers respectively reported most recently by the writer and n readers.

The algorithm is clearly wait-free: Each simulated operation performs a fixed number of low-level operations—n for a write operation and $2n + 1$ for a read operation. To prove that the simulation is correct it remains to show that every admissible execution is linearizable.

Consider any admissible execution α. To show that α is linearizable, we have to show that there is a permutation π of the high-level operations in α that preserves the order of non-overlapping operations, and in which every read operation returns the value of the latest preceding write.

For this algorithm, we prove linearizability by explicitly constructing π, as was done in Section 10.2.1. We construct π in two steps.

First, we put in π all the write operations according to the order in which they occur in α; because write operations are executed sequentially by the unique writer,

this sequence is well-defined. Note that this order is consistent with that of the timestamps associated with the values written.

Next, we add the read operations to π. We consider the reads one by one, in the order of their responses in α. A read operation that returns a value with timestamp T is placed immediately before the write that follows (in π) the write operation that generated timestamp T. (If there is no such write, then it goes at the end.)

By the defined placement of each read, every read returns the value of the latest preceding write and therefore π is legal.

Lemma 10.4 shows that π preserves the real-time order of non-overlapping operations.

Lemma 10.4 *Let op_1 and op_2 be two high-level operations in α such that op_1 ends before op_2 begins. Then op_1 precedes op_2 in π.*

Proof. By construction, the real-time order of write operations is preserved.

Consider some read operation r by p_i that returns a value associated with timestamp T.

Consider a write w that follows r in α. Suppose in contradiction that read r is placed after write w in π. Then the write w' that generates timestamp T must be either w or a later write, implying that w' occurs after r in α, which is a contradiction.

Consider a write w that precedes r in α. Since r occurs after w, r reads from $Val[i]$ the value written by w or a later write, by the linearizability of $Val[i]$. By the semantics of max on integers and because timestamps are increasing, r returns a value whose associated timestamp is generated by w or a later write. Thus r is not placed before w in π.

Consider a read r' by p_j that follows r in α. By linearizability, p_j obtains a timestamp from $Report[i]$ during r' that is written during r or later. Since the timestamps are increasing integers, no timestamp written to $Report[i]$ after r was generated before T was generated. Thus the max in r' returns a timestamp that was generated at least as recently as T, and thus r' will not be placed before r. □

Note that the simulation of a single register requires n low-level (single-reader) registers for communication from the writer to the readers and $n(n-1)$ low-level registers for communication among the n readers.

Thus we have:

Theorem 10.5 *There exists a wait-free simulation of an n-reader register using $O(n^2)$ single-reader registers in which each high-level operation performs $O(n)$ low-level operations.*

10.2.3 Multi-Writer from Single-Writer

The final step we make in this section is to construct a multi-writer multi-reader read/write register from single-writer multi-reader registers. As for the previous simulation, we assume here that the registers can hold an unbounded number of values.

The idea of the algorithm is to have each writer announce each value it desires to write to all the readers, by writing it in its own (single-writer multi-reader) register; each reader reads all the values written by the writers and picks the most recent one among them.

Once again, the crucial issue is to find a way to compare the values written by the writes and find the most recent one among them. To achieve this, we assign a timestamp to each value written. Unlike the previous algorithm, in this one the timestamps are not generated by a single processor (the single writer), but by several processors (all possible writers). The most important requirement of the timestamps is that they be totally ordered. Furthermore, the timestamps should reflect the order of non-overlapping operations; that is, if a write operation completely precedes another write operation then the first operation should get a lower timestamp.

Interestingly, we have already seen a method for creating such timestamps, in the context of vector clocks (Chapter 6). Recall that these timestamps are vectors of length m, where m is the number of writers, and that the algorithm for picking a timestamp is as follows: The new timestamp of a processor is the vector consisting of the local timestamps read from all other processors, with its local timestamp increased by 1.

We apply lexicographic order to the vectors of timestamps; that is, two vectors are ordered according to the relative order of the values in the first coordinate in which the vectors differ. This is a total order extending the partial order defined for vector timestamps in Chapter 6 (Section 6.1.3).

The m writers are $p_0, \ldots, p_{m-1}$, and all of the processors, $p_0, \ldots, p_{n-1}$, are the readers. The pseudocode appears in Algorithm 28. It uses the following shared arrays of single-writer multi-reader read/write registers:

***TS*[*i*]:** The vector timestamp of writer p_i, $0 \leq i \leq m-1$. It is written by p_i and read by all writers.

***Val*[*i*]:** The latest value written by writer p_i, $0 \leq i \leq m-1$, together with the vector timestamp associated with that value. It is written by p_i and read by all readers.

Clearly, the algorithm is wait-free because any simulated operation requires a linear number of low-level operations.

To prove that the simulation is correct we have to show that every admissible execution is linearizable. The proof is very similar to the linearizability proof of the previous simulation.

Consider any admissible execution α. To show α is linearizable, we have to show that there is a permutation π of the high-level operations in α that preserves the order of non-overlapping operations, and in which every read operation returns the value of the latest preceding write.

The key to the proof is the lexicographic order on timestamps. The proof of Lemma 10.6 is left as an exercise to the reader (Exercise 10.6).

Lemma 10.6 *The lexicographic order of the timestamps is a total order consistent with the partial order in which they are generated.*

Algorithm 28 Simulating a multi-writer register R from single-writer registers: code for readers and writers.

Initially $TS[i] = \langle 0, \ldots, 0 \rangle$ and
$Val[i]$ equals the desired initial value of R, $0 \leq i \leq m-1$

```
1:  when read_r(R) occurs:          // reader p_r reads from register R, 0 ≤ r < n
2:      for i := 0 to m − 1 do (v[i], t[i]) := Val[i]      // v and t are local
3:      let j be such that t[j] = max{t[0], ..., t[m − 1]}   // lexicographic max
4:      return_r(R, v[j])

5:  when write_w(R, v) occurs: // writer p_w writes v to register R, 0 ≤ w ≤ m − 1
6:      ts := NewCTS()                                          // ts is local
7:      Val[w] := (v, ts)                                  // write to shared register
8:      ack_w(R)

9:  procedure NewCTS_w():          // writer p_w obtains a new vector timestamp
10:     for i := 0 to m − 1 do
11:         lts[i] := TS[i].[i]    // extract the ith entry from TS of ith writer
12:     lts[w] := lts[w] + 1                                  // increment own entry
13:     TS[w] := lts                                       // write to shared register
14:     return lts
```

Inspection of the pseudocode reveals that the values written to each TS variable are written in nondecreasing lexicographic order, and therefore we have the following lemma:

Lemma 10.7 *For each i, if vector timestamp VT_1 is written to $Val[i]$ and later vector timestamp VT_2 is written to $Val[i]$, then $VT_1 \leq VT_2$.*

The candidate linearization order π is defined in two steps, in a manner similar to that in Section 10.2.2.

First, we put into π all the write operations according to the lexicographic ordering on the timestamps associated with the values they write. Second, we consider each read operation in turn, in the order in which its response occurs in α. A read operation that returns a value associated with timestamp VT is placed immediately before the write operation that follows (in π) the write operation that generated VT.

By the defined placement of each read, π is legal, since each read returns the value of the latest preceding write.

Lemma 10.8 shows that π preserves the real-time order of non-overlapping operations.

Lemma 10.8 *Let op_1 and op_2 be two high-level operations in α such that op_1 ends before op_2 begins. Then op_1 precedes op_2 in π.*

Proof. By construction and Lemma 10.6, the real-time order of write operations is preserved.

Consider a read operation, r, by p_i that returns a value associated with timestamp VT.

Consider a write w that follows r in α. Suppose in contradiction that read r is placed after write w in π. Then the write w' that generates timestamp VT must be either w or a later write, implying that w' occurs in α after r, a contradiction.

Consider a write w by p_j that precedes r in α. Since r occurs after w, r reads from $Val[j]$ the value written by w or a later write, by the linearizability of $Val[j]$. By the semantics of max on vectors of integers and Lemma 10.6, r returns a value whose associated timestamp is generated by w or a later write. Thus r is not placed before w in π.

Consider a read r' by p_j that follows r in α. During r, p_i reads all *Val* variables and returns the value whose associated timestamp is the lexicographic maximum. Later, during r', p_j does the same thing. By Lemma 10.7, the timestamps appearing in each *Val* variable are in non-decreasing lexicographic order, and thus by Lemma 10.6 the timestamps are in non-decreasing order of when they were generated. Thus r' obtains timestamps from the *Val* variables that are at least as recent as those obtained by r. By the semantics of max on vectors of integers, the timestamp associated with the value returned by r' is at least as recent as the timestamp associated with the value returned by r. Thus r' is not placed before r in π. □

Note that the construction of a single register requires $O(m)$ low-level (single-writer) registers, where m is the number of writers.

Thus we have:

Theorem 10.9 *There exists a wait-free simulation of an m-writer register using $O(m)$ single-writer registers in which each high-level operation performs $O(m)$ low-level operations.*

10.3 ATOMIC SNAPSHOT OBJECTS

The shared objects we have seen so far in this chapter allow only a single data item to be accessed in a memory operation, although they allow several processors to access it (for reading, writing, or both). We now turn to atomic snapshot objects—shared objects partitioned into segments. Each segment belongs to a different processor and is written separately, but all segments can be read at once by any processor. In this section, we present a wait-free simulation of an atomic snapshot object from bounded-size read/write registers.

As we shall see (e.g., in Chapter 16), the ability to read all segments atomically can simplify the design and verification of shared memory algorithms. Because they can be simulated from read/write registers, there is no loss of generality in assuming the existence of atomic snapshot objects.

The sequential specification of an *atomic snapshot object* provides two kinds of operations for each user i, $0 \leq i \leq n-1$:

- A scan_i invocation whose response is $\text{return}_i(V)$, where V is an n-element vector called a *view* (with a value for each segment), and
- An $\text{update}_i(d)$ invocation whose response is ack_i, where d is the data to be written to p_i's segment

A sequence of scan and update operations is in the allowable set if and only if, for each V returned by a scan operation, $V[i]$ equals the parameter of the latest preceding update_i operation, for all i. If there is no preceding update_i operation, then $V[i]$ equals the initial value of p_i's segment of the object.

We describe a simulation of an atomic snapshot object from single-reader, single-writer read/write registers of bounded size.

The main idea of the scan is to collect (i.e., read) all the segments twice; this is called a *double collect*. If no segment is updated during the double collect, then the result of each collect is clearly a snapshot, as no updates occur in between the two collects. When a successful double collect occurs, the scan can return; this is the crux of the algorithm. There are two difficulties in implementing this idea: how to tell if the segments have been updated and what to do if the segments have been updated.

A simple way to solve the first problem is to include an (unbounded) sequence number with the data item in each segment (see the chapter notes). A more space-efficient solution is to employ a *handshaking mechanism*. The handshaking mechanism is not powerful enough to indicate precisely how many changes have been made; instead, it can indicate whether at least one change has been made to a segment or whether at most one change has been made.

To solve the problem of what to do if a change to a segment is observed, we show that if a scanner observes several changes in the segment of a specific updater then the updater has performed a complete update during the scan. We embed a scan operation at the beginning of the update; the view obtained in this scan is written with the data. The scanner returns the view obtained in the last collect. As we prove below, this view is an allowed response for this scan.

10.3.1 Handshaking Procedures

We now describe the general method of handshaking that provides two properties that are crucial to the correct operation of the atomic snapshot algorithm.

Consider a fixed ordered pair of distinct processors (p_i, p_j). Four procedures are defined for this ordered pair[2]: tryHS_i, tryHS_j, checkHS_i, and checkHS_j. The procedures interact via two shared single-reader, single-writer binary read/write registers, called the *handshaking bits*: h_i, which is written by p_i and read by p_j, and h_j, which is written by p_j and read by p_i. Processor p_i tries to handshake by modifying its own bit to make the two bits equal, whereas p_j tries to make the bits unequal. Then p_i checks whether a handshake occurred (informally, whether p_j tries to handshake

[2]The atomic snapshot algorithm only uses three of the procedures; for generality we include the fourth one.

Algorithm 29 Handshaking procedures for the ordered pair of processors (p_i, p_j).

```
1:  procedure tryHS_i():                    // p_i tries to handshake
2:      h_i := h_j                          // by trying to make the bits equal
3:      return

4:  procedure tryHS_j():                    // p_j tries to handshake
5:      h_j := ¬h_i                         // by trying to make the bits unequal
6:      return

7:  procedure checkHS_i():                  // p_i checks whether a handshake occurred
8:      return (h_i ≠ h_j)

9:  procedure checkHS_j():                  // p_j checks whether a handshake occurred
10:     return (h_j = h_i)
```

between the time of p_i's last handshake try and the check) by testing whether the bits are unequal. To check whether a handshake occurs, p_j tests whether the bits are equal.

The pseudocode for the handshaking procedures appears in Algorithm 29. To make the code more readable, it appears as if both processors read each bit, but because each bit is single-writer, the writer can simply remember in a local variable what it last wrote to the bit.

Consider an admissible execution containing multiple calls to the handshaking procedures and a fixed linearization of the read and write operations embedded in the procedures. In the sequel we refer to these operations as occurring before, after, or between other operations with respect to this linearization. The properties we wish to have are (cf. Fig. 10.9):

Handshaking Property 1: If a checkHS$_i$ returns true, then there exists a tryHS$_j$ whose write occurs between the read of the previous tryHS$_i$ and the read of the checkHS$_i$. The same is also true with i and j reversed.

Handshaking Property 2: If a checkHS$_i$ returns false, then there is no tryHS$_j$ whose read and write occur between the write of the previous tryHS$_i$ and the read of the checkHS$_i$. The same is also true with i and j reversed.

Note that Handshaking Property 2 is not quite the negation of Handshaking Property 1; this uncertainty is the weakness mentioned earlier in our discussion of the mechanism.

The following theorem proves that the handshaking properties above are ensured by the procedures of Algorithm 29.

Theorem 10.10 *Consider any execution of the handshaking procedures that is correct for the read/write registers communication system, in which only the handshaking procedures alter the handshaking bits. Then the procedures satisfy the handshaking properties.*

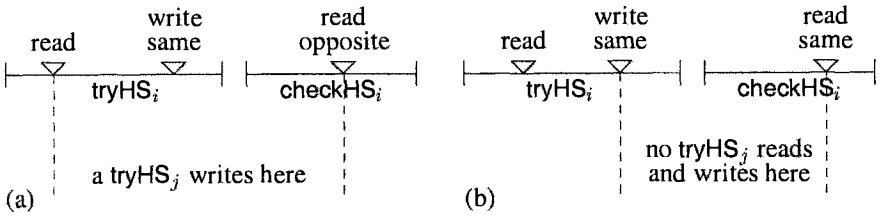

Fig. 10.9 Demonstrations for Handshaking Properties 1 (a) and 2 (b): Triangles indicate linearization points of low-level operations.

Proof. We prove the handshaking properties as stated; the proofs when i and j are reversed are left as an exercise (cf. Exercise 10.7).

Handshaking Property 1: Suppose a call to checkHS$_i$ returns true. Then p_i observes the bits to be unequal, that is, its read of h_j returns the opposite of what was written to h_i in the preceding tryHS$_i$. Without loss of generality, suppose p_i reads h_j to be 1 in the checkHS$_i$, while h_i is 0. Then in the preceding tryHS$_i$, p_i wrote a 0 to h_i because it observed h_j to be 0. Thus there must be a call to tryHS$_j$ whose write to h_j (changing it from 0 to 1) is linearized between the specified points.

Handshaking Property 2: Suppose a call to checkHS$_i$ returns false. Then p_i observes the bits to be equal, that is, its read of h_j returns the same value that was written to h_i in the preceding tryHS$_i$. Without loss of generality, suppose this value is 0. Suppose in contradiction there is a call to tryHS$_j$ whose read and write occur between the specified points. In particular, the read of h_i in tryHS$_j$ follows the prior tryHS$_i$, and will return 0; hence, 1 will be written to h_j in tryHS$_j$. Any subsequent calls to tryHS$_j$ in that interval will do the same. But then the read of h_j in checkHS$_i$ returns 1, not 0, a contradiction. □

10.3.2 A Bounded Memory Simulation

For each processor p_i acting as a scanner and each (distinct) processor p_j acting as an updater, we have a separate handshaking mechanism in the atomic snapshot simulation, denoted by the ordered pair (p_i, p_j). The first entry in the pair indicates the scanner, and the second indicates the updater. Thus we need a way to distinguish the procedure names and the variable names so that, for instance, the variables for the pair (p_i, p_j) will be distinct from those for (p_j, p_i) and so that the procedure names will not be confused. We will use the superscript (i, j) on the procedure and variable names for the ordered pair (p_i, p_j), for example, $h_i^{(i,j)}$ and checkHS$_i^{(i,j)}$.

A scanner tries to handshake with all the other processors, then does the double collect, and then checks the handshakes with all the other processors. Because handshaking and collecting, on the scanner's side, and writing and handshaking, on the updater's side, are done in separate operations, an update to some segment may not be observed (see Fig. 10.10). To differentiate two consecutive updates to the

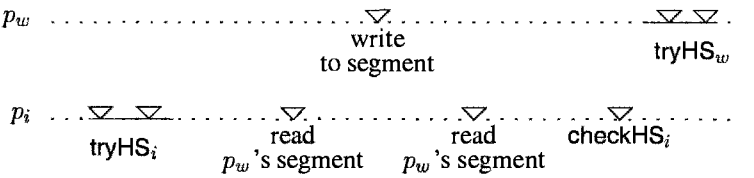

Fig. 10.10 Why toggle bits are needed.

same segment, we include a bit with the data of each segment, which the updater toggles in each update.

The scanner, p_i, repeatedly tries to handshake, double collects, and checks for handshakes until it observes three changes in the segment of some processor p_j. The implication of observing three changes in p_j's segment is that p_j has performed a complete update during p_i's scan. We alter the update code to first perform a scan and then include the view returned by the scan with the data and toggle bit. The scanner p_i returns the view obtained from p_j in the last collect.

The pseudocode appears as Algorithm 30. Each processor p_i has a local array $shook[0 \ldots n-1]$, which counts, for each updater p_j, the number of times p_j was observed by p_i to have tried to handshake with p_i.

The following lemmas are with respect to an admissible execution α of Algorithm 30. When we refer to a scan execution, we mean the execution of the procedure scan inside either a scan or an update operation.

Lemma 10.11 explains how the condition in Line 15 "catches" an intervening update operation.

Lemma 10.11 *If, during some execution by p_i of Line 15 in the* scan *procedure, the condition returns true for some j, then p_j executes either Line 6 or the write of the handshake with p_i in Line 7 between the previous execution in Line 12 of the read of* tryHS$_i^{(i,j)}$ *and this execution of Line 15.*

Proof. The condition in Line 15 is true for j either because $a[j].toggle \neq b[j].toggle$ or because checkHS$_i^{(i,j)}$ returns true. In the first case, p_j writes to $Segment[j]$ (Line 6) during p_i's double collect (Lines 13–14). In the second case, Handshaking Property 1 implies that there has been a write of tryHS$_j^{(i,j)}$ (Line 7) since the read of the previous tryHS$_i^{(i,j)}$ (Line 12). □

Note that a scan execution can either return a view in Line 18, in which case it is called a *direct* scan, or borrow a view, in Line 16, in which case it is called an *indirect* scan. Lemma 10.12 indicates why it is reasonable for an indirect scan to borrow the view returned by another scan.

Lemma 10.12 *An indirect* scan *returns the view of a direct* scan *whose execution is enclosed within the execution of the indirect* scan.

Algorithm 30 Wait-free atomic snapshot object simulation from read/write registers: code for processor p_i, $0 \leq i \leq n-1$.

Initially $Segment[i].data = v_i$, $Segment[i].view$ equals $\langle v_0, \ldots, v_{n-1} \rangle$, where v_j is the initial value of p_j's segment, and (local) $shook[j] = 0$, for all j

```
1:  when scan_i() occurs:                                   // scan_i is an input event
2:      view := scan()                                      // scan is a procedure; view is local
3:      return_i(view)                                      // return_i is an output event

4:  when update_i(d) occurs:          // processor p_i updates its segment to hold d
5:      view := scan()
6:      Segment[i] := (d, view, ¬Segment[i].toggle)        // flip the toggle bit
7:      for all j ≠ i do tryHS_i^(j,i)()                   // handshake with all scanners
8:      ack_i

9:  procedure scan() :
10:     for all j ≠ i do shook[j] := 0                      // shook is local
11:     while true do
12:         for all j ≠ i do tryHS_i^(i,j)()                // handshake with all updaters
13:         for all j ≠ i do a[j] := Segment[j]              // first collect
14:         for all j ≠ i do b[j] := Segment[j]              // second collect
15:         if, for some j ≠ i, checkHS_i^(i,j)()
            or (a[j].toggle ≠ b[j].toggle) then             // update progress observed
16:             if shook[j] = 2 then return(b[j].view)      // indirect scan
17:             else shook[j] := shook[j] + 1
18:         else return(⟨b[0].data, ..., b[n − 1].data⟩)    // direct scan
```

Proof. Let p_i be the processor that executes the indirect scan. Assume that the indirect scan borrows a view to return from processor p_j. Thus p_i's indirect scan evaluates the condition in Line 15 to be true for j three times. By Lemma 10.11, p_j performs Line 6 or Line 7 in three distinct intervals; hence, p_j performs Line 6 or Line 7 three different times. It follows that the third execution by p_j of Line 6 or 7 is part of an update$_j$ operation that starts after p_i's scan starts. The reason is that a single update operation can cause the condition in Line 15 to be true twice, first in Line 6 and then in Line 7. Thus the scan embedded in that update$_j$ operation, which provides the borrowed view, is enclosed within the execution of the indirect scan of p_i.

If that embedded scan is direct, we are done. If not, then this argument can be applied inductively, noting that there can be at most n concurrent operations in the system. Hence, eventually the embedded scan is direct, and the result follows by the transitivity of containment of the embedded scan intervals. □

Because the execution α is correct for the read/write registers communication system, the reads and writes on the *Segment*[i] shared registers are linearizable. For each configuration C_k in α, define the *actual value of the snapshot object* to be the vector $\langle d_0, \ldots, d_{n-1} \rangle$, where d_i is the first parameter of the latest write by processor p_i to *Segment*[i] (in Line 6) that is linearized before C_k. If there is no such write, then d_i is the initial value of *Segment*[i].*data*.

A direct scan has a successful double collect when the test in Line 15 is false for all j. That is, no processor is observed to make progress in between the two collects in Lines 13 and 14. Lemma 10.13 proves that the values returned by a direct scan constitute a "snapshot" after the first collect of the successful double collect.

Lemma 10.13 *A direct* scan *in α returns the actual value of the atomic snapshot object in the configuration immediately following the last read in the first collect of the successful double collect.*

Proof. Suppose p_i performs a direct scan. Let p_j be any other processor.

Consider the behavior of p_i during the final execution of the while loop (Lines 11–18). Let s_i be the $\mathsf{tryHS}_i^{(i,j)}$ execution in Line 12. Let r_1 be the linearization point of the last read in the first collect (Line 13). Let r_2 be the linearization point of the read of *Segment*[j] in the second collect (Line 14). Let c_i be the $\mathsf{checkHS}_i^{(i,j)}$ execution in Line 15.

Since the direct scan returns the value read at r_2, we must show that no write to *Segment*[j] is linearized in between r_1 and r_2.

Since c_i returns false, Handshaking Property 2 implies that there is no complete tryHS_j execution between the point of s_i and the point of c_i. Thus at most one write to *Segment*[j] can take place in that interval. If there were one such write, and it took place between r_1 and r_2, then it would cause the toggle bit to change between the reads of *Segment*[j] in the two collects. This contradicts the fact that the if condition in Line 15 was false. □

To prove linearizability, we identify a linearization point for each operation, inside its interval, in a way that preserves the semantics of the snapshot object. The proposed linearization for the scan and update operations in α is the result of ordering the operations according to these linearization points.

A scan operation whose embedded call to procedure scan is direct is linearized immediately after the last read of the first collect in the successful double collect. A scan operation whose embedded call to procedure scan is indirect is linearized at the same point as the direct scan whose view is borrowed. Lemma 10.12 guarantees that such a direct scan exists and is entirely contained in the interval of the scan operation. Thus all scan operations are linearized inside their intervals. Furthermore, Lemma 10.13 implies:

Lemma 10.14 *Every scan operation returns a view that is the actual value of the atomic snapshot object at the linearization point of the scan.*

An update operation by p_i is linearized at the same point in the execution as its embedded write to *Segment*[i]. By Lemma 10.14, data values returned by a scan

operation are simultaneously held in all the registers at the linearization point of the operation. Therefore, each scan operation returns the value for the ith segment written by the latest update operation by p_i that precedes it in the linearization.

This completes the proof of linearizability, and leaves only the wait-free requirement. Each unsuccessful double collect by p_i can be attributed to some j, for which the condition in Line 15 holds. By the pigeonhole principle, in $2n + 1$ unsuccessful double collects three are attributed to the same j. Two of these double collects will increment $shook_i[j]$ to 1 and then 2 (in Line 17), and the third will borrow an indirect view from p_j (in Line 16).

Hence scan operations are wait-free, because the tryHS and checkHS procedures are wait-free. This, in turn, implies that update operations are wait-free. The same argument shows that each high-level operation performs $O(n^2)$ low-level operations.

Theorem 10.15 *There exists a wait-free simulation of an atomic snapshot object using read/write registers.*

10.4 SIMULATING SHARED REGISTERS IN MESSAGE-PASSING SYSTEMS

The last simulation we present in this chapter shows how to take algorithms designed for shared memory systems and run them in asynchronous message-passing systems. If we are not concerned with failures or slowdowns of processors, then the methods of Chapter 9 can be used to simulate read/write registers (e.g., the algorithm of Section 9.3.1). However, these methods are based on waiting for acknowledgments (in the underlying total broadcast algorithm) and hence are not resilient to failures.

We describe a simulation of a single-reader single-writer read/write register by n processors, in the presence of f failures, where $f < n/2$. The simulated register satisfies the definition of f-resilient shared memory. The simulation can be replicated to provide multiple shared registers.

When there are failures in a message-passing system, a processor cannot be sure that another processor will receive a message sent to it. Moreover, it cannot wait for an acknowledgment from the receiver, because the receiver may fail. To provide tolerance to $f < n/2$ failures, we use all the processors, not just the designated reader and writer of the register, as extra storage to help with the simulation of each shared register. In order to be able to pick the latest value among those stored at various processors, the values are accompanied by a sequence number.

Consider a particular simulated register. When the writer wants to write a value to this register, it increments a local counter to produce a sequence number and sends a $\langle newval \rangle$ message containing the new value and the sequence number to all the processors. Each recipient updates its local copy of the register with the information, if the sequence number is larger than what it currently has; in any event, the recipient sends back an $\langle ack \rangle$ message. Once the writer receives an $\langle ack \rangle$ from at least $\lfloor \frac{n}{2} \rfloor + 1$ processors, it finishes the write. Because $n > 2f$, it follows that $\lfloor \frac{n}{2} \rfloor + 1 < n - f$, and thus, the writer is guaranteed to receive responses from at least that many processors.

When the reader wants to read the register, it sends a ⟨*request*⟩ message to all the processors. Each recipient sends back a ⟨*value*⟩ message containing the information it currently has. Once the reader receives a ⟨*value*⟩ message from at least $\lfloor \frac{n}{2} \rfloor + 1$ processors, it returns the value with the largest sequence number among those received.

Each read and write operation communicates with a set of at least $\lfloor \frac{n}{2} \rfloor + 1$ processors, which is a majority of the processors. Thus there is at least one processor in the intersection of the subsets for each read and write operation. This intersection property guarantees that the latest value written will be obtained by each read.

To overcome potential problems caused by the asynchrony of the underlying message system, both the reader and writer maintain a local array $status[0..n-1]$, to manage the communication; $status[j]$ contains one of the following values:

not_sent: The message for the most recent operation has not yet been sent to p_j (because p_j has not acknowledged a previous message).

not_acked: The message for the most recent operation has been sent to p_j but not yet acknowledged by p_j.

acked: The message for the most recent operation has been acknowledged by p_j.

In addition, an integer counter, *num_acks*, counts the number of responses received so far during the current operation.

The pseudocode for the writer and reader appear as Algorithms 31 and 32, respectively; the code common to all processors appears as Algorithm 33. To avoid cluttering the code, the register name is not explicitly included; however, we do need a separate copy of all the local variables for each simulated register, and the register name should be a parameter to the high-level event names.

To prove the correctness of the algorithm, consider an arbitrary admissible execution of the algorithm. The next two lemmas show that the *status* variables ensure that the responses received were indeed sent in reply to the message sent during the current read or write operation.

Lemma 10.16 *When a write operation completes, at least* $\lfloor \frac{n}{2} \rfloor + 1$ *processors store a value in their copy of the variable last whose sequence number equals the sequence number of the write.*

Proof. Choose any processor p_j. The following facts about the writer p_w's $status[j]$ variable can be verified by induction.

- If $status[j]$ equals *acked*, then no message is in transit from p_w to p_j or vice versa.
- If $status[j]$ equals *not_acked*, then exactly one message is in transit from p_w to p_j or vice versa, either ⟨*newval*⟩ or ⟨*ack*⟩. Furthermore, if it is ⟨*newval*⟩, then the data in the message is for p_w's most recent write.

Algorithm 31 Read/write register simulation by message passing: code for processor p_w, the (unique) writer of the simulated register.

Initially *status*[j] equals *acked* for all j, *seq* equals 0, and *pending* equals *false*

```
1:  when write_w(v) occurs:                    // processor p_w writes v to the register
2:      pending := true
3:      seq := seq + 1
4:      num_acks := 0
5:      for all j do
6:          if status[j] = acked then          // got response for previous operation
7:              enable send_w ⟨newval, (v, seq)⟩ to p_j
8:              status[j] := not_acked
9:          else status[j] := not_sent

10: when ⟨ack⟩ is received from p_j:
11:     if not pending then status[j] := acked          // no operation in progress
12:     else if status[j] = not_sent then               // response to a previous message
13:         enable send_w ⟨newval, (v, seq)⟩ to p_j
14:         status[j] := not_acked
15:     else                                            // response for the current operation
16:         status[j] := acked
17:         num_acks := num_acks + 1
18:         if num_acks ≥ ⌊n/2⌋ + 1 then                // acks received from majority
19:             pending := false
20:             enable ack_w
```

- If *status*[j] equals *not_sent*, then exactly one message is in transit from p_w to p_j or vice versa, either ⟨*newval*⟩ or ⟨*ack*⟩. Furthermore, if it is ⟨*newval*⟩, then the data in the message is for a write prior to the most recent one.

Then *num_acks* correctly counts the number of acks received for the current write, and each one indicates that the sending processor has set its variable *last* to the data for the current write. □

Lemma 10.17 can be proved in a very similar way (see Exercise 10.8).

Lemma 10.17 *When a read operation completes, the reader has received a* ⟨*value*⟩ *message from at least* $\lfloor \frac{n}{2} \rfloor + 1$ *processors containing the value of the sender's variable last at some point since the read began.*

Lemma 10.18 deals with the ordering of the value returned by a read operation and the value written by a write operation that completely precedes it.

Lemma 10.18 *If write operation w completes before read operation r starts, then the sequence number assigned to w (in Line 3) is less than or equal to the sequence number associated with the value returned by r (cf. Line 15).*

Algorithm 32 Read/write register simulation by message passing: code for processor p_r, the (unique) reader of the simulated register.

Initially *status*[j] equals *acked* for all j, *seq* equals 0, and *pending* equals *false*

```
1:  when read_r() occurs:                          // processor p_r reads from the register
2:      pending := true
3:      num_acks := 0
4:      for all j do
5:          if status[j] = acked then              // got response for previous operation
6:              enable send_r ⟨request⟩ to p_j
7:              status[j] := not_acked
8:          else status[j] := not_sent

9:  when ⟨value,(v, s)⟩ is received from p_j:
10:     if not pending then status[j] := acked     // no operation in progress
11:     else if status[j] = not_sent then          // response to a previous message
12:         enable send_r ⟨request⟩ to p_j
13:         status[j] := not_acked
14:     else                                       // response for the current operation
15:         if s > seq then { val := v; seq := s } // more recent value
16:         status[j] := acked
17:         num_acks := num_acks + 1
18:         if num_acks ≥ ⌊n/2⌋ + 1 then           // acks received from majority
19:             pending := false
20:             enable return_r(val)
```

Proof. Let x be the sequence number of w. By Lemma 10.16, when w ends, at least $\lfloor \frac{n}{2} \rfloor + 1$ processors store data in their copies of the variable *last* with sequence number x. Call this set of processors S_w. By Lemma 10.17, when r ends, it has received at least $\lfloor \frac{n}{2} \rfloor + 1$ messages from processors containing the values of their copies of the variable *last* at some point since r began. Call this set of processors S_r. Since n is the total number of processors, S_w and S_r have a nonempty intersection. See Figure 10.11.

Let p_j be some processor in their intersection. Since the sequence number in p_j's variable $last_j$ never decreases, the sequence number reported by p_j to the read r is at least x, and the lemma follows. □

Obviously, each read returns a value written by some write that began before the read ended. Since the sequence numbers of writes are strictly increasing, we have:

Lemma 10.19 *If write operation w starts after the completion of read operation r, then the sequence number assigned to w is greater than the sequence number associated with the value returned by r.*

Finally, we can prove (see Exercise 10.9):

Algorithm 33 Read/write register simulation by message passing: code for every processor p_i, $0 \leq i \leq n-1$, including p_w and p_r.

Initially *last* equals $(v_0, 0)$, where v_0 is the initial value of the simulated register

```
1:  when ⟨newval,(v, s)⟩ is received from p_w:
                                        // v is more recent than current value
2:      last := (v, s)
3:      enable send_i ⟨ack⟩ to p_w

4:  when ⟨request⟩ is received from p_r:
5:      enable send_i ⟨value,last⟩ to p_r
```

Lemma 10.20 *If read operation r_1 occurs before read operation r_2, then the sequence number associated with the value returned by r_1 is less than or equal to the sequence number associated with the value returned by r_2.*

These three lemmas prove that the algorithm is correct (see Exercise 10.10). Each invocation of a read or write operation requires sending one message to a majority of the processors and receiving their responses. Clearly, at most $2n$ messages are sent as the result of each invocation of an operation. Moreover, if each message takes at most one time unit, then the existence of a nonfaulty majority implies that the time complexity is $O(1)$.

Theorem 10.21 *If $f < n/2$, then there exists a simulation of a single-reader single-writer read/write register for n processors using asynchronous message passing, in the presence of f crash failures. Each register operation requires $O(n)$ messages and $O(1)$ time.*

What happens if we have multiple readers? It can be shown that new-old inversions, similar to those described in Section 10.2.2, can occur when there are concurrent read operations (Exercise 10.11). One solution is to employ Algorithm 27 (see Exercise 10.12). However, this requires another layer of sequence numbers and increases the time and message complexity.

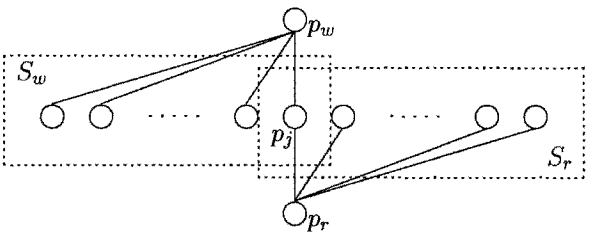

Fig. 10.11 Illustration for the proof of Lemma 10.18.

An alternative, more optimized, approach is to piggyback on the sequence numbers already used in our algorithm. The only modification we need to make is to have a reader communicate the value it is about to return to a majority of the processors (in a manner similar to the algorithm performed by the writer). This way, any read operation that starts later will observe a larger timestamp and return a later value.

Once we have multi-reader registers, we can then use Algorithm 28 to build multi-writer registers in the message passing model. We can also simulate atomic snapshot objects in message passing.

We conclude this section by proving that the requirement that $f < n/2$ is necessary, that is, if more than half the processors may fail then read/write registers cannot be simulated in message passing.

Theorem 10.22 *In any simulation of a single-reader single-writer read/write register for n processors using asynchronous message passing, the number of failures f must be less than $n/2$.*

Proof. Suppose in contradiction there is such a simulation A with $n \leq 2f$. Partition the set of processors into two sets S_0 and S_1, with $|S_0| = \lceil n/2 \rceil$ and $|S_1| = \lfloor n/2 \rfloor$. Note that both sets have size at most f.

Consider an admissible execution α_0 of A in which the initial value of the simulated register is 0, all processors in S_0 are nonfaulty, and all processors in S_1 crash initially. Suppose processor p_0 in S_0 invokes a write operation with value 1 at time 0 and no other operations are invoked. Since the algorithm must tolerate up to f failures, p_0's write operation will eventually finish at some time t_0.

Consider a second admissible execution α_1 of A in which the initial value of the simulated register is 0, all processors in S_0 crash initially, and all processors in S_1 are nonfaulty. Suppose processor p_1 in S_1 invokes a read operation at time $t_0 + 1$ and no other operations are invoked. Since the algorithm must tolerate up to f failures, p_1's read operation will eventually finish at some time t_1 and, by linearizability, will return the value 0.

Finally, consider an admissible execution β of A that is a "merger" of α_0 and α_1. In more detail, processors in S_0 experience the same sequence of events up through time t_1 as they do in α_0, while processors in S_1 experience the same sequence of events up through time t_1 as they do in α_1. There are no failures in β, but messages that are sent between the two groups have their delivery delayed until after time t_1. Since p_1 cannot distinguish between α_1 and β, it still returns 0 for its read. However, this violates linearizability, since the latest preceding write (the write by p_0) has the value 1. □

Exercises

10.1 Expand on the critical section idea for simulating shared memory (in a non-fault-tolerant way).

10.2 Show that the two definitions of wait-free simulation discussed in Section 10.1 are equivalent.

10.3 Suppose we attempt to fix the straw man multi-reader algorithm of Section 10.2.2 without having the readers write, by having each reader read the array B twice. Show a counterexample to this algorithm.

10.4 In the proof of Theorem 10.3, show that w_{j_i} must be a write to a register in S_i, for $i = 1, 2$.

10.5 Does there exist a wait-free simulation of an n-reader register from single-reader registers in which only one reader writes, when $n > 2$?

Does there exist such a simulation for $n > 2$ readers in which only $n - 1$ readers write?

10.6 Prove Lemma 10.6.

10.7 Prove the properties in Theorem 10.10 when i and j are reversed.

10.8 Prove Lemma 10.17.

10.9 Prove Lemma 10.20.

10.10 Prove Theorem 10.21.

10.11 Construct an execution in which multiple readers run the simulation of a single-writer single-reader register in the message-passing model (Section 10.4) and experience a new-old inversion.

10.12 Show how to combine Algorithm 27 and the simulation of a single-writer single-reader register from message passing (Section 10.4) to obtain a simulation of a single-writer multi-reader read/write register in a message-passing system with f failures, $f < 2n$. What are the message and time complexities of the resulting algorithm?

10.13 Show a direct simulation of a single-writer multi-reader register from message passing extending the algorithm of Section 10.4, without using an extra layer of sequence numbers. Prove the correctness of this algorithm.

Chapter Notes

This chapter concentrated on the simulation of read/write objects from other, lower-level, read/write objects and from asynchronous message passing.

The original definitions of different types of registers based on their sharing patterns were proposed by Lamport [158, 159]. Lamport also defined weaker types of registers, called *safe* and *regular*. Loosely speaking, safe registers are guaranteed to return correct values only for read operations that do not overlap write operations;

regular registers guarantee that a read operation returns a "current" value but do not prohibit new-old inversions between reads.

The general notion of linearizability was extended to wait-free algorithms by Herlihy [134]. Algorithm 26, the simulation of the multi-valued register from binary registers, is due to Vidyasankar [257]. Algorithm 27, the unbounded multi-reader algorithm, is due to Israeli and Li [140]. Algorithm 28, the unbounded multi-writer algorithm, is due to Vitányi and Awerbuch [258].

The unbounded timestamps used in the algorithms for simulating multi-reader registers and multi-writer registers can be replaced with bounded timestamps using ideas of Dolev and Shavit [98] and Dwork and Waarts [102]; see Chapter 16 of [35] for details.

Other simulations of multi-valued registers from binary ones appear in Chaudhuri, Kosa, and Welch [76]. Bounded simulations of multi-reader registers were given by Singh, Anderson, and Gouda [243]. Li, Tromp, and Vitányi [168] presented a bounded simulation of multi-writer multi-reader registers from single-writer single-reader registers.

The atomic snapshot problem was defined using three different specification methods by Afek et al. [3], by Anderson [13], and by Aspnes and Herlihy [22]; these papers also presented simulations of atomic snapshots from registers.

The simulation we presented is based on the algorithm of Afek et al. [3]. This algorithm uses $O(n^2)$ read and write operations for each snapshot operations; the best algorithm to date, using $O(n \log n)$ operations, was given by Attiya and Rachman [31]. The handshaking bits were first used by Peterson [213] and by Lamport [159].

We have specified *single-writer* atomic snapshot objects, which allow only one processor to write to each segment; *multi-writer* snapshot objects were defined by Anderson [14], who also showed how they can be simulated using *single-writer* snapshot objects and multi-writer read/write registers.

Atomic snapshots resemble distributed snapshots, studied in Chapter 6, in requiring an instantaneous view of many system components. An atomic snapshot gives the effect of reading several memory segments at once; a distributed snapshot records the local states of several processors. One difference is in the specification: atomic snapshots order the views obtained by scanners, whereas distributed snapshots do not provide any guarantee about the consistent cuts obtained in different invocations. Another difference is in the simulations: the atomic snapshot algorithm is wait-free and can tolerate processor failures, whereas the distributed snapshot algorithm does not tolerate failures.

The simulation of read/write registers from message passing is due to Attiya, Bar-Noy, and Dolev [25]. It requires integer sequence numbers, and thus the size of messages is unbounded. To bound the message size, Attiya, Bar-Noy, and Dolev used bounded timestamps. An alternative simulation with bounded messages was given by Attiya [24]. The shared memory simulation uses a particular kind of *quorum system*, the majority quorum system, to ensure that each read obtains the value of the most recent write. This idea has been generalized to other quorum systems, including those that can tolerate Byzantine failures [179].

The simulation in Section 10.4 provides a distributed shared memory that is fault tolerant, in contrast to the distributed shared memory algorithms of Chapter 9, which were not fault tolerant. The fault tolerance provided here ensures that the data in the shared objects remains available to correct processors, despite the failures of some processors. Processors that fail are no longer of interest, and no guarantees are made about them. One-resilient algorithms for the same notion of fault-tolerant distributed shared memory were presented by Stumm and Zhou [248].

DSMs that can tolerate processor failures and recoveries have been a subject of recent interest. Many sequentially consistent DSM systems that are tolerant of processors that crash and recover have been described in the literature (e.g., [248, 262, 229, 147]). There has also been work on recoverable DSMs that support weaker consistency conditions (e.g., [143, 148, 200]). In these papers, techniques including checkpoints of consistent cuts and logging information to stable storage are employed to reintegrate recovered processors.

Distributed Computing: Fundamentals, Simulations and Advanced Topics, Second Edition
Hagit Attiya and Jennifer Welch
 ISBN: 0-471-45324-2

11

Simulating Synchrony

As we have seen in previous chapters, the possible behaviors of a synchronous system are more restricted than the possible behaviors of an asynchronous system. Thus it is easier to design and understand algorithms for synchronous systems. However, most real systems are at least somewhat asynchronous. In Chapter 6, we have seen ways to observe the causality structure of events in executions. In this chapter, we show how they can be employed to run algorithms designed under strict synchronization assumptions, in systems that provide weaker synchronization guarantees.

Throughout this chapter, we only consider message-passing systems in which there are no failures. The topology of the communication system can be arbitrary and, as always, the only (direct) communication is between nodes that are neighbors in the topology.

We begin the chapter by specifying synchronous message-passing systems in our layered model of computation.

Then we show how a small modification to the logical clocks from Chapter 6, called *logical buffering*, can be used to provide the illusion that the processors take steps in lockstep, when in reality they are totally asynchronous. Thus logical buffering is a way to simulate a system in which processors are synchronous and messages are asynchronous with a (totally) asynchronous system. Another way to view this simulation is that asynchronous processors can simulate processors with the most powerful kind of hardware clocks, those that always equal real time, as long as message delay remains asynchronous.

The second synchrony simulation we study in this chapter is one to simulate the synchronous system with the asynchronous system. Logical buffering can be used to simulate synchronous *processors*, but it does not ensure any bounds on message

delay. To create the illusion of synchronous message delay, more effort is required. A simulation from the synchronous system to the asynchronous system is called a *synchronizer*. We present a simple synchronizer in this chapter.

Both simulations are *local*, not *global*. We conclude the chapter by discussing some implications of this fact.

11.1 SYNCHRONOUS MESSAGE-PASSING SPECIFICATION

In this section, we give a problem specification of the communication system we wish to simulate, a synchronous message-passing system. The key difference from the asynchronous case is that send and recv events at different processors must be interleaved in a regular way to achieve the lockstep rounds that characterize the synchronous model.

Formally, the inputs and outputs are the same as for asynchronous message passing, defined in Chapter 7, that is, the inputs are of the form $\text{send}_i(M)$ and the outputs are of the form $\text{recv}_i(M)$, where i indicates a processor and M is a set of messages (including possibly the empty set).

As in the model used in Part I, a round for a processor consists of sending messages, receiving messages sent in that round, and performing local computation, which will determine what messages to send in the next round. Conceptually, each round occurs concurrently at all the processors. Because our notion of execution is a sequence of events, we must choose some (arbitrary) total order of events that is consistent with the partial order. There are a number of total orders that will work. We choose a particular total order here that is consistent with our requirement, from Chapter 7, that all enabled events take place on a node before any event on another node can take place. This total order is embodied in the Round-Robin property below that is placed on allowable sequences of the synchronous message passing model. According to the Round-Robin property, all the round 1 send events take place. Then the receipts of the round 1 messages occur. Because of the node atomicity requirement, when a processor receives a message, the very next event must be the send that is triggered by that receipt, which in this case is the send for round 2. Because the receive for round 1 cannot be separated from the send for round 2, we have a series of recv-send pairs, one for each processor, in which messages for round 1 are received and messages for round 2 are sent. Then we have another series of recv-send pairs, in which messages for round 2 are received and messages for round 3 are sent.

Formally, every sequence in the allowable set must conform to the following "round" structure:

Round-Robin: The sequence is infinite and consists of a series of subsequences, where the first subsequence has the form $\text{send}_0, \ldots, \text{send}_{n-1}$, and each later subsequence has the form $\text{recv}_0, \text{send}_0, \text{recv}_1, \text{send}_1, \ldots, \text{recv}_{n-1}, \text{send}_{n-1}$. The kth send_i and the kth recv_i events form *round* k for processor p_i.

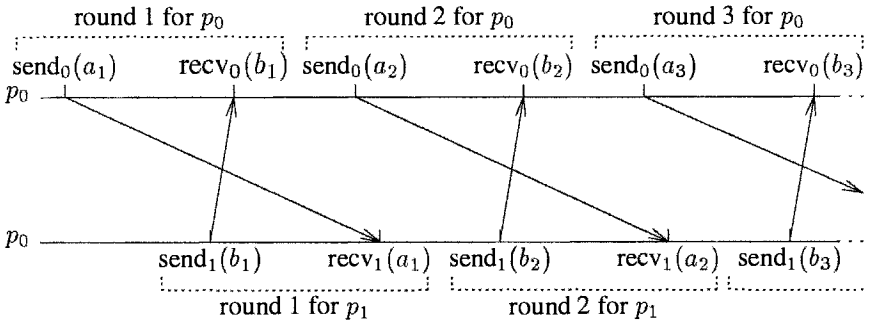

Fig. 11.1 How rounds overlap.

The Round-Robin property imposes constraints on the inputs (the send events) from the environment; they must occur in Round-Robin order and be properly interleaved with the outputs (the recv events).

As an example, consider three rounds of a system consisting of two processors p_0 and p_1, which are directly connected:

$$\text{send}_0(a_1), \text{send}_1(b_1), \text{recv}_0(b_1), \text{send}_0(a_2), \text{recv}_1(a_1), \text{send}_1(b_2),$$
$$\text{recv}_0(b_2), \text{send}_0(a_3), \text{recv}_1(a_2), \text{send}_1(b_3), \text{recv}_0(b_3), \ldots$$

(See Fig. 11.1.)

The sets of messages sent and received must satisfy the following conditions:

Integrity: Every message received by processor p_i from processor p_j in round k was sent in round k by p_j.

No Duplicates: No message is received more than once. Thus each recv event contains at most one message from each neighbor.

Liveness: Every message sent in round k is received in round k.

The environment (that is, the algorithm using the synchronous message-passing system) decides unilaterally what should be sent in the first round for each processor. In later rounds, the environment can decide based on what it has previously received.

11.2 SIMULATING SYNCHRONOUS PROCESSORS

In this section, we consider two systems with asynchronous communication (i.e., unbounded message delays), one with synchronous (lockstep) processors, called SynchP, and one with asynchronous processors, called Asynch. We will show that Asynch can locally simulate SynchP.

Asynch is the standard asynchronous message-passing communication system. SynchP is a weakening of the standard synchronous message-passing communication

Algorithm 34 Logical buffering to simulate synchronous processors: code for p_i, $0 \leq i \leq n-1$.

Initially *round* $= 0$ and *buffer* $= \emptyset$

1: when SynchP-send$_i(S)$ occurs:
2: *round* $:=$ *round* $+ 1$
3: enable Asynch-send$_i(\{\langle m, round\rangle : m \in S\})$

4: when Asynch-recv$_i(M)$ occurs:
5: add M to *buffer*
6: *ready* $:= \{m : \langle m, tag\rangle \in$ *buffer* and *tag* $\leq$ *round*$\}$
7: remove from *buffer* every element that contributed to *ready*
8: enable SynchP-recv$_i$(*ready*)

system; an allowable sequence is still an infinite series of rounds, but there is no longer the requirement that every message sent in round k be received in round k. Instead, every message sent in round k must be received in round k or later. Messages need not be delivered in FIFO order, but no message is delivered more than once.

The only situation observable to the processors that can happen in Asynch but not in SynchP is for a message that was sent at the sender's kth send step to be received at the recipient's jth recv step, where $j < k$. If this situation were to occur when processors take steps in lockstep, it would mean that the message is received before it is sent. To avoid this situation, the simulation employs *logical buffering*; each processor keeps a round counter that counts how many send steps it has taken, each message is tagged with the sender's round count, and the recipient delays processing of a message until its round count equals or exceeds that on the message.

Logical buffering provides properties similar to those of logical clocks; however, with logical clocks some logical times may be skipped, but not with logical buffering. Skipping some logical times may have drawbacks, for instance, if certain actions are scheduled to occur at certain logical times. With logical buffering, the round counts themselves are consistent with the happens-before relation.

The pseudocode is presented in Algorithm 34. SynchP-send indicates the send for system SynchP and is an input to the logical buffering algorithm. SynchP-recv indicates the receive for system SynchP and is an output of the logical buffering algorithm. Asynch-send and Asynch-recv are the communication primitives used by the logical buffering processes to communicate with each other over the asynchronous communication system. The result of the occurrence of Asynch-send$_i$ and SynchP-recv$_i$ is simply to disable that event, as explained in Section 7.6. There must be an infinite number of SynchP-send$_i$ events, for each i; however, the empty set of messages is an allowable parameter for SynchP-send$_i$.

Theorem 11.1 *System Asynch locally simulates system SynchP.*

Proof. Consider the logical buffering algorithm, Algorithm 34. Obviously, it has the correct top and bottom interfaces.

Let α be an execution of the logical buffering algorithm that is locally admissible for (SynchP,Asynch). Locally user compliant for SynchP means that each node alternates sends and receives, but the sends and receives at different nodes are not necessarily interleaved regularly as they would be in the actual SynchP system. We must show that there exists a sequence σ in *seq*(SynchP) such that $\sigma|i = top(\alpha)|i$, for all i, $0 \leq i \leq n-1$.

Define σ to be the result of taking the n sequences $top(\alpha)|i$, $0 \leq i \leq n-1$, and interleaving them so as to conform to the Round-Robin property of the synchronous system.

To verify that σ is in *seq*(SynchP), we only need to check that every message sent in some round is eventually received in the same or a later round and is not received twice. No message is delivered too early, since messages are tagged and held in buffer variables in execution α. Every message is eventually delivered, since the round counter increments without bound because of the infinite number of SynchP-send$_i$ events, and the tags do not change in α. No message is received twice since messages are removed from *buffer* once they become ready. □

Thus logical buffering can provide the illusion that processors possess more synchrony than they already do, going from totally asynchronous to lockstep. However, the level of synchrony of the communication is not improved.

11.3 SIMULATING SYNCHRONOUS PROCESSORS AND SYNCHRONOUS COMMUNICATION

Previously in this chapter, we have seen how to (locally) simulate synchronous processors with asynchronous processors. In this section we show how to (locally) simulate the fully synchronous message-passing communication system, denoted Synch, with Asynch, the fully asynchronous message-passing communication system. Such a simulation is called a *synchronizer.*

In addition to having synchronous processors, the synchronous system guarantees that in the kth round of each processor it receives all the messages that were sent to it by its neighbors in their kth round. In this section, we show how to achieve this property, by using a synchronizer and thus locally simulating the synchronous system by the asynchronous system.

The main difficulty in ensuring the above property is that a node does not usually know which of its neighbors have sent a message to it in the current round. Because there is no bound on the delay a message can incur in the asynchronous system, simply waiting long enough before generating the (synchronous) receive event does not suffice; additional messages have to be sent in order to achieve synchronization.

11.3.1 A Simple Synchronizer

In this section, we present a simple synchronizer called ALPHA. Synchronizer ALPHA is efficient in terms of time but inefficient in terms of messages; the chapter notes

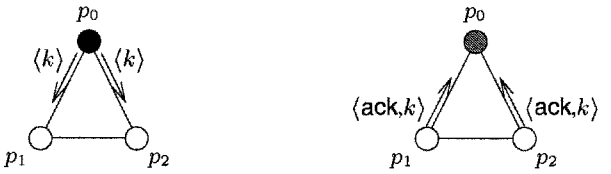

Fig. 11.2 Processor p_0 sends round k messages to its neighbors (left) and then receives $\langle ack \rangle$ messages (right).

discuss another synchronizer that provides a trade-off between time complexity and message complexity.

Before generating the receive for round k, the synchronizer at a node p_i must know that it has received all the round k messages that were sent to this node. The idea is to have the neighbors of p_i check whether all their messages were received and have them notify p_i. It is simple for a node to know whether all its messages were received, if we require each node to send an acknowledgment for every message (of the original synchronous algorithm) received. If all round k messages sent by a node have been acknowledged, then the node is said to be *safe* for round k. Observe that the acknowledgments only double the number of messages sent by the original algorithm. Also observe that each node detects that it is safe for round k a constant time after it generates the synchronous send for round k (using the method of measuring time complexity for asynchronous algorithms from Chapter 2).

Figure 11.2 presents a simple execution in which p_0 sends messages to its neighbors and receives acknowledgments; at this point, p_0 is safe. In this figure and Figure 11.3, a black node represents a processor just starting a round and a gray node represents a safe processor; otherwise, the node is white.

A node can generate its next synchronous receive once all its neighbors are safe. Most synchronizers use the same mechanism of acknowledgments to detect that nodes are safe; they differ in the mechanism by which nodes notify their neighbors that they are safe. In synchronizer ALPHA, a safe node directly informs all its neighbors by sending them messages. When node p_i has been informed that all its neighbors are safe for round k, p_i knows that all round k messages sent to it have arrived, and it generates the synchronous receive for round k.

Figure 11.3 presents the extension of the execution in Figure 11.2: p_0 and p_2 indicate they are safe, allowing p_1 to do its round k receive and the move on to send its round $k+1$ messages.

The pseudocode appears in Algorithm 35. Synch-send and Synch-recv indicate the events of the synchronous system, whereas Asynch-send and Asynch-recv indicate those of the asynchronous system. This version of the code uses unbounded space for simplicity. An exercise is to reduce the space usage (Exercise 11.3). As we did in Chapter 2, we describe the algorithm as if each message of the underlying asynchronous system is received separately. Similar arguments justify this simplification.

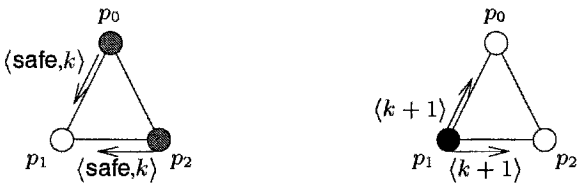

Fig. 11.3 Processors p_0 and p_2 send ⟨safe⟩ messages for round k (left) and then p_1 sends round $k+1$ message (right).

Theorem 11.2 *Asynch locally simulates Synch.*

Proof. Consider synchronizer ALPHA of Algorithm 35. Obviously, it has the correct top and bottom interfaces.

Let α be any execution of synchronizer ALPHA that is (Synch,Asynch)-locally-admissible. We must show that there exists a sequence σ in *seq*(Synch) such that $\sigma|i = top(\alpha)|i$, for all i, $0 \leq i \leq n-1$.

Define σ to be the result of taking the n sequences $top(\alpha)|i$, $0 \leq i \leq n-1$, and interleaving them so as to conform to the Round-Robin property of the synchronous system.

To verify that σ is in *seq*(Synch), we prove three lemmas. Lemma 11.3 states that the Synch-recv events at p_j are done in the correct order, that is, that if p_j performs Synch-recv for round r, then the previous Synch-recv performed by p_j was for round $r-1$. Lemma 11.4 states that if processor p_j performs Synch-recv for round r, then *buffer*$[r]$ consists exactly of all round r messages sent to p_j. Lemma 11.5 states that p_j performs an infinite number of Synch-recv events.

The proof of Lemma 11.3 is left as an exercise (Exercise 11.4).

Lemma 11.3 *If p_j performs Synch-recv for round r in α, then the previous Synch-recv performed by p_j is for round $r-1$.*

Lemma 11.4 *If p_j performs Synch-recv for round r, then buffer$[r]$ consists of all round r messages sent to p_j in α.*

Proof. The use of the round tag and the separate variables for each round ensure that no extraneous messages are in *buffer*$[r]$.

Let m be a message Synch-sent by p_i to p_j in round r. After the Synch-send, p_i Asynch-sends m to p_j. On its arrival, p_j puts it in *buffer*$[r]$. Suppose in contradiction p_j has already done Synch-recv for round r. Then *safe*$_j[r]$ must have included i at the time when p_j performed the Synch-recv. So p_j received a ⟨safe,r⟩ message from p_i. But then p_i would have already received an ⟨ack,r⟩ message from p_j for this message, which it has not sent yet, a contradiction. □

Lemma 11.5 *Each processor performs a Synch-recv event for round r, for every $r \geq 1$.*

Algorithm 35 Simple synchronizer ALPHA with unbounded space: code for p_i, $0 \leq i \leq n-1$.

Initially *round* $= 0$ and
buffer$[r]$, *safe*$[r]$, and *ack-missing*$[r]$ are empty, for all $r \geq 1$

```
1: when Synch-send_i(S) occurs:
2:   round := round + 1
3:   ack-missing[round] := {j : p_j is a recipient of a message in S}
4:   enable Asynch-send_i(⟨m,round⟩) to p_j, for each m ∈ S with recipient p_j

5: when Asynch-recv_i(⟨m,r⟩) from p_j occurs:
6:   add (m, j) to buffer[r]
7:   enable Asynch-send_i(⟨ack,r⟩) to p_j                    // acknowledge

8: when Asynch-recv_i(⟨ack,r⟩) from p_j occurs:
9:   remove j from ack-missing[r]
10:  if ack-missing[r] = ∅ then          // all neighbors have acknowledged
11:      enable Asynch-send_i(⟨safe,r⟩) to all neighbors          // p_i is safe

12: when Asynch-recv_i(⟨safe,r⟩) from p_j occurs:
13:  add j to safe[r]
14:  if safe[r] includes all neighbors then          // all neighbors are safe
15:      enable Synch-recv_i(buffer[r])                          // for round r
```

Proof. Suppose in contradiction that the lemma is false. Let r be the smallest round number such that some processor p_i never performs Synch-recv for round r.

First we argue that Synch-send for round r occurs at every node. If $r = 1$, then the Synch-sends occur because α is locally user compliant for Synch. If $r > 1$, then every processor performs Synch-recv for round $r-1$ by choice of r, and so Synch-send for round r occurs at every node because α is locally user compliant for Synch.

Thus every neighbor p_j of p_i experiences Synch-send for round r. Then p_j does Asynch-send to the appropriate neighbors, gets back the ⟨ack⟩ messages, becomes safe, and sends ⟨safe⟩ messages to all neighbors, including p_i.

Thus p_i receives ⟨safe⟩ messages from all neighbors and performs Synch-recv for round r, a contradiction. □

These three lemmas show that Asynch locally simulates Synch using ALPHA. □

The total complexity of the resulting algorithm depends on the overhead introduced by the synchronizer and, of course, on the time and message complexity of the synchronous algorithm.

Clearly, the time overhead of ALPHA is $O(1)$ per round. Because two additional messages per edge are sent (one in each direction), the message overhead of ALPHA

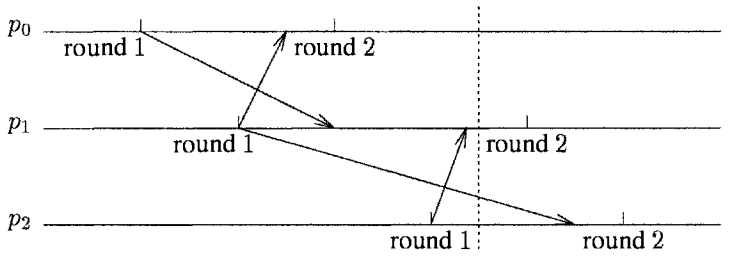

Fig. 11.4 Example execution of a synchronizer.

is $O(|E|)$ per round. Note that $O(|E|)$ messages are sent per round, regardless of the number of original messages sent during this round. The chapter notes discuss another synchronizer that exhibits a tradeoff between time and message overhead.

11.3.2 Application: Constructing a Breadth-First Search Tree

To see the usefulness of a synchronizer, we consider a problem that is much easier to solve in a synchronous system than an asynchronous one—construction of a breadth-first search (BFS) tree of a network (with arbitrary topology). Recall that the modified flooding algorithm of Chapter 2 (Algorithm 2) solves the BFS tree problem in synchronous systems, given a particular node to serve as the root of the tree. On the other hand, in an asynchronous system, the spanning tree constructed by the algorithm is not necessarily a BFS tree.

This problem is one of the prime examples for the utility of the synchronizer concept; in fact, the most efficient known asynchronous solutions for this problem were achieved by applying a synchronizer to a synchronous algorithm.

Consider now an execution of the synchronous BFS tree algorithm (Algorithm 2), on top of synchronizer ALPHA. The composition forms a BFS tree algorithm for the asynchronous case. Because its time complexity in the synchronous system is $O(D)$, where D is the diameter of the communication graph, it follows that in the asynchronous system, its time complexity is $O(D)$ and its message complexity is $O(D \cdot |E|)$.

11.4 LOCAL VS. GLOBAL SIMULATIONS

Consider an asynchronous message-passing system in which three nodes are arranged in a chain: There is a link from p_0 to p_1 and from p_1 to p_2. There is an execution of the synchronizer in this system in which the following events take place (see Fig. 11.4):

- p_0 simulates round 1;
- p_1 simulates round 1;

- p_0 simulates round 2, receiving p_1's round 1 message;
- p_2 simulates round 1;
- p_1 simulates round 2, receiving p_0's and p_2's round 1 messages;
- p_2 simulates round 2, receiving p_1's round 1 message.

Suppose the synchronous algorithm running on top of the synchronizer solves the session problem from Chapter 6 in the synchronous system, for two sessions, by having each processor perform special actions at rounds 1 and 2. Unfortunately, the transformed algorithm is not correct, because there is only one session in the execution described above, not two.

This scenario indicates the limitations of local simulations as opposed to global simulations. In the synchronous system, the rounds of the processors are guaranteed to be correctly interleaved so as to achieve two sessions. But the synchronizer, although it mimics the same execution on a per-node basis, cannot achieve the same interleaving of events at different nodes.

Informally speaking, local simulations preserve correctness for *internal* problems, those whose specifications do not depend on the real time at which events occur. The existence of a synchronizer implies that there is no difference in what can be computed in the synchronous and asynchronous systems, as long as we restrict our attention to internal problems in the absence of failures.

The lower bound on the running time for the session problem presented in Chapter 6 implies that any simulation of the synchronous system by the asynchronous system that preserves the relative order of events across nodes will require time overhead of a factor of the diameter of the communication network, roughly. The reason is that such a general simulation would transform the synchronous algorithm for the session problem into an algorithm for the asynchronous system, which by this lower bound must incur the stated overhead.

Exercises

11.1 Does logical buffering work for simulating system SynchP by system AsynchP in the presence of crash failures? If so, why? If not, then modify it to do so. What about Byzantine failures?

11.2 Is wait-free consensus possible in the system SynchP? What if there is at most one failure?

Hint: Think about Exercise 11.1.

11.3 Show how to bound the space complexity of synchronizer ALPHA.

11.4 Prove Lemma 11.3.

11.5 What are the worst-case time and message complexities of the asynchronous BFS tree algorithm that results from applying synchronizer ALPHA? What network topology exhibits this worst-case behavior?

11.6 Apply synchronizer ALPHA to both of the synchronous leader election algorithms in Chapter 3. What are the resulting time and message complexities? How do they compare to the lower bounds for asynchronous leader election?

11.7 Is mutual exclusion an internal problem?

11.8 If a specification is not internal, does that mean it cannot be implemented in an asynchronous system?

Chapter Notes

This chapter described two methods for simulating synchrony: logical buffering and synchronizers.

Logical buffering was independently developed by Neiger and Toueg [199] and by Welch [259]. Neiger and Toueg's paper defined the class of internal problems for which the translation is valid and applied it to systems with synchronized clocks, as well as other situations. Welch's paper gave fault-tolerant implementations of logical buffering in the asynchronous case. The latter paper contains the answer to Exercise 11.1, as well as to Exercise 11.2. An alternative answer to Exercise 11.2 can be found in Dolev, Dwork, and Stockmeyer [92].

Synchronizers were introduced by Awerbuch [36], who also suggested ALPHA (Algorithm 35) and the application to BFS tree construction; additional applications appear in [38]. Awerbuch [36] also describes two other synchronizers, BETA, with low message overhead, and GAMMA, which provides a trade-off between message and time overhead. Chapter 18 of [35] describes synchronizer ZETA, having the same trade-off as GAMMA.

Awerbuch [36] proved that the trade-off given by GAMMA is essentially optimal, if the synchronizer must simulate one round after the other. Some improved synchronizers have been suggested under various restrictions, for example on the network topology (Peleg and Ullman [209]). Awerbuch and Peleg [40] showed that if a synchronizer is not required to work in a round-by-round manner then the above trade-off is not inherent. Peleg's book [208] includes a thorough discussion of these topics.

Distributed Computing: Fundamentals, Simulations and Advanced Topics, Second Edition
Hagit Attiya and Jennifer Welch
 ISBN: 0-471-45324-2

12

Improving the Fault Tolerance of Algorithms

In Chapter 11, we saw examples of simulating a more well-behaved situation (namely, a synchronous system) with a less well-behaved situation (namely, an asynchronous system). The advantage of such simulations is that often algorithms can be developed for the more well-behaved case with less effort and complexity and then automatically translated to work in the less well-behaved case.

The same idea can be applied in the realm of fault tolerance. As we have seen in Chapter 5, more benign types of faults, such as crash failures, are easier to handle than more severe types of faults, such as Byzantine failures. In particular, more benign types of faults can often be tolerated with simpler algorithms. In this chapter, we explore methods that automatically translate algorithms designed to tolerate more benign faults into algorithms that can tolerate more severe faults. This chapter deals only with message-passing systems, because usually only crash failures are considered in the context of shared memory systems. We also extend the layered model of computation to handle synchronous processors.

12.1 OVERVIEW

The bulk of this chapter presents a simulation that makes Byzantine failures appear to be crash failures in the synchronous case. Although it is possible to achieve this in one step (see the notes at the end of the chapter), it is conceptually easier to break this task down into subtasks, each one focusing on masking a particular aspect of Byzantine failures.

There are three problematic aspects of Byzantine behavior. The first is that a faulty processor can send messages with different contents to different processors in the same round. The second is that a faulty processor can send messages with arbitrary content (even if it sends the same message to all processors). The third is that this type of bad behavior can persist over many rounds.

Our approach to designing the overall simulation is to tackle each problem in isolation, designing three fault model simulations. The first simulation is from Byzantine to "identical Byzantine" (like Byzantine, but where faulty processors are restricted so that each one sends either nothing or the same message to all processors at each round). The second simulation is from identical Byzantine to "omission" (faulty processors send messages with correct contents but may fail to receive or send some messages). The third simulation is from omission to crash. More precise definitions of the communication system in the presence of these types of faults are given below.

In going from Byzantine to identical Byzantine, we must prevent faulty processors from sending conflicting messages. This can be done by having processors echo to each other the messages received over the lower level and then receiving at the higher level only those messages for which sufficiently many echoes were received. By setting this threshold high enough and assuming enough processors are nonfaulty, this scheme will ensure that no conflicting messages are received.

In going from identical Byzantine to omission, the contents of the messages received must be validated as being consistent with the high level algorithm A being run on top of the omission system. This is achieved by sending along with each message of the algorithm A the set of messages received by the algorithm at the previous simulated round, as justification for the sending of this message.

In going from omission to crash, once a processor exhibits a failure, we must ensure that it takes no more steps. This is accomplished by having processors echo to each other the messages received over the lower level. If a processor does not receive enough echoes of its own message, then it halts (crashes itself).

However, there is a problem with this approach. In order to validate messages in going from identical Byzantine to omission, a processor must have more information about messages sent by faulty processors than that described above. It is possible that a faulty processor p_k's message at some round will be received at one processor, p_i, but not at another, p_j. However, for p_j to correctly validate later messages from p_i, it needs to know that p_i received p_k's message.

As a result, the interface of the identical Byzantine model is slightly different than the other interfaces. Each message includes a round tag, meaning that the message was originally sent in the round indicated by the tag. The identical Byzantine model guarantees that if one nonfaulty processor receives a message from a faulty processor, then *eventually* every nonfaulty processor receives that message, although not necessarily in the same round.

After describing the modifications to the formal model in Section 12.2, we start in Section 12.3 with a simulation of the identical Byzantine model on top of Byzantine failures (Fig. 12.1(a)). This simulation masks inconsistent messages of faulty processors, but doubles the number of rounds.

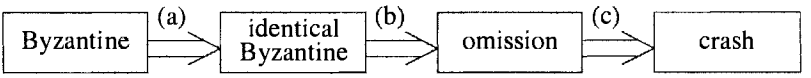

Fig. 12.1 Summary of simulations for the synchronous case.

Next, in Section 12.4 we show a simulation of omission failures on top of the identical Byzantine model (Fig. 12.1(b)). This simulation is based on validating the messages of faulty processors to make sure they are consistent with the application program; it has no overhead in rounds.

Finally, in Section 12.5 we show a simulation of crash failures on top of omission failures (Fig. 12.1(c)). This simulation is based on having processors crash themselves if they perform an omission failure; it doubles the number of rounds.

Throughout this chapter, f is the upper bound on the number of faulty processors.

The faulty behavior in all cases is pushed into the communication system. Thus in the formal model, the processes always change state correctly (according to what information they receive from the communication system). The communication system decides to stop delivering some processor's message, or to corrupt the contents, etc. (Exercise 12.2 asks you to prove that this model is equivalent to the previous definition.)

We assume that the topology of the message-passing system is fully connected, and that each processor sends the same message to all the processors at each round, that is, the processor is actually doing a broadcast.

When we consider a particular simulation from one model to another, we sometimes call the high level send event *broadcast* and the high-level receive event *accept*, to distinguish them from the low-level send and receive events.

12.2 MODELING SYNCHRONOUS PROCESSORS AND BYZANTINE FAILURES

In Chapter 11, we gave a specification of a synchronous message-passing communication system and showed how to locally simulate it with asynchronous systems. We now consider the other side of the coin: how to *use* a synchronous message-passing communication system to solve some other problem. In this section, we describe restrictions on the behavior of synchronous processors in our layered model.

In the synchronous message-passing system, we impose a more structured requirement on the interaction of processes at a node. The transition function of each process must satisfy the following: When an input from the layer below occurs at a process, at most one output is enabled, either to the layer above or the layer below. When an input from the layer above occurs at a process, at most one output is enabled and it must be to the layer below. The resulting behavior of all the processes on a node is that activity propagates down the process stack, or propagates up the process stack, or propagates up the stack and turns around and propagates down. The reason for the

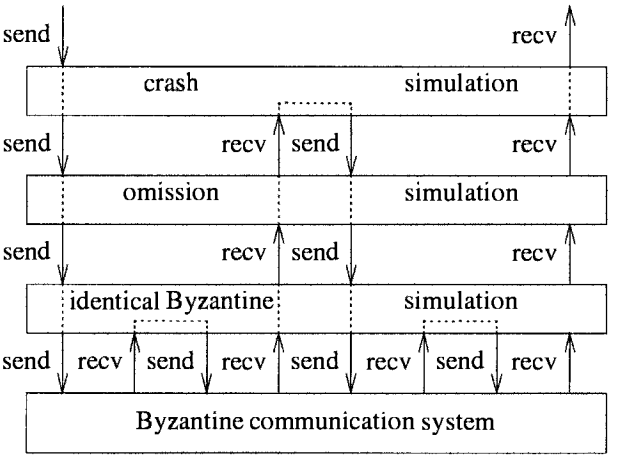

Fig. 12.2 Sample execution of all layers at one node; a single round of the environment with Byzantine failures is two rounds of the crash failure simulation and the omission failure simulation and is four rounds of the identical Byzantine failures simulation.

asymmetry is that we want to model the response to receiving a set of messages as happening atomically with the receipt, but the response to an environment input will have to wait (generally, for some communication).

In the synchronous message-passing model, in addition to the four conditions specified at the end of Section 7.3, an execution must satisfy the following:

5. The first event at each node is an input from the environment. Each output to the environment at a node, except for the last one for that node, is immediately followed by an input from the environment at the same node.

This condition places restrictions on the behavior of the environment and ensures that environment events and communication system events are interleaved properly.

Figure 12.2 shows node steps and rounds in the fault tolerance simulations; in the figure, dotted lines link together events that happen atomically at the node.

In pseudocode descriptions of algorithms for synchronous message-passing systems, the order in which all the events triggered by a node input occur must be specified carefully. In this case, the triggered events must conform to a specific pattern (e.g., going up from the bottom layer, then turning around and coming back down). Instead of stating that output X has become enabled (and being rather loose about when it should occur), we will have a line of code with the name of the output, meaning that the output must actually occur before proceeding.

Next, we define the synchronous message-passing model subject to Byzantine processor failures.

The inputs are of the form Byz-send$_i(m)$ and the outputs are of the form Byz-recv$_i(M)$, where i indicates a processor, m is a message (to be sent to all the processors), and M is a set of messages, at most one coming from each processor.

A sequence is in the allowable set if it conforms to the standard synchronous round structure described in Chapter 7.

Furthermore, there must be a partitioning of the processor ids into faulty and nonfaulty, with at most f faulty, satisfying the following conditions (here and in the rest of the chapter, we assume that variables range over the appropriate values):

Nonfaulty Integrity: If nonfaulty processor p_i receives message m from nonfaulty processor p_j in round k, then p_j sends m in round k.

Nonfaulty Liveness: If nonfaulty processor p_i sends m in round k, then nonfaulty processor p_j receives m from p_i in round k.

Note that there are no requirements on the messages received from or by faulty processors.

When there is the potential for confusion, "send" is replaced with "Byz-send" and "recv" with "Byz-recv."

12.3 SIMULATING IDENTICAL BYZANTINE FAILURES ON TOP OF BYZANTINE FAILURES

In this section we specify a communication system model called *identical Byzantine* that restricts the power of Byzantine processors. The basic idea is that faulty processors can still send arbitrary messages, but all processors that receive a message from a faulty processor receive the same message. We start with a precise definition of the identical Byzantine fault model and then describe how to simulate it in the presence of (unrestricted) Byzantine failures.

12.3.1 Definition of Identical Byzantine

The formal definition of a synchronous message-passing communication system subject to identical Byzantine failures is the same as for (unrestricted) Byzantine failures with these changes. Each message received has the format (m, k), where m is the content of the message and k is a positive integer (the *round tag*); and the conditions to be satisfied are the following:

Nonfaulty Integrity: If nonfaulty processor p_i receives (m, k) from nonfaulty processor p_j, then p_j sends m in round k.

Faulty Integrity (Identical Contents): If nonfaulty processor p_i receives (m, k) from p_h and nonfaulty processor p_j receives (m', k) from p_h, then $m = m'$.

No Duplicates: Nonfaulty processor p_i receives only one message with tag k from p_j.

Nonfaulty Liveness: If nonfaulty processor p_i sends m in round k, then nonfaulty processor p_j receives (m, k) in round k.

Faulty Liveness (Relay): If nonfaulty processor p_i receives (m, k) from (faulty) processor p_h in round r, then nonfaulty processor p_j receives (m, k) from p_h by round $r + 1$.

When there is the potential for confusion, "send" is replaced with "id-send" and "recv" with "id-recv."

12.3.2 Simulating Identical Byzantine

We now present a simulation of the identical Byzantine failure model in the Byzantine failure model. Intuitively, to successfully broadcast a message, a processor has to obtain a set of "witnesses" for this message. A nonfaulty processor accepts a message only when it knows that there are enough witnesses for this broadcast. The simulation uses two rounds of the underlying Byzantine system to simulate each round of the identical Byzantine system and requires that $n > 4f$.

Round k of the identical Byzantine system is simulated by rounds $2k - 1$ and $2k$ of the underlying Byzantine system, which are also denoted $(k, 1)$ and $(k, 2)$.

To broadcast m in simulated round k, the sender, p_i, sends a message $\langle \mathsf{init}, m, k \rangle$ to all[1] processors in round $(k, 1)$. When a processor receives the first init message of p_i for round k, it acts as witness for this broadcast and sends a message $\langle \mathsf{echo}, m, k, i \rangle$ to all processors in round $(k, 2)$. When a processor receives $n - 2f$ echo messages in a single round, it becomes a witness to the broadcast and sends its own echo message to all processors; at this point, it knows that at least one nonfaulty processor is already a witness to this message. When a processor receives $n - f$ echo messages in a single round, it accepts that message, if it has not already accepted a message from p_i for simulated round k.

The pseudocode appears in Algorithm 36. The function first-accept(k', j) returns true if and only if the processor has not already accepted a message from p_j for round k'.

We now prove that this algorithm simulates identical Byzantine failures on top of Byzantine failures, according to the definition of (global) simulation in Chapter 7.

Theorem 12.1 *In any execution of Algorithm 36 that is admissible (i.e., fair, user compliant for the identical Byzantine specification, and correct for the Byzantine communication system), the five conditions defining identical Byzantine are satisfied.*

Proof. Since α is correct for the Byzantine communication system, $bot(\alpha)$ is in the allowable set of sequences for that specification. In particular, there is a partitioning of the processor ids into faulty and nonfaulty, with at most f faulty. Throughout this proof, this partitioning is what defines which processors are faulty and which are nonfaulty.

Nonfaulty Integrity: Suppose nonfaulty processor p_i accepts (m, k) from nonfaulty processor p_j. Then (m, k) with sender p_j is in p_i's accepted set in round

[1] Throughout this chapter, this means including itself.

Algorithm 36 Simulating round $k \geq 1$ of the identical Byzantine fault model: code for processor p_i, $0 \leq i \leq n-1$.

Initially $S = \emptyset$ and *accepted* $= \emptyset$

```
1:  round (k, 1): in response to id-send_i(m):
2:      Byz-send_i(S ∪ {⟨init,m,k⟩})
3:      Byz-recv_i(R)
4:      S := { ⟨echo,m',k,j⟩ : there is a single ⟨init,m',k⟩ in R with sender p_j }
5:      S := S ∪ { ⟨echo,m',k',j⟩ : k' < k and
            m' is the only message for which at least n − 2f
            ⟨echo,m',k',j⟩ messages have been received in this round }
6:      accepted := {(m', k') with sender p_j : at least n − f ⟨echo,m',k',j⟩
            messages have been received in this round and first-accept(k', j)}

7:  round (k, 2):
8:      Byz-send_i(S)
9:      Byz-recv_i(R)
10:     S := { ⟨echo,m',k',j⟩ : k' ≤ k and
            m' is the only message for which at least n − 2f
            ⟨echo,m',k',j⟩ messages have been received in this round }
11:     accepted := accepted ∪{(m', k') with sender p_j : at least n − f
            ⟨echo,m',k',j⟩ messages have been received in this round
            and first-accept(k', j)}
12:     id-recv_i(accepted)
```

$(r, 2)$, for some r. Thus p_i receives at least $n - f$ $\langle$echo,$m,k,j\rangle$ messages in round $(r, 2)$. Consequently, at least $n-2f$ nonfaulty processors sent $\langle$echo,$m,k,j\rangle$ in round $(r, 2)$. Let p_h be among the first nonfaulty processors to send $\langle$echo,$m,k,j\rangle$ (in any round). Since the threshold for sending an echo due to the receipt of many echoes is $n - 2f > f$, and since only faulty processors have sent $\langle$echo,$m,k,j\rangle$ in earlier rounds, p_h sends $\langle$echo,$m,k,j\rangle$ due to the receipt of $\langle$init,$m,k\rangle$ from p_j in round $(k, 1)$ (see Line 4 of the code). Since p_j is nonfaulty, p_i broadcasts m in simulated round k.

Note that this proof only requires that $n > 3f$; however, a later property, Faulty Liveness, requires that $n > 4f$.

No Duplicates: This condition holds because of the first-accept check performed in Lines 6 and 11.

Nonfaulty Liveness: This condition follows in a straightforward way from the code.

Faulty Liveness (Relay): Suppose nonfaulty processor p_i accepts (m, k) from processor p_h in simulated round r. Then there are at least $n-2f$ nonfaulty processors that send $\langle$echo,$m,k,h\rangle$ for some h, say, in round $(r, 2)$.

Thus every nonfaulty processor p_j receives at least $n-2f$ $\langle$echo,$m,k,h\rangle$ messages in round $(r, 2)$. The number of additional $\langle$echo,$*,k,h\rangle$ messages received by p_j in

round $(r, 2)$ is at most $2f$; since $n > 4f$, this number is smaller than $n - 2f$, and thus p_j cannot send a different $\langle$echo,$*$,k,$h\rangle$ message. So p_j sends $\langle$echo,m,k,$h\rangle$ in round $(r + 1, 1)$. Consequently, every nonfaulty processor receives at least $n - f$ $\langle$echo,m,k,$h\rangle$ messages in round $(r + 1, 1)$ and accepts (m, k) from p_h by simulated round $r + 1$.

Faulty Integrity: Suppose nonfaulty processor p_i accepts (m, k) from processor p_h in round r and and nonfaulty processor p_j accepts (m', k) from p_h in round r'. Toward a contradiction, assume $m \neq m'$. If $r < r'$, then by Faulty Liveness, p_j accepts (m, k) from p_h, and by No Duplicates, it cannot accept (m', k) later; the same argument holds if $r' < r$. Thus it suffices to consider the case in which p_i accepts (m, k) from p_h and p_j accepts (m', k) from p_h in the same round.

As argued above for Nonfaulty Integrity, there are at least $n - 2f$ nonfaulty processors that send $\langle$echo,m,k,$h\rangle$ in this round, and at least $n - 2f$ nonfaulty processors that send $\langle$echo,m',k,$h\rangle$ in this round. Since each nonfaulty processor only sends one $\langle$echo,$*$,k,$h\rangle$ message in each round, these sets of nonfaulty processors are disjoint. Thus the total number of nonfaulty processors, $n-f$, is at least $2(n-2f)$. This implies that $n \leq 3f$, a contradiction. □

What are the costs of this simulation? Clearly, the simulation doubles the number of rounds.

As specified, the algorithm requires processors to echo messages in all rounds after they are originally sent. Yet, once a processor accepts a message, Faulty and Nonfaulty Integrity guarantee that all processors accept this message within at most a single round. Thus we can modify the simulation so that a processor stops echoing a message one round after it accepts it.

We now calculate the maximum number of bits sent in messages by nonfaulty processors. Consider message m broadcast in round k by a nonfaulty processor p_i. As a result, p_i sends $\langle$init,m,$k\rangle$ to all processors, and then all nonfaulty processors exchange $\langle$echo,m,k,$i\rangle$ messages. The total number of bits sent due to m is $O(n^2(s+\log n + \log k))$, where s is the size of m in bits.

Theorem 12.2 *Using Algorithm 36, the Byzantine model simulates the identical Byzantine model, if $n > 4f$. Every simulated round requires two rounds, and the number of message bits sent by all nonfaulty processors for each simulated round k message m from a nonfaulty processor is $O(n^2(s + \log n + \log k))$, where s is the size of m in bits.*

12.4 SIMULATING OMISSION FAILURES ON TOP OF IDENTICAL BYZANTINE FAILURES

The *omission* model of failure is intermediate between the more benign model of crash failures and the more malicious model of Byzantine failures. In this model, a faulty processor does not fabricate messages that are not according to the algorithm;

however, it may omit to send or receive some messages, send a message to some processors and not to others, etc.

After defining the model more precisely, we describe a simulation of omission failures on top of identical Byzantine failures.

12.4.1 Definition of Omission

The formal definition of a synchronous message-passing communication system subject to omission failures is the same as for the Byzantine case, except that the conditions to be satisfied are the following:

Integrity: Every message received by processor p_i from processor p_j in round k was sent in round k by p_j.

Nonfaulty Liveness: The message sent in round k by nonfaulty processor p_i is received by *nonfaulty* processor p_j in round k.

Note that a faulty processor may fail to receive messages sent by some nonfaulty processors, and messages sent by a faulty processor may fail to be received at some processors.

When there is the potential for confusion, "send" is replaced with "om-send" and "recv" with "om-recv."

The omission failure model is a special case of the Byzantine failure model. However, the omission failure model does not allow the full range of faulty behavior that Byzantine failures can exhibit; for instance, the contents of messages cannot be arbitrary. Thus omission failures are strictly weaker than Byzantine failures.

12.4.2 Simulating Omission

The identical Byzantine fault model gives us some of the properties of omission failures. That is, if two nonfaulty processors receive a message in some round from another processor, faulty or nonfaulty, then it must be the same message. Yet, the identical Byzantine model still allows a nonfaulty processor to send incorrect messages, not according to the protocol, for example, claiming it received a message that was never sent to it.

To circumvent this problem, we apply a *validation* procedure, exploiting the fact that the same messages should be received by all processors in the identical Byzantine model, although possibly with a one round lag. Thus, whenever a processor p_i sends a message m, it also sends its *support* set, i.e., the messages that cause p_i to generate m. If p_i generated this message correctly, then the receiver of m has also received the support messages, and the receiver can check the validity of generating m.

Unlike the other simulations we have seen, this one requires knowledge of the application program A, i.e., the environment that is using the communication system being simulated.

The pseudocode for the simulation appears in Algorithm 37. We assume, without loss of generality, that the message to be sent in round k of A by a processor is the

Algorithm 37 Round k of the simulation of omission failures: code for processor p_i, $0 \leq i \leq n-1$.

Initially $valid = \emptyset$, $accepted = \emptyset$, and $pending = \emptyset$

```
1:  In response to om-send_i(m):
2:      id-send_i(⟨m,accepted⟩)
3:      id-recv_i(R)
4:      add R to pending
5:      validate(pending)
6:      accepted := {m' with sender p_j : (m', j, k) ∈ valid}
7:      om-recv_i(accepted)

8:  procedure validate(pending):
9:      for each ⟨m',support,k'⟩ ∈ pending with sender p_j,
            in increasing order of k', do
10:         if k' = 1 then
11:             if m' is an initial state of the A process on p_j then
12:                 add (m', j, 1) to valid and remove it from pending
13:         else                                              // k' > 1
14:             if (m'', h, k' − 1) ∈ valid for each m'' ∈ support with sender p_h
                        and (v, j, k' − 1) ∈ valid for some v
                        and m' = transition_A(j, v, support)
15:             then add (m', j, k') to valid and remove it from pending
```

current state of that processor. The function $\mathsf{transition}_A(i, s, R)$ returns the state of p_i resulting from applying algorithm A's transition function when p_i is in state s (which encodes the round number) and accepts the set R of messages.

We would like to show that Algorithm 37 allows the identical Byzantine failure model to simulate the omission failure model. Recall that the definition of simulation from Chapter 7 is the following: For every execution α of the algorithm that is admissible (i.e., fair, user compliant for the omission failures specification, and correct for the identical Byzantine communication system), the restriction of α to the omission failure interface satisfies the conditions of the omission failure model. However, the omission failure model places an integrity condition on messages received by faulty processes, as well as nonfaulty: Any message received must have been sent. Yet the identical Byzantine model has no such condition: Messages received by faulty processors can have arbitrary content. Thus there is no way to guarantee for *all* the processes the illusion of the omission failure model in the presence of identical Byzantine failures. Instead we weaken the definition of simulation so that it only places restrictions on the views of the *nonfaulty* processes.

Communication system $\mathcal{C}_1$ *simulates* communication system $\mathcal{C}_2$ *with respect to the nonfaulty processors when the environment algorithm is known* if, for every algorithm A whose bottom interface is the interface of $\mathcal{C}_2$, there exists a collection of processes,

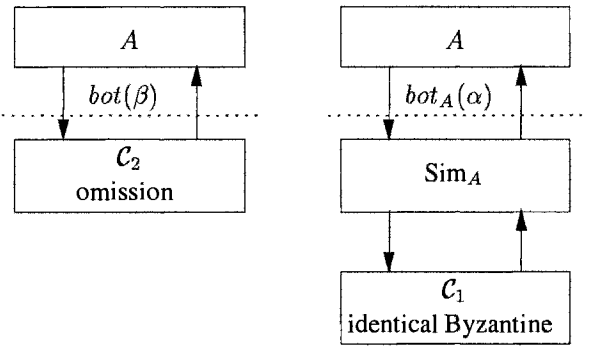

Fig. 12.3 Illustration for the definition of simulating with respect to the nonfaulty processors; identical Byzantine simulates omission failures with respect to the nonfaulty processors.

one for each node, called Sim_A (the simulation program) that satisfies the following (see Fig. 12.3):

1. The top interface of Sim is the interface of $\mathcal{C}_2$.
2. The bottom interface of Sim is the interface of $\mathcal{C}_1$.
3. Let α be any execution of the system in which A is layered on top of Sim_A that is correct for communication system $\mathcal{C}_1$. There must exist an execution β of A that is correct for communication system $\mathcal{C}_2$ such that, informally speaking, the nonfaulty algorithm A processes have the same views in β as they do in α. More formally, $bot(\beta)|P_{\text{NF}} = bot_A(\alpha)|P_{\text{NF}}$, where P_{NF} is the set of processors that are nonfaulty in α and $bot_A(\alpha)$ is the restriction of α to the events of the bottom interface of A.

We define execution β of A as follows. We have the same nonfaulty processors in β as in α. The execution β has the standard synchronous rounds structure.

The initial state of processor p_j in β is the content of any round 1 message from p_j that is validated by a nonfaulty processor in α. If no nonfaulty processor ever does so, then p_j's initial state is arbitrary. By the definition of identical Byzantine, the initial state of p_j is well-defined—no two nonfaulty processors will validate different round 1 messages from p_j.

In round k of β, processor p_j broadcasts its current state and changes state according to A depending on its current state and the messages accepted.

In round k of β, processor p_j accepts message m from processor p_i if and only if, in α, some nonfaulty processor validates p_j's round $k+1$ message claiming that p_j accepted m from p_i in round k. Although it may seem counter-intuitive, a message from p_j is defined to be received in β only if some nonfaulty processor validates it in the next round of α; later, we prove that messages sent by nonfaulty processors are always received.

First, we show that if a nonfaulty processor validates another processor's round r message in α, then by the next round every nonfaulty processor has done so, and the contents of the message are consistent with the algorithm A state in β.

Lemma 12.3 *If nonfaulty processor p_i validates processor p_j's round r message m in round k of α, then*

1. *Every nonfaulty processor validates p_j's round r message m by round $k+1$ of α, and*
2. *m is the state of the algorithm A process at p_j at the beginning of round r in β.*

Proof. First we prove Part 1.

Suppose nonfaulty processor p_i validates processor p_j's round r message m in round k of α. Then p_i validates all messages in the support set of p_j's round r message by round k. Furthermore, p_i validates p_j's round $r-1$ message v by round k. Finally, $m = \mathsf{transition}_A(j, v, \mathit{support})$.

Processor p_i is able to validate all these messages because it has received some set S of messages. By the identical Byzantine Nonfaulty and Faulty Liveness conditions, every nonfaulty processor receives all the messages in S by round $k+1$, and validates p_j's round r message.

We prove Part 2 by induction on k.

Basis: $k = 1$. Suppose nonfaulty processor p_i validates processor p_j's round r message m in round 1 of α. Then $r = 1$, m is an initial state of p_j in A, and p_i receives m as p_j's round 1 message in round 1. By definition, m is the state of p_j at the beginning of round 1 in β.

Induction: $k > 1$. Suppose nonfaulty processor p_i validates processor p_j's round r message m in round k of α. Then p_i validates all messages in the support set of p_j's round r message by round k. Furthermore, p_i validates p_j's round $r-1$ message v by round k. Finally, $m = \mathsf{transition}_A(j, v, \mathit{support})$.

By the inductive hypothesis for Part 2, v is the state of p_j at the beginning of round $r-1$ in β. By the construction of β, *support* is the set of messages received by p_j in round $r-1$ of β. Therefore, m is the state of p_j at the beginning of round r of β. □

Lemma 12.4 states that if a nonfaulty processor broadcasts a message in α, then all the nonfaulty processors validate that message in the same round and the message content is consistent with the algorithm A state in α. The proof relies on Lemma 12.3.

Lemma 12.4 *If nonfaulty processor p_j sends m as its round k message in α, then*

1. *Every nonfaulty processor validates p_j's round k message in round k of α, and*
2. *m is the state of the algorithm A process at p_j at the beginning of round k in α.*

Proof. We prove this lemma by induction on k.

Basis: $k = 1$. Suppose nonfaulty processor p_j sends m as its round 1 message in α. Part 2: Since A is running on top of the simulation, the round 1 message for p_j is an initial state of p_j in A. Part 1: By the identical Byzantine Nonfaulty Liveness condition, p_i receives p_j's round 1 message in round 1 and validates it.

Induction: $k > 1$. Suppose the lemma is true for $k - 1$. Suppose nonfaulty processor p_j sends m as its round k message in α.

Part 1: By the inductive hypothesis for Part 1, nonfaulty processor p_i validates p_j's round $k - 1$ message v in round $k - 1$. By Lemma 12.3, Part 1, p_i validates all the messages in the support set for p_j's round k message by round k (since they are validated by p_j by round $k - 1$).

Finally, $m = \text{transition}_A(j, v, \textit{support})$ and p_i validates p_j's round k message. The reason is that v is the state of the algorithm A process at p_j at the beginning of round $k - 1$ in α, by the inductive hypothesis for Part 2, and *support* is the set of messages accepted by p_j in round $k - 1$.

Part 2 follows from the previous paragraph. □

Lemma 12.5 uses the last two lemmas to show that β satisfies the omission conditions.

Lemma 12.5 *The execution β satisfies the definition of the omission model.*

Proof. *Integrity:* Suppose p_i accepts m from p_j in round k of β. Then in α, some nonfaulty processor p_h validates p_i's round $k + 1$ message claiming p_i accepted m from p_j in round k. We must show that p_j broadcast m in round k of β.

Since p_h validates p_i's round $k + 1$ message, it validates all the messages in the support set for p_i's round $k + 1$ message, including p_j's round k message containing m. Since p_h validates p_j's round k message, by Lemma 12.3, Part 2, m is the state of p_j at the beginning of round k in β. Thus p_j broadcasts m in round k of β.

Nonfaulty Liveness: Suppose nonfaulty processor p_i broadcasts message m in round k of β. Let p_j be any nonfaulty processor. We must show that, in α, some nonfaulty processor p_h validates p_j's round $k + 1$ message claiming that p_j accepted m from p_i in round k.

By Lemma 12.4, Part 2, the algorithm A process at p_i in α is in state m at the beginning of round k. Thus p_i broadcasts m in round k of α. Thus p_i sends m and its support set to all processors in round k of α.

By Lemma 12.4, Part 1, p_j validates p_i's round k message in round k of α. Thus p_j's round $k + 1$ message m' includes p_i's round k message m in its support set.

By Lemma 12.3, Part 1, p_h validates all the supporting messages for p_j's round $k + 1$ message by round $k + 1$ of α. By Lemma 12.4, Part 1, p_h validates p_j's round k message v in round k. Finally, p_h validates p_j's round $k + 1$ message m', since $m' = \text{transition}_A(j, v, \textit{support})$. The reason is that the following three facts are true: v is the state of the algorithm A process at p_j at the beginning of round k in α (by Lemma 12.4, Part 2); *support* is the set of messages accepted by p_j in round k; and m' is the message broadcast by p_j in round k of α. □

We finish by showing that the application A processes on nonfaulty nodes have the same views in α and β.

Lemma 12.6 $bot(\beta)|P_{\mathrm{NF}} = bot_A(\alpha)|P_{\mathrm{NF}}$.

Proof. Let p_j be a nonfaulty processor. We must show that in each round, the messages it receives from the underlying omission communication system in β are the same as the messages it accepts in α. Suppose p_j receives message m from processor p_i in round k of β. By the definition of β, some nonfaulty processor p_h validates p_j's round $k+1$ message in α, claiming that p_j accepted m from p_i in round k. By the identical Byzantine Nonfaulty Integrity condition, p_j really did send this message to p_h, and thus p_j did accept m from p_i in round k. □

The simulation has no overhead in rounds over the identical Byzantine—it just requires some additional tests at the end of each round. However, the messages sent by the simulation are significantly longer than messages sent by the identical Byzantine simulation, because each message is accompanied by its support set. In the worst case, the support set includes n messages, one from each processor, and thus each message requires $O(n \cdot s)$ bits, where s is the maximum size of an algorithm A message in bits.

We summarize this section with the following theorem:

Theorem 12.7 *Using Algorithm 37, the identical Byzantine failure model simulates the omission failure model with respect to the nonfaulty processors, when the environment algorithm of the omission system is known. The simulation adds no additional rounds and multiplies the number of message bits (sent by a nonfaulty processor) by* n.

12.5 SIMULATING CRASH FAILURES ON TOP OF OMISSION FAILURES

In this section, we make an additional step and show how to simulate crash failures in a system with omission failures with respect to nonfaulty processors. The environment algorithm need not be known. By combining this simulation with the simulations described in the Sections 12.3 and 12.4, we can simulate crash failures with respect to the nonfaulty processors in a system with Byzantine failures, when the environment algorithm is known. (See Exercise 12.13.)

12.5.1 Definition of Crash

The formal definition of a synchronous message-passing communication system subject to crash failures is the same as for the omission case, except that the conditions to be satisfied are the following:

Integrity: Every message received by processor p_i from processor p_j in round k was sent in round k by p_j.

Nonfaulty Liveness: The message sent in round k by nonfaulty processor p_i is received by (faulty or nonfaulty) processor p_j in round k.

Faulty Liveness: If processor p_i fails to receive (faulty) processor p_j's round k message, then no processor receives any message from p_j in round $k+1$.

The Faulty Liveness condition implies that a faulty processor works correctly, sending and receiving the correct messages, up to some round. The faulty processor might succeed in delivering only some of its messages for that round. Subsequently, no processor ever hears from it again.

When there is the potential for confusion, "send" is replaced with "crash-send" and "recv" with "crash-recv."

12.5.2 Simulating Crash

In both crash and omission failure models, processors fail by not sending (or receiving) some of the messages. However, in the crash failure model, once a processor omits to send a message it does not send any further messages, whereas in the omission failure model, a processor may omit to send a message in one round, and then resume sending messages in later rounds.

Our approach for simulating crash failures is by having a processor p_i "crash" itself if it omits a message. A processor crashes itself by sending a special ⟨crashed⟩ message, with empty content, in every subsequent round, which is ignored by the recipients.

How can processor p_i detect that it has omitted to send a message? We require processors to echo messages they receive; then, if some processor, say, p_j, does not echo p_i's message, then either p_i omitted to send this message, or p_j omitted to receive this message. If p_i receives less than $n-f$ echoes of its message, then it blames itself for not sending the message and crashes itself; otherwise, it blames p_j (and the other processors who did not echo the message) and continues.

Unfortunately, it is possible that p_i is faulty and omits to send a message only to a single nonfaulty processor, p_j; it is also possible that p_j will not even know this has happened. (See Exercise 12.5.) To get around this problem, we require that $n > 2f$. In this case, if p_i decides not to crash itself, that is, if it sees at least $n-f$ echoes of its own message, then, because $n-f > f$, at least one nonfaulty processor echoes p_i's message. Processors accept any echoed message they receive, even if they did not receive it directly. (See Exercise 12.6.)

Thus each round k of the crash failure model translates into two rounds of the omission failure model, $2k-1$ and $2k$, also denoted $(k,1)$ and $(k,2)$.

In the first round, a processor sends to all processors the message it is supposed to broadcast. In the second round, processors echo the messages they have received. If a processor receives an echo of its own message from less than $n-f$ processors, then it crashes itself (i.e., sends special crash messages in subsequent rounds). If a processor receives an echo of a message it did not receive directly, it accepts it. The pseudocode for the simulation appears in Algorithm 38.

Algorithm 38 Simulating round $k \geq 1$ for crash failures on top of omission failures: code for processor p_i, $0 \leq i \leq n-1$.

```
1: round (k, 1): in response to crash-send_i(m):
2:     om-send_i(⟨init,m⟩)
3:     om-recv_i(R)
4:     S := { ⟨echo,m',j⟩ : ⟨init,m'⟩ with sender p_j is in R}

5: round (k, 2):
6:     om-send_i(S)
7:     om-recv_i(R)
8:     if < n − f messages in R contain ⟨echo,m,i⟩ then crash self
9:     crash-recv_i ( {m' with sender p_j : ⟨echo,m',j⟩ is contained in
                                            a message in R} )
```

We will prove that this algorithm enables the omission failure model to simulate the crash failure model with respect to the nonfaulty processors. We cannot show that the omission failure model simulates the crash model for *all* processors. The reason is that the crash Nonfaulty Liveness condition states that even faulty processors must receive every message sent be a nonfaulty processor; yet in the omission model, the faulty processors can experience receive omissions. However, unlike Section 12.4, the environment algorithm need not be known. The definition of *simulating with respect to the nonfaulty processors* is the same as the definition of (globally) simulating, from Section 7.5, except that condition 3 becomes:

3'. For every execution α of Sim that is $(\mathcal{C}_2, \mathcal{C}_1)$-admissible, there exists a sequence $\sigma \in seq(\mathcal{C}_2)$ such that $\sigma|P_{\text{NF}} = top(\alpha)|P_{\text{NF}}$, where P_{NF} is the set of processors identified as nonfaulty by the partition that exists since σ is in $seq(\mathcal{C}_2)$.

Fix an admissible execution α of Algorithm 38 (i.e., it is fair, user compliant for the crash specification, and correct for the omission communication system).

We will define a sequence σ of crash events and then show that σ satisfies the specification of the crash system and that nonfaulty processors have the same views in α and σ.

The sequence σ conforms to the basic round structure required by the definition of the crash system. The message in the round k crash-send$_j$ event in σ is the same as the message in the round k crash-send$_j$ event in α, for all k and j. The set of messages in the round k crash-recv$_j$ event in σ contains message m from p_i if and only if p_i broadcasts m in round $(k, 1)$ of α and either p_i has not crashed by the end of round $(k, 2)$ of α or p_j accepts a message from p_i in round $(k, 2)$ of α.

We first show, in Lemma 12.8, that nonfaulty processors never crash themselves in α. Then we show, in Lemma 12.9, that the same messages are accepted at each round in α and σ by processors that have not yet crashed themselves in α. These two lemmas are used to show, in Lemma 12.10, that σ satisfies the crash properties. Finally, Lemma 12.11 states that the application processes on nonfaulty nodes have the same views in α and σ.

Lemma 12.8 *If processor p_i is nonfaulty, then p_i never crashes itself in α.*

Proof. We prove by induction on k that p_i has not crashed by the beginning of round $(k, 1)$.

The basis, $k = 1$, follows because processes are initialized to be not yet crashed.

Suppose $k > 1$. By the inductive hypothesis, p_i has not crashed by the beginning of round $(k-1, 1)$. Thus it sends an $\langle \mathsf{init} \rangle$ message to all processors in round $(k-1, 1)$. All nonfaulty processors receive p_i's $\langle \mathsf{init} \rangle$ message, because of the omission Nonfaulty Liveness condition, and echo it. Thus p_i receives at least $n - f$ echoes for its own round $k-1$ message and does not crash itself by the beginning of round $(k, 1)$. □

Lemma 12.9 *For all $k \geq 1$, and every processor p_i that has not crashed by the beginning of round $(k, 1)$ in α, the messages that p_i accepts in round $(k-1, 2)$ of α are the same as the messages received by p_i in round $k-1$ of σ.*

Proof. Suppose p_i has not crashed itself by the beginning of round $(k, 1)$. We will show that the messages accepted by p_i in round $(k-1, 2)$ of α are the same as those accepted in round $k-1$ of σ. Consider processor p_j. It broadcasts m' in round $(k-1, 2)$, and thus, by construction of σ, it broadcasts m' in round $k-1$ of σ. By the definition of σ, p_i receives m' from p_j in round $k-1$ of σ if and only if p_j has not crashed by the end of round $(k, 2)$ or p_i accepted a message from p_j in round $(k, 2)$ of α.

The only potential discrepancy between α and σ is if p_i does not accept the message from p_j in round $(k-1, 2)$ of α, yet p_j does not crash by the end of round $(k-1, 2)$. Since p_j does not crash, it receives at least $n - f$ echoes for its own round $k-1$ message. Since, by assumption, p_i does not yet crash either, it gets at least $n - f$ echoes for its own round $k-1$ message. Since p_i does not accept p_j's round $k-1$ message, these echoes are from processors distinct from the processors that echoed p_j's round $k-1$ message.

Thus n, the total number of processors, must be at least $2(n - f)$, implying that $n \leq 2f$, a contradiction. □

Lemma 12.10 *The sequence σ satisfies the definition of the crash model.*

Proof. *Integrity.* Suppose p_j accepts m from p_i in round k of σ. By the definition of σ, p_i broadcasts m in round $(k, 1)$ of α. By construction, p_i broadcasts m in round k of σ.

Nonfaulty Liveness. Suppose nonfaulty processor p_i broadcasts m in round k of σ. By construction of σ, p_i broadcasts m in round $(k, 1)$ of α. Since p_i is nonfaulty, by Lemma 12.8 p_i has not yet crashed itself, and thus it sends $\langle \mathsf{init}, m \rangle$ in round $(k, 1)$ of α. By the definition of σ, p_j accepts m from p_i in round k of σ.

Faulty Liveness. Suppose p_j fails to accept p_i's round k message in σ. Then by the definition of σ, p_i has crashed itself by the end of round $(k, 2)$ of α. Thus p_i sends only special crashed messages in round $(k+1, 1)$ of α, and no processor accepts a message from p_i in round $k+1$ of σ. □

Finally, we must show that the nonfaulty processors have the same views in α as in σ. We must show that the messages a nonfaulty processor receives in each round of σ are the same as the messages it accepts in the corresponding simulated round of α. This is true because of Lemmas 12.8 and 12.9.

Lemma 12.11 $\sigma|P_{\text{NF}} = top(\alpha)|P_{\text{NF}}$.

Similarly to Algorithm 36, the number of message bits sent by a nonfaulty processor is $O(n^2(s+\log n))$, where s is the maximum size of an environment message in bits. This implies the following simulation result.

Theorem 12.12 *Using Algorithm 38, the omission failure model simulates the crash failure model with respect to the nonfaulty processors, if $n > 2f$. Every simulated round requires two rounds, and the number of message bits sent by a nonfaulty processor is $O(n^2(s + \log n))$, where s is the maximum size of an environment message in bits.*

By combining Theorem 12.12 with Theorems 12.2 and 12.7, we obtain the following important simulation result.

Theorem 12.13 *If $n > 4f$, the Byzantine failure model simulates the crash model with respect to the nonfaulty processors when the environment algorithm is known. Every simulated round requires four rounds, and the number of message bits sent by a nonfaulty processor for simulated round k is $O(n^5(s + \log n + \log k)(s + \log n))$, where s is the maximum size of an environment message in bits.*

12.6 APPLICATION: CONSENSUS IN THE PRESENCE OF BYZANTINE FAILURES

As a simple example, we apply the simulations developed in this chapter to derive an algorithm for solving consensus in the presence of Byzantine failures. Recall the very simple consensus algorithm that tolerates crash failures from Chapter 5 (Algorithm 15). This algorithm requires $f+1$ rounds and messages of size $n \log |V|$ bits, where $|V|$ is the number of input values. We can run this algorithm together with the simulation of crash failures in a system with Byzantine failures. By appealing to Theorem 12.13, we have:

Theorem 12.14 *If $n > 4f$, then there exists an algorithm that solves the consensus problem in the presence of f Byzantine failures. The algorithm requires $4(f + 1)$ rounds and messages of size $O(n^5(n|V| + \log n + \log f)(n \log |V| + \log n))$.*

Note that this algorithm is inferior to Algorithm 16, which requires $2(f+1)$ rounds and one-bit messages. (This algorithm also requires that $n > 4f$.) A simple way to reduce the number of rounds required is to note that Algorithm 15 also tolerates omission failures (see Exercise 12.7). Therefore, we can employ the simulation of omission failures, which has smaller overhead (Theorems 12.2 and 12.7), to get:

Theorem 12.15 *If $n > 4f$, then there exists an algorithm that solves the consensus problem in the presence of f Byzantine failures. The algorithm requires $2(f+1)$ rounds and messages of size $O(n^3(n \log |V| + \log n + \log f))$.*

There is a simulation of identical Byzantine failures that requires only that $n > 3f$ (Exercise 12.11); this implies a simulation of crash failures in a system with Byzantine failures that only requires that $n > 3f$. This, in turn, implies:

Theorem 12.16 *If $n > 3f$, then there exists an algorithm that solves the consensus problem in the presence of f Byzantine failures. The algorithm requires $3(f+1)$ rounds and messages of size $O(n^3(n \log |V| + \log n + \log f))$.*

12.7 ASYNCHRONOUS IDENTICAL BYZANTINE ON TOP OF BYZANTINE FAILURES

We have seen simulations of crash failures in a system with more severe failures—omissions or even Byzantine failures. These simulations applied to the synchronous model; similar simulations exist also for the asynchronous model, but they are fairly restricted and can only be applied to deterministic algorithms. Because many interesting problems, for example, consensus, have only non-deterministic fault-tolerant solutions in asynchronous systems, even in the presence of the most benign failures, the benefit of such simulations is rather limited. More useful is the asynchronous version of Algorithm 36, which simulates identical Byzantine failures in the presence of Byzantine failures. This simulation works for any algorithm and is not restricted to deterministic algorithms; as we shall see later (in Chapters 13 and 14), this makes it particularly helpful in designing clock synchronization algorithms for Byzantine failures and randomized asynchronous algorithms for consensus in the presence of Byzantine failures.

12.7.1 Definition of Asynchronous Identical Byzantine

The definition of the identical Byzantine fault model for the synchronous case in Section 12.3.1 refers explicitly to round numbers, and therefore, it has to be altered in order to fit the asynchronous model. We replace the reference to specific rounds with the requirement for *eventual* delivery.

In the synchronous model, processors could broadcast different messages at different rounds. It was guaranteed that, for each round, at most one message is accepted from each processor. In the asynchronous model, there is no similar notion of round. Instead, we assume that each processor assigns distinguishing tags to the messages it broadcasts.

The definition of asynchronous identical Byzantine is the same as for the asynchronous crash point-to-point system in Chapter 8, except that the event names are id-send and id-recv; each message sent and received has the format (m, k), where

m is the message content and k is the tag, and the conditions to be satisfied are the following:

Uniqueness: There is at most one id-send$_i(*, k)$ event, for each i and k. This is a restriction on the inputs from the environment.

Nonfaulty Integrity: If nonfaulty processor p_i receives (m, k) from nonfaulty processor p_j, then p_j sent (m, k).

Faulty Integrity (Identical Contents): If nonfaulty processor p_i receives (m, k) from processor p_h and nonfaulty processor p_j receives (m', k) from p_h, then $m = m'$.

No Duplicates: Nonfaulty processor p_i receives only one message with tag k from p_j.

Nonfaulty Liveness: If nonfaulty processor p_i sends (m, k), then nonfaulty processor p_j receives (m, k) from p_i.

Faulty Liveness (Relay): If nonfaulty processor p_i receives (m, k) from (faulty) processor p_h, then nonfaulty processor p_j receives (m, k) from p_h.

These latter five conditions are analogous to those in the synchronous case (cf. Section 12.3), but with no reference to rounds.

12.7.2 Definition of Asynchronous Byzantine

We want to simulate the asynchronous identical Byzantine fault model specified in Section 12.7.1 in an asynchronous point-to-point system subject to (unrestricted) Byzantine failures. In this subsection we define the implementation system.

The definition of the asynchronous Byzantine model is the same as for the asynchronous crash point-to-point system in Chapter 8, except that the conditions to be satisfied are the following:

Nonfaulty Integrity: If nonfaulty processor p_i receives m from nonfaulty processor p_j, then p_j sent m to p_i.

No Duplicates: No message sent is received more than once.

Nonfaulty Liveness: If nonfaulty processor p_i sends m to nonfaulty processor p_j, then p_j receives m from p_i.

When there is the potential for confusion, "send" is replaced with "Byz-send" and "recv" with "Byz-recv."

12.7.3 Simulating Asynchronous Identical Byzantine

The synchronous algorithm for simulating identical Byzantine failures (Algorithm 36) can be modified to work in the asynchronous case; this simulation also assumes that

Algorithm 39 The asynchronous identical Byzantine simulation: code for processor p_i, $0 \leq i \leq n-1$.

```
1:  when id-send_i(m, k) occurs:
2:      enable Byz-send_i(⟨init,m,k⟩) to all processors

3:  when Byz-recv_i(⟨init,m,k⟩) from p_j occurs:
4:      if first-echo(k, j) then enable Byz-send_i(⟨echo,m,k,j⟩) to all processors

5:  when Byz-recv_i(⟨echo,m,k,j⟩) occurs:
6:      num := number of copies of ⟨echo,m,k,j⟩ received so far
            from distinct processors
7:      if num ≥ n − f and first-ready(k, j) then
8:          enable Byz-send_i(⟨ready,m,k,j⟩) to all processors

9:  when Byz-recv_i(⟨ready,m,k,j⟩) occurs:
10:     num := number of copies of ⟨ready,m,k,j⟩ received so far
            from distinct processors
11:     if num ≥ n − 2f and first-ready(k, j) then
12:         enable Byz-send_i(⟨ready,m,k,j⟩) to all processors
13:     if num ≥ n − f and first-accept(k, j) then
14:         enable id-recv_i(m, k) from p_j
```

$n > 4f$. We do not present this simulation here, and leave it as an exercise to the reader (Exercise 12.10). Instead, we present another simulation that only requires that $n > 3f$, but uses three types of messages. Interestingly, this implementation can be modified to work in the synchronous model (Exercise 12.11).

To broadcast a high-level message (m, k), the sender, p_i, sends a message $\langle \text{init}, m, k \rangle$ to all processors (including itself). Processors receiving this init message act as witnesses for this broadcast and send a message $\langle \text{echo}, m, k, i \rangle$ to all processors. Once a processor receives $n - f$ echo messages, it notifies the other processors it is about to accept (m, k) from p_i, by sending a $\langle \text{ready}, m, k, i \rangle$ message. Once a processor receives $n - f$ $\langle \text{ready}, m, k, i \rangle$ messages, it accepts (m, k) from p_i. In addition, if a processor receives $n - 2f$ ready messages, it also sends a ready message.

The pseudocode appears in Algorithm 39. The function first-echo(k, j) returns true if and only if the processor has not already sent an echo message with tag k for p_j; first-ready(k, j) returns true if and only if the processor has not already sent a ready message with tag k for p_j; first-accept(k, j) returns true if and only if the processor has not yet accepted a message from p_j with tag k.

We now show that the latter five properties of asynchronous identical Byzantine faults are satisfied by this algorithm. The following lemma and theorem are with respect to an arbitrary admissible execution α (i.e., it is fair, user compliant for the

asynchronous identical Byzantine specification, and correct for the asynchronous Byzantine communication system).

Lemma 12.17 *If one nonfaulty processor sends* $\langle \text{ready}, m, k, h \rangle$ *and another nonfaulty processor sends* $\langle \text{ready}, m', k, h' \rangle$, *then* m *must equal* m'.

Proof. Suppose in contradiction $m \neq m'$. Let p_i be the first nonfaulty processor to send $\langle \text{ready}, m, k, h \rangle$. Since p_i can only receive ready messages from the f faulty processors up till this point and f is less than $n - 2f$, p_i received at least $n - f$ $\langle \text{echo}, m, k, h \rangle$ messages. At least $n - 2f$ of these are from nonfaulty processors.

Similarly, p_j, the first nonfaulty processor to send $\langle \text{ready}, m', k, h \rangle$, received at least $n - 2f$ $\langle \text{echo}, m', k, h \rangle$ messages from nonfaulty processors.

Since each nonfaulty processor sends only one echo message for k and h, the total number of nonfaulty processors, $n - f$, must be at least $2(n - 2f)$, implying $n \leq 3f$, a contradiction. □

Theorem 12.18 *The latter five conditions for asynchronous identical Byzantine are satisfied.*

Proof. *Nonfaulty Integrity:* Suppose nonfaulty processor p_i accepts (m, k) from nonfaulty processor p_j. Thus p_i receives at least $n - f$ $\langle \text{ready}, m, k, j \rangle$ messages, at least $n - 2f$ of which are from nonfaulty processors. Let p_h be the first nonfaulty processor to send $\langle \text{ready}, m, k, j \rangle$. Then p_h cannot have received $n - 2f$ ready messages already, since $n - 2f > f$ (recall that $n > 3f$) and up till now only faulty processors have sent $\langle \text{ready}, m, k, j \rangle$. Thus p_h receives at least $n - f$ $\langle \text{echo}, m, k, j \rangle$ messages, at least $n - 2f$ of which are from nonfaulty processors. A nonfaulty processor only sends $\langle \text{echo}, m, k, j \rangle$ if it receives $\langle \text{init}, m, k \rangle$ from p_j. Since p_j is nonfaulty, p_j did broadcast (m, k).

Faulty Integrity (Identical Contents): Suppose nonfaulty processor p_i accepts (m, k) from processor p_h and nonfaulty processor p_j accepts (m', k) from p_h. Assume for contradiction that $m \neq m'$. Thus p_i receives at least $n - f$ $\langle \text{ready}, m, k, h \rangle$ messages, at least $n - 2f$ of which are from nonfaulty processors. Similarly, p_j receives at least $n - f$ $\langle \text{ready}, m', k, h \rangle$ messages, at least $n - 2f$ of which are from nonfaulty processors. But this violates Lemma 12.17.

No Duplicates: This condition is ensured by the first-accept check in the code.

Nonfaulty Liveness: Suppose nonfaulty processor p_i broadcasts (m, k). Then p_i sends $\langle \text{init}, m, k \rangle$ to all processors and every nonfaulty processor p_j receives $\langle \text{init}, m, k \rangle$. This is the first $\langle \text{init}, *, k \rangle$ message that p_j has received from p_i by the uniqueness condition. Thus p_j sends $\langle \text{echo}, m, k, i \rangle$ to all processors.

Every nonfaulty processor p_j receives at least $n - f$ $\langle \text{echo}, m, k, i \rangle$ messages. So p_j receives at most f $\langle \text{echo}, m', k, i \rangle$ messages for any $m' \neq m$. Since $f < n - f$ (recall $n > 3f$), p_j sends $\langle \text{ready}, m, k, i \rangle$.

Every nonfaulty processor p_j receives at least $n - f$ $\langle \text{ready}, m, k, i \rangle$ messages, so p_j receives at most f $\langle \text{ready}, m', k, i \rangle$ messages for any $m' \neq m$. Since $f < n - f$, p_j accepts (m, k) from p_i.

Faulty Liveness (Relay): Suppose nonfaulty processor p_i accepts (m, k) from p_h. Then p_i receives at least $n - f$ $\langle$ready,$m,k,h\rangle$ messages, meaning that at least $n - 2f$ nonfaulty processors send $\langle$ready,$m,k,h\rangle$. Thus every nonfaulty processor p_j receives at least $n - 2f$ $\langle$ready,$m,k,h\rangle$ messages. By Lemma 12.17, p_j does not send $\langle$ready,$m',k,h\rangle$, with $m' \neq m$, and therefore, p_j sends $\langle$ready,$m,k,h\rangle$. Thus every nonfaulty processor receives at least $n - f$ $\langle$ready,$m,k,h\rangle$ messages and accepts (m, k) from p_h. □

It is easy to calculate the number of messages sent by nonfaulty processors. When a nonfaulty processor p_i broadcasts m, each nonfaulty processor sends one echo message to every processor, and then each nonfaulty processor sends one ready message to every processor. Hence, the simulation requires nonfaulty processors to send a total of $O(n^2)$ point-to-point messages per original broadcast message. The total number of bits is calculated as for Algorithm 36. We leave the time complexity analysis as an exercise to the reader (Exercise 12.12). To summarize:

Theorem 12.19 *The asynchronous Byzantine failures model simulates the asynchronous identical Byzantine failures model, if $n > 3f$. The number of messages sent by nonfaulty processors for each broadcast is $O(n^2)$ and the total number of bits is $O(n^2(s + \log n + \log k))$, where s is the size of m in bits. A message broadcast by a nonfaulty processor is accepted within $O(1)$ time.*

Exercises

12.1 Show that there is no loss of generality in assuming that at each round a processor sends the same message to all processors.

12.2 Show that assuming processors are nonfaulty and the network corrupts messages is equivalent to assuming processors are faulty and the network does not corrupt messages.

12.3 Explain why the following synchronous algorithm does not solve the consensus problem: Each processor broadcasts its input using Algorithm 36. Each processor waits for two rounds and then decides on the minimum value received.

12.4 Show how to reduce the size of messages in the synchronous simulation of identical Byzantine failures (Algorithm 36).

12.5 What happens in the simulation of crash failures on omission failures (Section 12.5) if $n \leq 2f$?

12.6 In the simulation of crash failures on omission failures (Section 12.5), why do we need processors to accept messages echoed by other processors?

12.7 Prove that the algorithm for consensus in the presence of crash failures (Algorithm 15) is correct even in the presence of omission failures.

12.8 Show a simulation of crash failures on top of *send omission* failures that assumes only that $n > f$. (Informally speaking, in the send omission failure model, a faulty processor can either crash permanently at some round or at intermittent rounds, the message it sends can fail to be delivered to some of the other processors.)

12.9 Show how to avoid validation of messages and use the simulation of identical Byzantine on top of Byzantine to get a simulation of Algorithm 15 with smaller messages.

Hint: Note that in this algorithm, messages include a sequence of processor identifiers and the support are messages with prefixes of this sequence.

12.10 Modify the algorithm of Section 12.3.2 to simulate asynchronous identical Byzantine faults using only two types of messages. Assume $n > 4f$. What is the asynchronous time complexity of this algorithm?

12.11 Modify the algorithm of Section 12.7.3 to simulate synchronous identical Byzantine faults assuming $n > 3f$ and using three rounds for each simulated round.

12.12 Show that the time complexity of Algorithm 39 is $O(1)$. That is, a message broadcast by a nonfaulty processor is received by all nonfaulty processors within $O(1)$ time.

12.13 (a) Show that if A can simulate B, then A can simulate B with respect to the nonfaulty processors.

(b) Show that if A can simulate B with respect to the nonfaulty processors, then A can simulate B with respect to the nonfaulty processors when the environment algorithm is known.

(c) Show that if A can simulate B according to one definition and B can simulate C according to another definition, then A can simulate C according to the weaker of the two definitions.

Chapter Notes

The identical Byzantine model is a variant on *authenticated broadcast* of Srikanth and Toueg [246]. Our simulation of the identical Byzantine model, which assumes $n > 4f$, as well as the validated broadcast and the simulation of omission failures on top of crash failures, are all based on the work of Neiger and Toueg [198]. The asynchronous simulation of identical Byzantine was first introduced by Bracha [60], which is also the source of the simulation presented here.

For the synchronous model, the first simulation of the type considered in this chapter, of crash failures on top of *send* omission failures, was given by Hadzilacos [128]. However, this simulation is not completely general and relies on certain assumptions on the behavior of faulty processors. The best (in terms of message and

time complexity) simulations of crash failures on top of Byzantine failures are due to Bazzi and Neiger [47, 46]. Their work contains a thorough study of the cost of such simulations, including lower bounds and trade-offs, in terms of the rounds overhead and the ratio of faulty processors tolerated.

Omission failures of the type considered here are sometimes called *general* omission failures, defined by Perry and Toueg [210]. Two specific types of omission failures—*send omission*, in which faulty processors only fail to send messages, and *receive omission*, in which faulty processors only fail to receive messages—have been suggested by Hadzilacos [128].

For the asynchronous model, a general simulation of crash failures on top of Byzantine failures was given by Coan [84]. It applies only to deterministic algorithms in a restricted form. One of Coan's contributions is an asynchronous simulation of identical Byzantine failures, assuming $n > 4f$; this is the origin of Neiger and Toueg's synchronous simulation we presented here (Algorithm 36); the paper [84] contains the solution to Exercise 12.10.

Distributed Computing: Fundamentals, Simulations and Advanced Topics, Second Edition
Hagit Attiya and Jennifer Welch
 ISBN: 0-471-45324-2

13

Fault-Tolerant Clock Synchronization

We now consider the problem of keeping real-time clocks synchronized in a distributed system when processors may fail. This problem is tantalizingly similar to the consensus problem, but no straightforward reductions are known between the problems.

In this chapter, we assume that hardware clocks are subject to drift. Therefore, the software clocks may drift apart as time elapses and subsequent periodic resynchronization is necessary. Now the clock synchronization problem has two parts: getting the clocks close together initially and keeping them close together.

In this chapter we focus on the problem of keeping the clocks close together, assuming that they are initially close. First, we show that, as for consensus, to solve the problem, the total number of processors, n, must be more than $3f$, where f is the maximum number of faulty processors. Then we present an algorithm for solving the problem in the presence of Byzantine failures. The algorithm is first described for a simpler fault model, and then fault-tolerant simulations from Chapter 12 are applied.

13.1 PROBLEM DEFINITION

As in Chapter 5, we have n processors in a fully connected network, in which the message delays are always in the range $[d - u, d]$, for some constants d and u. Each processor p_i has hardware clock $HC_i(t)$, adjustment variable adj_i, and adjusted clock $AC_i(t) = HC_i(t) + adj_i(t)$.

However, we now allow the possibility that up to f of the processors may exhibit Byzantine failures.

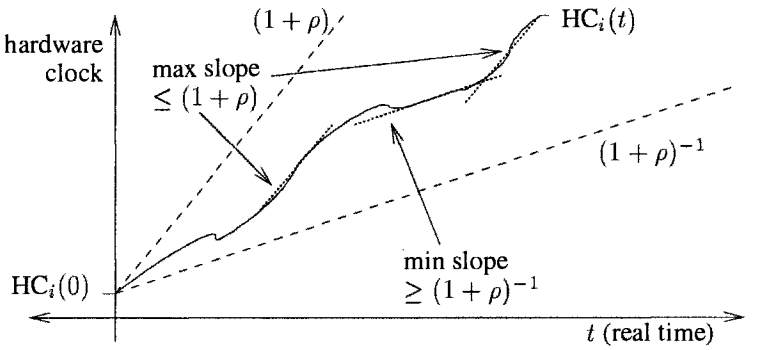

Fig. 13.1 Drift of a hardware clock.

We also consider the complications introduced when the hardware clocks can drift from real time. We assume that hardware clocks stay within a *linear envelope* of the real time; that is, there exists a positive constant ρ (the *drift*) such that each hardware clock HC_i satisfies the following property (see Fig. 13.1):

Bounded Drift: For all times t_1 and t_2, $t_2 > t_1$,

$$(1+\rho)^{-1}(t_2 - t_1) \leq HC_i(t_2) - HC_i(t_1) \leq (1+\rho)(t_2 - t_1).$$

The difference between hardware clocks of nonfaulty processors grows at a rate which is bounded by $\rho(2+\rho)(1+\rho)^{-1}$ (see Exercise 13.1).

Because hardware clocks can drift away from real time, either by gaining or losing time (or both), processors must continually resynchronize their clocks in order to keep them close. In this chapter we focus on the problem of keeping the clocks close together, assuming they begin close together. (Contrast this with the problem studied in Chapter 6, which was to get the clocks close together in the first place.)

We wish to guarantee that processors' clocks stay close to each other, assuming that they begin close to each other. To formalize the initialization assumption, we put an additional restriction on the definition of admissible execution, stating that at real time 0, the adjusted clocks of nonfaulty processors are within some bound B of each other. The amount B can be considered a parameter to the definition of admissible; the closeness of synchronization achievable may depend on B, the initial closeness. We require the following:

Clock Agreement: There exists a constant ϵ such that in every admissible timed execution, for all times t and all nonfaulty processors p_i and p_j,

$$|AC_i(t) - AC_j(t)| \leq \epsilon$$

A trivial solution would be to set all adjusted clocks to 0; to rule this out, we require that clocks stay within a linear envelope of their hardware clocks; formally this is stated as:

Clock Validity: There exists a positive constant γ such that in every admissible timed execution, for all times t and every nonfaulty processor p_i,

$$(1+\gamma)^{-1}(HC_i(t) - HC_i(0)) \leq AC_i(t) - AC_i(0) \leq (1+\gamma)(HC_i(t) - HC_i(0))$$

The clock validity condition states that the change in the adjusted clock since the beginning of the execution must be within a linear envelope of the change in the hardware clock since the beginning of the execution. Notice the difference from the hardware clock drift condition, which was a constraint on the instantaneous rate: The adjusted clock can change discontinuously, and therefore we can bound the change only over a long period. The clock validity condition is stated with respect to the hardware clocks, not real time. However, as Exercise 13.2 asks you to show, if the adjusted clock is within a linear envelope of the hardware clock and the hardware clock is within a linear envelope of real time, then the adjusted clock is within a linear envelope of real time, albeit a larger envelope.

The goal is to achieve clock agreement and validity with ϵ and γ that are as small as possible. Intuitively, the validity parameter, γ, cannot be smaller than the validity of the hardware clocks captured by ρ.

An algorithm for maintaining synchronized clocks will instruct processors to take actions periodically. A mechanism is needed for a processor to program itself to take a step when its hardware clock reaches a certain value. This ability is modeled by assuming that each processor p_i has a special state component $timer_i$ that it can set. For an execution to be admissible, each processor must take a step once its hardware clock reaches the current value of its timer.

13.2 THE RATIO OF FAULTY PROCESSORS

In this section, we show that there can be no algorithm to satisfy clock agreement and clock validity if $n \leq 3f$; this result holds for any constants ϵ and γ, regardless of their specific values.

We will prove this result using ideas similar to two we have already seen, namely, shifting of executions (used to prove the lower bound on closeness of synchronization in Chapter 6) and specifying faulty behavior with a big ring (used to prove the $n > 3f$ lower bound for consensus in Chapter 5).

Before proving the lower bound, we need a result similar to one we used to show the lower bound on the closeness of synchronization; if both hardware clocks and message delays are altered appropriately, processors cannot tell the difference. In Chapter 6, the alteration was to add certain quantities to the real times of occurrences, resulting in a shifted execution. Here we will multiply the real times by a certain quantity, resulting in a scaled execution.

Definition 13.1 *Let α be a timed execution with hardware clocks and let s be a real number. Define* $\text{scale}(\alpha, s)$ *to be the execution obtained by multiplying by s the real time associated with each event in α.*

Lemma 13.1 states the relationships between the clocks and between the message delays in a timed execution and its scaled version.

Lemma 13.1 *If α is a timed execution then in $\alpha' = \text{scale}(\alpha, s)$,*

(a) $HC_i'(t) = HC_i(t/s)$ for all times t, where HC_i is p_i's hardware clock in α and HC_i' is p_i's hardware clock in α', and

(b) $AC_i'(t) = AC_i(t/s)$ for all times t, where AC_i is p_i's adjusted clock in α and AC_i' is p_i's adjusted clock in α', and

(c) If a message has delay δ in α, then it has delay $s \cdot \delta$ in α'

Proof. The first two properties follow directly from the definition of scaling. For the last property, consider message m sent at real time t_s and received at real time t_r in α. Then in α' it is sent at real time $s \cdot t_s$ and received at real time $s \cdot t_r$. Thus its delay in α' is $s \cdot (t_r - t_s)$. □

If HC_i is a linear function, then the factor of $1/s$ can be brought out of the argument to the function, and we have $HC_i'(t) = HC_i(t)/s$.

If s is larger than 1, then the hardware clocks slow down and message delays increase. If s is smaller than 1, then the hardware clocks speed up and message delays decrease. The scaled execution may or may not be admissible, because message delays and drifts may be too large or too small; however, we still have the following result:

Lemma 13.2 *If a timed execution α satisfies the clock agreement condition with parameter ϵ or the clock validity condition with parameter γ for a set of processors, then the same is true in $\alpha' = \text{scale}(\alpha, s)$, for any $s > 0$.*

Proof. Suppose α satisfies the clock agreement condition for processors p_i and p_j. Denote p_i's adjusted clock in α by AC_i and p_i's adjusted clock in α' by AC_i'. By Lemma 13.1 (b), for any time t:

$$|AC_i'(t) - AC_j'(t)| = |AC_i(t/s) - AC_j(t/s)|$$

Since the adjusted clocks of p_i and p_j satisfy the clock agreement condition in α, AC_i and AC_j are within ϵ for every argument, including t/s, and the result follows.

Next, suppose α satisfies the clock validity condition for processor p_i. By Lemma 13.1 (b), for all times t:

$$AC_i'(t) - AC_i'(0) = AC_i(t/s) - AC_i(0/s)$$

Since the adjusted clock of p_i satisfies the clock validity condition in α,

$$AC_i(t/s) - AC_i(0/s) \leq (1+\gamma)(HC_i(t/s) - HC_i(0/s))$$

which by Lemma 13.1 (a) is equal to

$$(1+\gamma)(HC_i'(t) - HC_i'(0))$$

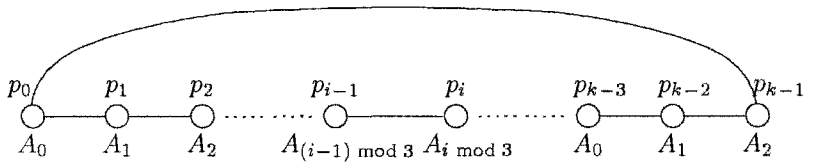

Fig. 13.2 Assignment of local algorithms in the big ring in the proof of Theorem 13.3.

The lower bound on $AC_i'(t) - AC_i'(0)$ is proved analogously. □

The main result of this section is that no algorithm can guarantee clock agreement and clock validity, if $n \leq 3f$. We only prove the special case when $f = 1$, and leave the general case as an exercise.

The proof requires that u, the uncertainty in the message delay, not be too small; specifically, u must be at least $d(1 - (1+\rho)^{-4})$. It is probably not clear at this point why this assumption is necessary, but it enables certain calculations to work out in the proof. Because typically ρ is on the order of 10^{-6}, this assumption is reasonable.

Theorem 13.3 *No algorithm can guarantee clock agreement and clock validity for $f = 1$ and $n = 3$, if $u \geq d(1 - (1+\rho)^{-4})$.*

Proof. Suppose in contradiction there is such an algorithm for $n = 3$ and $f = 1$, guaranteeing clock agreement and validity with constants ϵ and γ. Let A_i be the (local) algorithm run by p_i, for $i = 0, 1, 2$. Choose a constant k such that

1. k is a multiple of 3 and
2. $(1+\gamma)^{-1}(1+\rho)^{2(k-1)} > 1+\gamma$

The reasons for these conditions on k will be pointed out as they arise.

Consider a ring network of k processors, p_0 through p_{k-1}, in which for each i, $0 \leq i \leq k-1$, p_i runs local algorithm $A_{i \bmod 3}$ (see Figure 13.2). Here is where we use the fact that k is a multiple of 3.

We now specify a timed execution β of this ring. In β, for each i, $0 \leq i \leq k-1$ (see Fig. 13.3):

- The hardware clock of p_i is $HC_i(t) = t(1+\rho)^{1-2i}$
- The adjusted clock of p_i equals 0 at time 0, i.e., $AC_i(0) = 0$ and
- The delay of every message between p_i and $p_{(i-1) \bmod k}$ (in both directions) is $d(1+\rho)^{2i-4}$, for $0 \leq i \leq k-1$

We cannot claim that β satisfies the clock synchronization properties because the network has more than three processors and is not fully connected; moreover, the hardware clock drifts and message delays are not all admissible. However, we will be able to make some deductions about the behavior of β, by showing that pieces

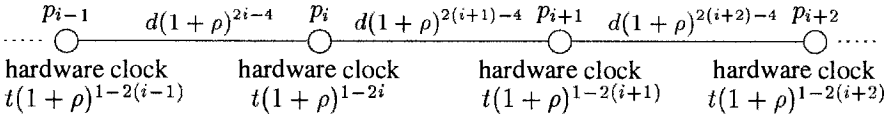

Fig. 13.3 Timed execution β of the big ring in the proof of Theorem 13.3.

of the ring "look like" certain systems in which the algorithm *is* supposed to behave properly.

Lemma 13.4 states that the adjusted clocks of two adjacent processors in the timed execution β of the ring satisfy the clock agreement condition and each adjusted clock in β satisfies the clock validity condition. The statement of the clock validity condition is simplified because the hardware clocks and adjusted clocks are all initially 0.

Lemma 13.4 *For all times t:*

(a) $|AC_i(t) - AC_{i+1}(t)| \leq \epsilon$, *for all* i, $0 \leq i \leq k-2$ *and*

(b) $(1+\gamma)^{-1}HC_i(t) \leq AC_i(t) \leq (1+\gamma)HC_i(t)$, *for all* i, $0 \leq i \leq k-1$

Proof. Fix i, $0 \leq i \leq k-2$. Take processors p_i and p_{i+1} from the big ring and put them in a three-processor ring (which is also fully connected) with a third processor that is faulty in such a way that it acts like p_{i-1} from the big ring toward p_i and acts like p_{i+2} from the big ring toward p_{i+1} (see Fig. 13.4). The hardware clock times of p_i and p_{i+1} and the message delays for messages to and from p_i and p_{i+1} are the same as in the big ring's timed execution β. Call the resulting timed execution α.

As was done in the proof of Theorem 5.7, a simple induction can be used to show that p_i and p_{i+1} have the same views in α as they do in the timed execution of the big ring and thus they have the same adjusted clocks.

Let $\alpha' = \text{scale}(\alpha, (1+\rho)^{-2i})$ (see Fig. 13.5).

We will now verify that α' is admissible. By Lemma 13.1 (a), p_i's hardware clock in α' is

$$HC_i'(t) = HC_i(t(1+\rho)^{2i}) = t(1+\rho)$$

By Lemma 13.1 (a), p_{i+1}'s hardware clock in α' is

$$HC_{i+1}'(t) = HC_{i+1}(t(1+\rho)^{2i}) = t(1+\rho)^{-1}$$

By Lemma 13.1 (c), the message delays between the faulty processor and p_{i+1} are $d(1+\rho)^{-2i}(1+\rho)^{2(i+2)-4}$, which equals d. Similarly, the message delays between p_{i+1} and p_i are $d(1+\rho)^{-2}$, and the message delays between p_i and the faulty processor are $d(1+\rho)^{-4}$. The assumption that $u \geq d(1-(1+\rho)^{-4})$ ensures that all message delays are between $d-u$ and d.

Since α' is admissible, the clock agreement and clock validity conditions hold for the adjusted clocks of p_i and p_{i+1} in α'. Lemma 13.2 implies that these conditions

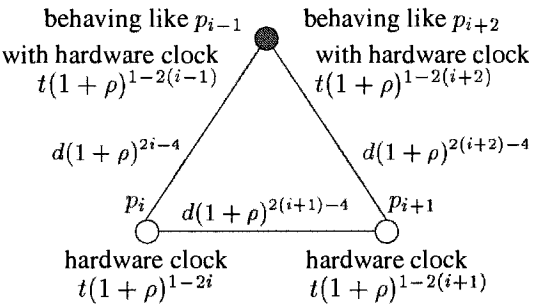

Fig. 13.4 A triangle based on the ring, for the proof of Lemma 13.4; gray node is faulty.

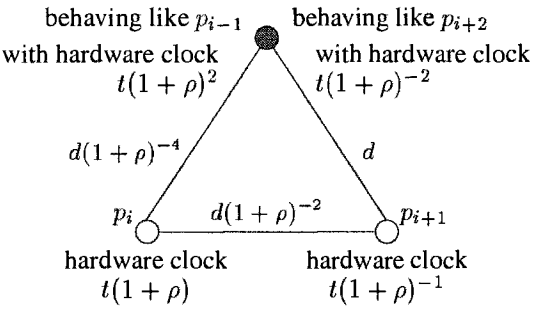

Fig. 13.5 Scaling the triangle by $(1+\rho)^{-2i}$.

also hold in α. Since p_i and p_{i+1} have the same adjusted clocks in α as in β, the lemma follows. □

We can now complete the main proof. Repeated application of Lemma 13.4 (a), implies the following inequalities:

$$\begin{aligned} AC_0(t) &\leq AC_1(t) + \epsilon \\ &\leq AC_2(t) + 2\epsilon \\ &\leq \ldots \\ &\leq AC_{k-1}(t) + (k-1)\epsilon \end{aligned}$$

Rearranging the terms produces:

$$AC_{k-1}(t) \geq AC_0(t) - (k-1)\epsilon$$

Lemma 13.4 (b) implies that $AC_0(t) \geq (1+\gamma)^{-1} HC_0(t)$, and the definition of β implies that $HC_0(t) = (1+\rho)^{2(k-1)} HC_{k-1}(t)$. Thus

$$AC_{k-1}(t) \geq (1+\gamma)^{-1}(1+\rho)^{2(k-1)} HC_{k-1}(t) - (k-1)\epsilon$$

Lemma 13.4 (b) implies that

$$AC_{k-1}(t) \leq (1+\gamma)HC_{k-1}(t)$$

The last two inequalities are combined to show:

$$(1+\gamma)^{-1}(1+\rho)^{2(k-1)}HC_{k-1}(t) - (k-1)\epsilon \leq (1+\gamma)HC_{k-1}(t)$$

Rearranging produces:

$$((1+\gamma)^{-1}(1+\rho)^{2(k-1)} - (1+\gamma))HC_{k-1}(t) \leq (k-1)\epsilon$$

HC_{k-1} increases without bound as t grows, and by choice of k, $(1+\gamma)^{-1}(1+\rho)^{2(k-1)} - (1+\gamma)$ is positive. (This is where we need the second condition in the definition of k.) Thus the left-hand side of the inequality increases without bound. Yet the right-hand side, $(k-1)\epsilon$, is a constant, which is a contradiction. □

The case when $f > 1$ is proved by reduction to this theorem, as was done in Theorem 5.8; the details are left to an exercise.

13.3 A CLOCK SYNCHRONIZATION ALGORITHM

We start with an algorithm tolerating f *timing* failures, in which nonfaulty processors fail either by crashing or by having hardware clocks whose drift exceeds the bounds given in the Bounded Drift condition and thus run faster or slower; the algorithm requires $n > 2f$. Later, we discuss how to modify the algorithm to handle identical Byzantine failures. Finally, the simulation of identical Byzantine failures in a totally asynchronous system, from Chapter 12, is used. The latter simulation requires $n > 3f$, matching the bound proved in Section 13.2.

13.3.1 Timing Failures

The algorithm proceeds in synchronization *epochs*. A processor starts the kth synchronization epoch by broadcasting a message of the form $\langle k \rangle$, when the value of its adjusted clock is $k \cdot P$, for some constant P that will be specified below. P will be chosen to ensure that the start of the $(k+1)$st synchronization epoch is still in the future, according to the newly adjusted clock. When a processor receives $f+1$ messages of the form $\langle k \rangle$, it sets its adjusted clock to be $k \cdot P + x$. The value of x will be specified and explained shortly; its value will ensure that the adjusted clocks are never set backwards.

We assume that the adjusted clocks are initialized so that at time 0, the adjusted clock of every nonfaulty processor is between x and $x + d(1+\rho)$. Thus the initial closeness of synchronization must be at most $d(1+\rho)$ in every admissible execution.

Assume we have picked $P > x + d(1+\rho)$.

The pseudocode appears in Algorithm 40; it uses the basic reliable broadcast algorithm of Chapter 8 (Algorithm 23).

Algorithm 40 A clock synchronization algorithm for drift and timing failures: code for processor p_i, $0 \leq i \leq n-1$.

Initially $k = 1$ and $count[r] = 0$, for all r

when $AC = k \cdot P$ // time for kth synchronization epoch
1: bc-send($\langle k \rangle$,reliable)

when bc-recv($\langle r \rangle$,j,reliable) occurs
2: $count[r] := count[r] + 1$
3: if $count[k] \geq f + 1$ then
4: $AC := k \cdot P + x$ // modify *adj* to accomplish this
5: $k := k + 1$

Because $P > x + d(1 + \rho)$ and k is initialized to 1, the nonfaulty processors' adjusted clocks at the beginning of an admissible execution have not yet reached the time to perform Line 1.

To prove that the algorithm satisfies agreement and validity with some constants ϵ and γ, which will be determined below, we look at the times processors broadcast their $\langle k \rangle$ messages.

The following real times are defined for all $k \geq 1$:

- $ready_k$ denotes the first time a nonfaulty processor broadcasts a $\langle k \rangle$ message, starting the kth synchronization epoch
- $begin_k$ denotes the first time a nonfaulty processor evaluates the condition in Line 3 to be true and sets its adjusted clock to $k \cdot P + x$ in Line 4
- end_k denotes the last time a nonfaulty processor evaluates the condition in Line 3 to be true and sets its adjusted clock to $k \cdot P + x$ in Line 4.

Let $end_0 = 0$ (see Fig. 13.6).

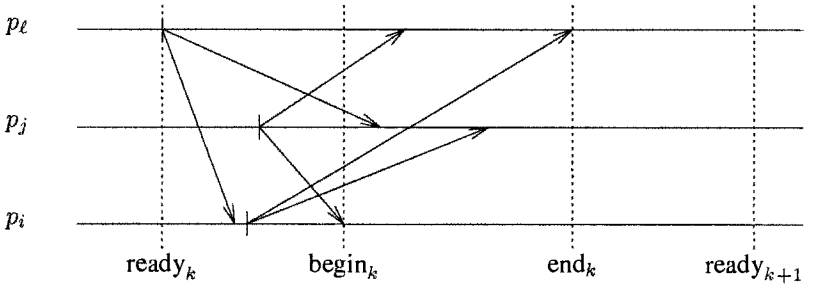

Fig. 13.6 Synchronization epochs.

The Faulty and Nonfaulty Liveness properties of reliable broadcast ensure that if one nonfaulty processor receives $f+1$ messages of the form $\langle k \rangle$, then eventually every nonfaulty processor will receive those $f+1$ messages of the form $\langle k \rangle$. Because the liveness properties are ensured in the simulation of reliable broadcast (Algorithm 23) by relaying messages immediately, those $f+1$ messages of the form $\langle k \rangle$ will be received within d time of when the first nonfaulty processor receives them. Thus we have the following lemma:

Lemma 13.5 *For all* $k \geq 1$, $end_k \leq begin_k + d$.

Let p_i be the first nonfaulty processor to start its kth synchronization epoch, and set its adjusted clock to $k \cdot P + x$. By Lemma 13.5, all nonfaulty processors set their adjusted clock to the same value at most d time later. During this time p_i's adjusted clock gains at most $d(1+\rho)$, implying the following lemma:

Lemma 13.6 *For all* $k \geq 1$ *and for any pair of nonfaulty processors,* p_i *and* p_j, $|AC_i(end_k) - AC_j(end_k)| \leq d(1+\rho)$.

Note that the above lemma is true by the initialization assumption for $k = 0$.

A nonfaulty processor broadcasts $\langle k \rangle$ at time $k \cdot P$ on its adjusted clock, that is, $P - x$ time on its clock after it has started the $(k-1)$st epoch. Thus all nonfaulty processors broadcast $\langle k \rangle$ messages at most $(P-x)(1+\rho)$ real time after end_{k-1}. Because $n > 2f$, all nonfaulty processors will get $f+1$ messages of the form $\langle k \rangle$ at most d real time later, and will start their next synchronization epoch. Therefore, we have the following lemma:

Lemma 13.7 *For all* $k \geq 1$, $end_k \leq end_{k-1} + (P-x)(1+\rho) + d$.

Because of the lower bound on P, we can prove that the start of the next synchronization epoch is still in the future:

Lemma 13.8 *For all* $k \geq 1$, $end_{k-1} < ready_k \leq begin_k$.

Proof. By Lemma 13.6, the maximum value of a nonfaulty adjusted clock at end_{k-1} is $(k-1)P + x + d(1+\rho)$, which, by the constraint on P, is less than $(k-1)P + P = kP$. Since $ready_k$ is the earliest real time when a nonfaulty adjusted clock reaches kP, $end_{k-1} < ready_k$.

To prove the second inequality, note that a nonfaulty processor starts the kth epoch only after receiving $\langle k \rangle$ messages from at least $f+1$ processors, at least one of which is nonfaulty. □

Choose $x = \rho P(2+\rho) + 2d$.

Together with the lower bound on P stated above, this implies that

$$P > (3d + \rho d)(1 - 2\rho - \rho^2)^{-1}$$

Lemma 13.9 shows that the adjusted clock of a nonfaulty processor is never set backwards, because x is chosen large enough.

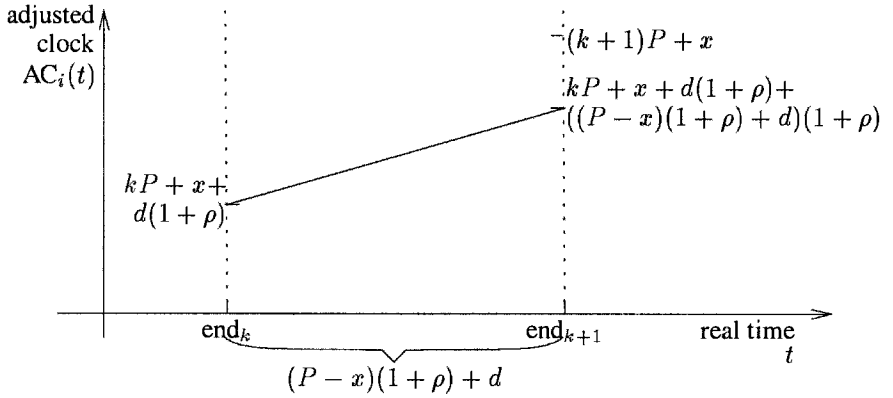

Fig. 13.7 p_i's adjusted clock is not set backwards.

Lemma 13.9 *For each nonfaulty processor p_i, $AC_i(t)$ is a nondecreasing function of t.*

Proof. The only situation in which this lemma might be violated is in the execution of Line 5. Let t be the real time at which p_i executes Line 5 to begin epoch $k+1$. The value of processor p_i's epoch k adjusted clock at time t is maximized if it has the maximum value $kP+x+d(1+\rho)$ at time end_k, p_i runs at the maximum rate $1+\rho$ until time end_{k+1}, and end_{k+1} occurs as late as possible (see Fig. 13.7). Thus the maximum value of p_i's epoch k adjusted clock at time t is $kP+x+((P-x)(1+\rho)+2d)(1+\rho)$. At time t, p_i's epoch $k+1$ clock is set to $(k+1)P+x$.

To check that

$$kP + x + ((P-x)(1+\rho) + 2d)(1+\rho) \leq (k+1)P + x$$

it is sufficient to show that

$$((P-x)(1+\rho) + 2d)(1+\rho) \leq P$$

By the choice of x, $((P-x)(1+\rho)+2d)(1+\rho)$ is equal to

$$((P - (\rho P(2+\rho) + 2d))(1+\rho) + 2d)(1+\rho)$$

which is less than

$$((P - (\rho P(2+\rho)(1+\rho)^{-2} + 2d))(1+\rho) + 2d)(1+\rho)$$

which is less than P. □

We now prove clock agreement, for $\epsilon = d(1+\rho) + x(1+\rho) - d$. Given the definition of x, ϵ is equal to $2d + 3\rho d + 2\rho P$ plus terms of order ρ^2. (When ρ has

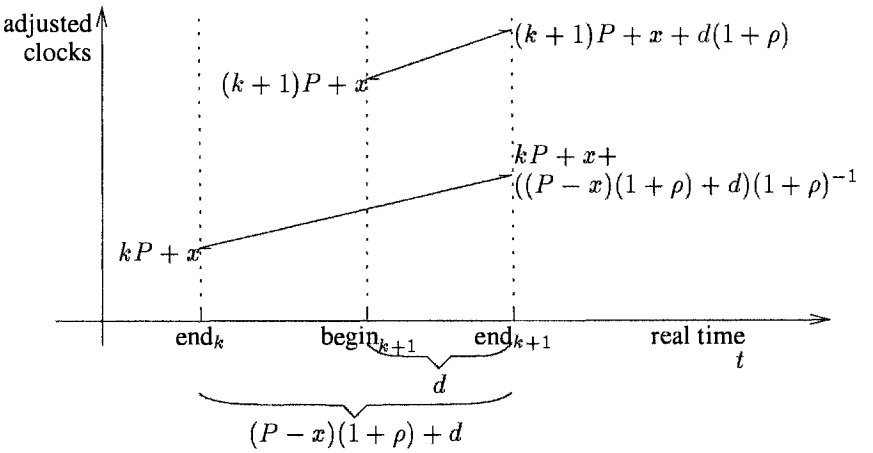

Fig. 13.8 Proof of Lemma 13.10: Epoch k clock and epoch $k+1$ clock.

a typical value of 10^{-6}, terms of order ρ^2 are negligible.) To minimize ϵ, P must be chosen to be as small as possible; in this case, ϵ is slightly more than $2d + 9\rho d$ plus terms of order ρ^2, when ρ is small. However, there are some disadvantages associated with making P very small—in particular, more resources are taken up by the clock synchronization procedure, because resynchronization messages are sent more often.

Lemma 13.10 (clock agreement) *For any time $t \geq 0$ and any two nonfaulty processors p_i and p_j, $|AC_i(t) - AC_j(t)| \leq \epsilon$.*

Proof. We partition time into intervals between *end* points and prove the lemma for any time t, $0 \leq t \leq end_k$, by induction on k.

Since clocks are initially synchronized, the lemma holds for the base case, $k = 0$, that is, for time 0. So, assume the lemma holds for k, that is, for any time t, $0 \leq t \leq end_k$.

First, we consider the case when both processors, p_i and p_j, have already done the $(k+1)$st synchronization; that is, AC_i and AC_j are both epoch $k+1$ clocks. By Lemma 13.6, their difference is at most $d(1+\rho)$, which is less than ϵ.

Next we consider the case when one processor, say, p_j, has already done the $(k+1)$st synchronization, but the other, p_i, has not; that is, AC_i is an epoch k clock and AC_j is an epoch $k+1$ clock. Since clocks are never set backward by Lemma 13.9, the difference between them is maximized at real time end_{k+1} if AC_j is set to $(k+1)P + x$ at $begin_{k+1}$ and has the maximum rate, AC_i is set to $kP + x$ at end_k and has the minimum rate, and end_{k+1} occurs as late as possible (see Fig. 13.8). Exercise 13.6 asks you to verify that the difference between the clocks is at most ϵ.

Last we consider the case when both processors, p_i and p_j, have yet not done the $(k+1)$st synchronization; that is, AC_i and AC_j are both epoch k clocks. They are

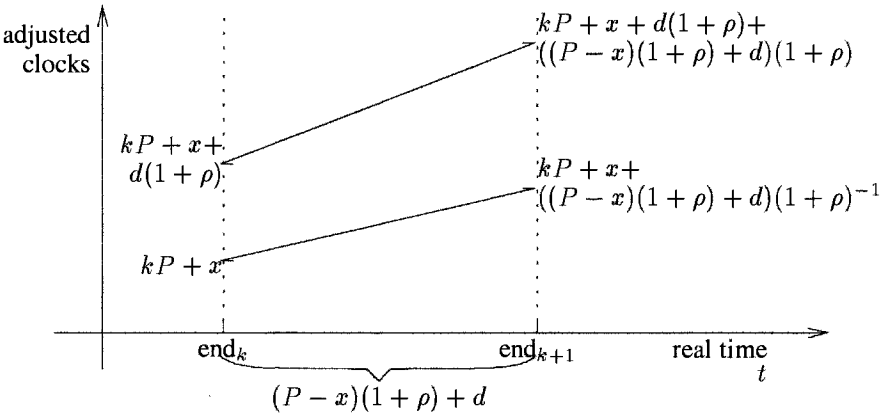

Fig. 13.9 Proof of Lemma 13.10: Epoch k clocks.

maximally far apart at real time end_{k+1} if AC_i has the minimum value $kP + x$ at real time end_k and has the minimum rate $(1+\rho)^{-1}$, AC_j has the maximum value $kP + x + d(1+\rho)$ at end_k and has the maximum rate $(1+\rho)$, and end_{k+1} occurs as late as possible (see Figure 13.9). Exercise 13.7 asks you to verify that the difference between the clocks is at most ϵ. □

We now show the clock validity condition, namely, that the adjusted clocks do not deviate too much from the hardware clocks. Let $\gamma = P(1+\rho)^2(P-x)^{-1} - 1$. Simple algebraic manipulations show that

$$\gamma = \frac{1}{c} - 1 + 2\rho + \rho^2$$

where $c = 1 - \rho(2+\rho) - 2d/P$. In the common case when ρ is small and P is large relative to d, c is slightly less than 1, and thus $\frac{1}{c} - 1$ is close to 0. Because ρ^2 is also extremely small, γ is approximately 2ρ. The rate of the adjusted clocks with respect to real time is approximately $(1+2\rho)(1+\rho)$, which is roughly $1 + 3\rho$, ignoring terms of order ρ^2. Thus the drift of the adjusted clocks is roughly three times that of the hardware clocks.

Lemma 13.11 (clock validity) *For all times t and every nonfaulty processor p_i,*

$$(1+\gamma)^{-1}(HC_i(t) - HC_i(0)) \leq AC_i(t) - AC_i(0) \leq (1+\gamma)(HC_i(t) - HC_i(0))$$

Proof. AC_i runs at the same rate as HC_i except for when Line 5 is executed.

By Lemma 13.9, AC_i is never set backwards. Thus the change in AC_i during the real time interval $[0, t]$ is always at least equal to the change in the corresponding hardware clock HC_i, and clearly

$$(1+\gamma)^{-1}(HC_i(t) - HC_i(0)) \leq AC_i(t) - AC_i(0)$$

(This condition holds for any $\gamma > -1$.)

We now have to consider how much faster AC_i can go than HC_i. Consider the real time t when AC_i is set to $kP + x$, for any k. The change in AC_i during the real time interval $[0, t]$ is at most $kP + x - x = kP$. That is,

$$AC_i(t) - AC_i(0) \leq kP$$

The change in HC_i during the same real time interval is minimized if we assume that each of the k resynchronization epochs takes the minimum amount of real time, $(P - x)(1 + \rho)^{-1}$, and HC_i has the slowest rate, $(1 + \rho)^{-1}$. Thus the change in HC_i is at least $k(P - x)(1 + \rho)^{-2}$. That is,

$$k(P - x)(1 + \rho)^{-2} \leq HC_i(t) - HC_i(0)$$

Therefore, to show that

$$AC_i(t) - AC_i(0) \leq (1 + \gamma)(HC_i(t) - HC_i(0))$$

it suffices to show that,

$$kP \leq (1 + \gamma)k(P - x)(1 + \rho)^{-2}$$

which follows by simple algebraic manipulations (see Exercise 13.8). □

To summarize, we have:

Theorem 13.12 *Algorithm 40 satisfies the clock agreement and clock validity conditions with*

$$\epsilon = d(1 + \rho) + (\rho P(2 + \rho) + 2d)(1 + \rho) - d$$

and

$$\gamma = P(1 + \rho)^2(P - (\rho P(2 + \rho) + 2d))^{-1} - 1$$

as long as $P > (3d + \rho d)(1 - 2\rho - \rho^2)^{-1}$, *in the presence of* $f < n/2$ *timing failures.*

13.3.2 Byzantine Failures

In the algorithm just presented, all communication between processors is via the reliable broadcast primitive. The particular simulation that we assumed in Section 13.3.1, Algorithm 23, works for asynchronous crash failures, and thus it works for timing failures. However, it does not work for Byzantine failures.

To get a clock synchronization algorithm that tolerates Byzantine failures, we can develop a simulation of reliable broadcast that tolerates Byzantine failures and use it in Algorithm 40. We will develop this simulation in two steps (a similar development appears in Chapter 14, for the randomized consensus algorithm in the presence of Byzantine failures).

First, to go from identical Byzantine failures to the timing failure model, we can employ a validation procedure as in Chapter 12. The validation makes the messages

longer and causes some additional local computation, but these changes do not affect the performance analysis of the clock synchronization algorithm.

Second, to go from Byzantine failures to identical Byzantine failures we use the asynchronous simulation (Algorithm 39) from Chapter 12. This simulation affects the performance of the clock synchronization algorithm in two ways. One impact is that the number of processors, n, must now be greater than $3f$, instead of only greater than $2f$. The other impact is that in the definitions of ϵ and γ for the clock agreement and validity conditions, every occurrence of d must be replaced by $3d$. The reason is that three "rounds" of communication are required in Algorithm 39 for each simulated "round."

13.4 PRACTICAL CLOCK SYNCHRONIZATION: IDENTIFYING FAULTY CLOCKS

The Internet standard Network Time Protocol (NTP) provides synchronized clocks. Certain nodes in the system are identified as time servers. Time servers are classified as primary or secondary — primaries get their time from a reference source, such as a satellite, whereas secondaries get their clock times from either a primary or other secondaries. Time servers are organized conceptually into a hierarchy based on how many hops they are away from a primary server; this distance is called the *stratum* of the server. When a path goes down between two time servers, the strata, or shortest path distances from a primary, are recalculated.

Time servers exchange timestamped messages periodically in order to estimate round trip delays, clock offsets, and error (cf. Section 6.3.6); filtering is applied to reduce timing noise. When a node wants to update its local clock, it selects an appropriate subset of its neighboring time servers; the choice of subset is made by an algorithm that has been carefully optimized on the basis of experimental data and depends on the strata of the neighbors, among other things. Finally, the offsets estimated for this subset are combined to calculate a new value for the local clock. The algorithms in NTP have been carefully tailored to work well with the statistical behavior of links in the Internet.

One of the interesting algorithms in NTP is that for choosing which set of clock values will be combined to compute a clock update. Like the algorithm presented in this chapter, the possibility of Byzantine failures (i.e., processors with arbitrarily faulty clocks) is considered. When a processor p_i obtains an estimate of the difference between its clock and that of another processor p_j, it actually gets an *interval* in which the difference lies. Given a set of m such time intervals, up to f of which might represent values of faulty clocks, the processor must choose a subset of time intervals to use in the combining step. The time intervals that are discarded when the subset is chosen should be those that have (relatively) bad data, that is, that came from processors with faulty clocks. The assumption is that nonfaulty clocks are close to real time, and thus their time intervals will be relatively close to each other. As a result, one way to identify bad time intervals is to see which ones are not sufficiently

close to enough other time intervals. In particular, the NTP algorithm finds the smallest interval I that contains the midpoint of at least $m - f$ time intervals. The time intervals that intersect the interval I are then used in the combining stage.

Exercises

13.1 Prove that the rate at which two hardware clocks of nonfaulty processors drift from each other is $\rho(2+\rho)(1+\rho)^{-1}$; that is, prove:

$$\max_{i,j} \frac{|HC_i(\Delta t) - HC_j(\Delta t)|}{\Delta t} \leq \rho(2+\rho)(1+\rho)^{-1}$$

13.2 Show that if the hardware clock is within a linear envelope of real time and the adjusted clock is within a linear envelope of the hardware clock, then the adjusted clock is within a linear envelope of real time.

Calculate the validity parameter of the adjusted clocks with respect to real time, as a function of ρ and γ.

13.3 What happens to Theorem 13.3 if there is no drift?

What happens to the result? That is, if there is no drift but there are Byzantine failures, do we need $n > 3f$ to keep the adjusted clocks synchronized?

13.4 Prove that if $\alpha' = \text{scale}(\alpha, s)$, then $\alpha = \text{scale}(\alpha', \frac{1}{s})$.

13.5 In the text it was shown that clock synchronization is impossible for three processors, one of which can be Byzantine. Extend this result to show that n must be larger than $3f$, for any value of f (not just $f = 1$).

Hint: Do not try to modify the previous proof. Instead use a reduction, as in the proof of Theorem 5.8.

13.6 Complete the second case in the proof of Lemma 13.10 by showing that the difference between a nonfaulty epoch k clock and a nonfaulty epoch $k+1$ clock is never more than ϵ.

13.7 Complete the third case in the proof of Lemma 13.10 by showing that the difference between two nonfaulty epoch $k+1$ clocks is never more than ϵ.

13.8 Complete the algebra in the proof of Lemma 13.11 by showing that

$$kP \leq (1+\gamma)k(P-x)(1+\rho)^{-2}$$

13.9 Show that the reliable broadcast simulation of Algorithm 23 does not work in the presence of Byzantine failures.

13.10 Work out the details sketched at the end of Section 13.3.2 for handling Byzantine failures in the clock synchronization algorithm and the effect on the performance.

Try to find ways to reduce the cost of validating the messages of this algorithm.

Chapter Notes

The problem of synchronizing clocks in the presence of Byzantine faults was first posed by Lamport and Melliar-Smith [162]. The lower bound on the number of processors, shown in Section 13.2, was first proved by Dolev, Halpern and Strong [94]. The proof presented here was developed by Fischer, Lynch and Merritt [109], who also proved that the connectivity of the topology graph must be at least $2f + 1$.

The clock synchronization algorithm presented in Section 13.3 is based on an algorithm of Srikanth and Toueg [245], which uses authenticated broadcast. It is similar in flavor to the algorithm of Dolev et al. [93]. Srikanth and Toueg also prove that that the clock validity parameter, γ, must be larger than or equal to hardware clocks' drift bound, ρ; they show how to modify the algorithm to get $\gamma = \rho$. The clock agreement parameter, ϵ, obtained in the algorithm of Dolev et al. is smaller than the parameter obtained in the algorithm of Srikanth and Toueg, presented here.

Welch and Lynch [260] and Mahaney and Schneider [178] designed algorithms based on approximate agreement, a problem defined in Chapter 16. In their seminal paper, Lamport and Melliar-Smith [162] presented algorithms based on using algorithms for consensus.

For some applications, discontinuities in the adjusted clock are undesirable; for instance, jobs that are supposed to begin automatically at certain times might be skipped when the clock is set forward. If adjusted clocks can be set backwards, as is the case in some algorithms, then some activities might be done twice, or a later event might be timestamped before an earlier one. These problems can be eliminated by amortizing the necessary adjustment over an interval of time, as suggested by Lamport and Melliar-Smith [162].

Solutions to the problem of achieving synchronization initially in the presence of faults are presented in [93, 245, 260]. Some of the literature on clock synchronization is described in the survey of Simons, Welch, and Lynch [242]. Other papers appear in the collection edited by Yang and Marsland [263].

The Network Time Protocol, described in Section 13.4, was developed by Mills [186, 187]. Marzullo and Owicki [180] proposed the idea of intersecting time intervals to discard faulty clock values.

Distributed Computing: Fundamentals, Simulations and Advanced Topics, Second Edition
Hagit Attiya and Jennifer Welch
 ISBN: 0-471-45324-2

Part III

Advanced Topics

Distributed Computing: Fundamentals, Simulations and Advanced Topics, Second Edition
Hagit Attiya and Jennifer Welch
 ISBN: 0-471-45324-2

14

Randomization

Previous chapters concentrated on specific problems (Part I) or on simulations between specific models of computation (Part II); this chapter concentrates on a specific type of distributed algorithms, which employ *randomization*. Randomization has proved to be a very powerful tool for designing distributed algorithms (as for many other areas). Randomization often simplifies algorithms and, more importantly, allows us to solve problems in situations where they cannot be solved by deterministic algorithms, or with fewer resources than the best deterministic algorithm.

This chapter extends the formal model to include randomization and describes randomized algorithms for three basic problems: leader election, mutual exclusion, and consensus.

For all three problems, randomization allows us to overcome impossibility results and lower bounds, by relaxing the termination conditions or the individual liveness properties (in the case of mutual exclusion).

14.1 LEADER ELECTION: A CASE STUDY

This section has the dual purpose of demonstrating a simple but powerful application of randomization and developing the formal definitions relating to randomization.

14.1.1 Weakening the Problem Definition

A *randomized* algorithm is an algorithm that has access to some source of random information, such as that provided by flipping a coin or rolling dice. More formally,

we extend the transition function of a processor to take as an additional input a random number, drawn from a bounded range under some fixed distribution. The assumption of a fixed distribution suffices for all algorithms we present in this chapter. Many other probability distributions can be implemented using this type of coin (by appropriate mappings).

The addition of random information alone typically will not affect the existence of impossibility results or worst-case bounds. For instance, even if processors have access to random numbers, they will not be able to elect a leader in an anonymous ring or solve consensus in fewer than $f + 1$ rounds in *all* (admissible) executions.

However, randomization in conjunction with a judicious *weakening* of the problem statement is a powerful tool for overcoming limitations. Usually the weakening involves the termination condition (for instance, a leader must be elected with a certain probability) while the other conditions are not changed (for instance, it should never be the case that two leaders are elected).

Randomization differs from average case analysis of a deterministic algorithm. In average case analysis, there are several choices as to what is being averaged over. One natural choice is the inputs. (Other possibilities in systems that have some degree of uncertainty are the interleavings of processor steps, the message delays, and the occurrences of failures.) There are two difficulties with this approach. One is that determining an accurate probability distribution on the inputs (not to mention the processor scheduling, the message delays, or the failure events) is often not practical. Another drawback is that, even if such distributions can be chosen with some degree of confidence, very little is guaranteed about the behavior of the algorithm on a particular input. For instance, even if the *average* running time over all inputs is determined to be small, there still could be some inputs for which the running time is enormous.

In the randomized approach, more stringent guarantees can be made. Because the random numbers introduce another dimension of variability even for the same inputs, there are many different executions for the same input. A good randomized algorithm will guarantee good performance with some probability for each individual input. Typically the performance of a randomized algorithm is defined to be the *worst-case* probability over all inputs.

The simplest use of randomization is to create initial asymmetry in situations that are inherently symmetric. One such situation is anonymous rings (studied in Chapter 3). Recall that in anonymous rings, where processors do not have distinct identifiers, it is impossible to elect a unique leader (Theorem 3.2). This impossibility result holds even for randomized algorithms. However, a randomized algorithm *can* ensure that a leader is elected *with some probability*. Thus we can solve a variant of the leader election problem that relaxes the condition that eventually a leader must be elected in every admissible execution.

The relaxed version of the leader election problem requires:

Safety: In every configuration of every admissible execution, at most one processor is in an elected state.

Liveness: At least one processor is elected with some nonzero probability.

The safety property has to hold with certainty; that is, the algorithm should never elect two leaders. The liveness condition is relaxed, and the algorithm need not *always* terminate with a leader, rather, it is required to do so *with nonzero probability*. (We will spend some time exploring exactly how to define this probabilistic condition.)

An algorithm that satisfies this weakened liveness condition can fail to elect a leader either by not terminating at all or by terminating without a leader. As demonstrated below, these two ways to express liveness are typically related, and one can be traded off against the other.

14.1.2 Synchronous One-Shot Algorithm

First, let us consider synchronous rings. There is only one admissible execution on an anonymous synchronous ring for a deterministic algorithm. For a randomized algorithm, however, there can be many different executions, depending on the random choices.

The approach we will use to devising a randomized leader election algorithm is to use randomization to create asymmetry by having processors choose random *pseudo*-identifiers, drawn from some range, and then execute a deterministic leader election algorithm.

Not every deterministic leader election algorithm can be employed. Regardless of the method used for generating the pseudo-identifiers, there is always a chance that they are not distinct. The deterministic leader election algorithm must guarantee that at most a single processor terminates as a leader even in this case. Also, it is important that the algorithm does not freeze when all processors choose the same pseudo-identifier. Finally, it is helpful if the deterministic leader election algorithm detects that no leader was elected.

A simple deterministic leader election algorithm with these properties is the following. Each processor sends a message around the ring to collect all pseudo-identifiers. When the message returns (after collecting n pseudo-identifiers), a processor knows whether it is a unique maximum or not.

The pseudocode appears in Algorithm 41.

In this algorithm, the pseudo-identifier is chosen to be 2 with probability $1/n$ and 1 with probability $(1 - 1/n)$, where n is the number of processors on the ring. Thus each processor makes use of its source of randomness exactly once, and the random numbers are drawn from the range $[1..2]$.

The set of all possible admissible executions of this algorithm for fixed ring size n contains exactly one execution for each element of the set $\mathcal{R} = \{1, 2\}^n$. That is, by specifying which random number, 1 or 2, is obtained by each of the n processors in its first step, we have completely determined the execution. Given an element R of $\mathcal{R}$, we will denote the corresponding execution by $exec(R)$.

We would like to make some claims about the probabilistic behavior of this algorithm. These claims reduce to claims about the random choices.

Algorithm 41 Randomized leader election in an anonymous ring: code for processor p_i, $0 \leq i \leq n-1$.

```
1:  initially                                   // spontaneously or upon receiving the first message
2:      id_i := { 1  with probability 1 - 1/n
                { 2  with probability 1/n       // choose pseudo-identifier
3:      send ⟨id_i⟩ to left

4:  upon receiving ⟨S⟩ from right
5:      if |S| = n then                         // your message is back
6:          if id_i is the unique maximum of S then become elected   // the leader
7:          else become non-elected                                  // a nonleader
8:      else                                    // concatenate your id to the message and forward
9:          send ⟨S · id_i⟩ to left
```

Let P be some predicate on executions, for example, at least one leader is elected. Then $\Pr[P]$ is the probability of the event

$$\{R \in \mathcal{R} : exec(R) \text{ satisfies } P\}$$

When does the randomized leader election algorithm terminate with a leader? This happens when a single processor has the maximum identifier, 2. The probability that a single processor draws 2 is the probability that $n-1$ processors draw 1, and one processor draws 2, times the number of possible choices for the processor drawing 2, that is,

$$\binom{n}{1} \frac{1}{n} \left(1 - \frac{1}{n}\right)^{n-1} = \left(1 - \frac{1}{n}\right)^{n-1} = c$$

The probability c is greater than $(1 - \frac{1}{n})^n$, which converges from above to e^{-1} as n increases, where e is the constant $\approx 2.71\ldots$.

It is simple to show that every processor terminates after sending exactly n messages (Exercise 14.1); moreover, at most one processor terminates in an elected state (Exercise 14.2). In some executions, for example, when two processors choose the pseudo-identifier 2, no processor terminates as a leader. However, the above analysis shows that this happens with probability less than $1 - 1/e$. We have shown the following theorem:

Theorem 14.1 *There is a randomized algorithm that, with probability $c > 1/e$, elects a leader in a synchronous ring; the algorithm sends $O(n^2)$ messages.*

14.1.3 Synchronous Iterated Algorithm and Expectation

It is pleasing that the probability of termination in Algorithm 41 does not decrease with the ring size n. However, we may wish to increase the probability of termination, at the expense of more time and messages.

The algorithm can be modified so that each processor receiving a message with n pseudo-identifiers checks whether a unique leader exists (in Lines 5–7). If not, the processor chooses a new pseudo-identifier and iterates the algorithm. We will show that this approach amplifies the probability of success.

We can either have the algorithm terminate after some number of iterations and experience a nonzero probability of not electing a leader or iterate until a leader is found and risk that the algorithm does not terminate.

Let us consider the second option in more detail. In order to repeat Algorithm 41, each processor will need to access the random number source multiple times, in fact, potentially infinitely often. To completely specify an execution of the algorithm, we will need to specify, for each processor, the *sequence* of random numbers that it obtains. Thus $\mathcal{R}$ now becomes the set of all n-tuples, each element of which is a possibly infinite sequence over $\{1, 2\}$.

For the iterated algorithm, the probability that the algorithm terminates at the end of the kth iteration is equal to the probability that the algorithm fails to terminate in the first $k-1$ iterations and succeeds in terminating in the kth iteration. The analysis in Section 14.1.2 shows that the probability of success in a single iteration is $c > 1/e$. Because the probability of success or failure in each iteration is independent, the desired probability is

$$(1-c)^{k-1}c$$

This probability tends to 0 as k tends to ∞; thus the probability that the algorithm terminates with an elected leader is 1.

We would now like to measure the time complexity of the iterated algorithm. Clearly, a worst-case analysis is not informative, since in the worst case the required number of iterations is infinite. Instead, we will measure the *expected* number of iterations. The number of iterations until termination is a geometric random variable whose expected value is $c^{-1} < e$. Thus the expected number of iterations is less than three.

In general, the expected value of any complexity measure is defined as follows. Let T be a random variable that, for a given execution, is the value of the complexity measure of interest for that run (for instance, the number of iterations until termination). Let $\mathrm{E}[T]$ be the expected value of T, taken over all $R \in \mathcal{R}$. That is,

$$\mathrm{E}[T] = \sum_{x \text{ is a value of } T} x \cdot \Pr[T = x]$$

Note that by the definition of $\Pr[P]$ above, this is ultimately taking probabilities over $\mathcal{R}$.

With this definition, we have the following theorem:

Theorem 14.2 *There is a randomized algorithm that elects a leader in a synchronous ring with probability 1 in $(1/c) \cdot n < e \cdot n$ expected rounds; the algorithm sends $O(n^2)$ expected messages.*

14.1.4 Asynchronous Systems and Adversaries

Now suppose we would like to find a randomized leader election algorithm for asynchronous anonymous rings. Even without the random choices, there are many executions of the algorithm, depending on when processors take steps and when messages arrive. To be able to calculate probabilities, we need a way to factor out the variations due to causes other than the random choices. That is, we need a way to group the executions of interest so that each group differs only in the random choices; then we can perform probabilistic calculations separately for each group and then combine those results in some fashion.

The concept used to account for all the variability other than the random choices is an adversary. An *adversary* is a function that takes an execution segment and returns the next event that is to occur, namely, which processor receives which pending messages in the next step. The adversary must satisfy the admissibility conditions for the asynchronous message-passing communication system: Every processor must take an infinite number of steps and every message must eventually be received. The adversary can also control the occurrence of failures in the execution, subject to the relevant admissibility conditions.

An execution of a specific anonymous leader election algorithm is uniquely determined by an adversary $\mathcal{A}$ and an element $R \in \mathcal{R}$; it is denoted $exec(\mathcal{A}, R)$.

We need to generalize the definitions of $\Pr[P]$ and $\mathrm{E}[T]$ from the previous subsections. Given a particular adversary $\mathcal{A}$, $\Pr_{\mathcal{A}}[P]$ is the probability of the event $\{R \in \mathcal{R} : exec(\mathcal{A}, R)$ satisfies $P\}$. Similarly,

$$\mathrm{E}_{\mathcal{A}}[T] = \sum_{x \text{ is a value of } T} x \cdot \Pr_{\mathcal{A}}[T = x]$$

The performance of the algorithm overall is taken to be the worst over all possible adversaries. Thus the liveness condition for leader election is that there is a nonzero probability of termination, for every adversary:

Liveness: There exists $c > 0$ such that $\Pr_{\mathcal{A}}[\text{a leader is elected}] \geq c$, for every (admissible) adversary $\mathcal{A}$.

Similarly, the expected number of iterations until termination taken by the algorithm over all possible adversaries is defined to be

$$\mathrm{E}[T] = \max_{\mathcal{A}} \mathrm{E}_{\mathcal{A}}[T]$$

where T is the number of iterations. $\mathrm{E}[T]$ is a "worst-case average."

Exercise 14.4 asks you to verify that both the one-shot and the iterated synchronous leader election algorithms have the same performance in the asynchronous case as they do in the synchronous case. The reason is that the adversaries actually cannot affect the termination of these algorithms. Once the pseudo-identifiers are chosen, the ability of the algorithm to terminate in that iteration is completely unaffected by the message delays or processor step times.

However, in most situations the adversary can have a great deal of impact on the workings of a randomized algorithm. As we shall see below, the interaction among the adversary, the algorithm, and the random choices can be extremely delicate.

Sometimes it is necessary to restrict the power of the adversary in order to be able to solve the problem. Formally, this is done by defining the input to the function modeling the adversary to be something less than the entire execution so far.

For example, in message-passing systems, a *weak* adversary cannot look at the contents of messages; it is defined as a function that takes as input the message pattern (indicating who sent messages to whom so far and when they were received, but not including the contents of the messages).

In shared memory systems, where the output of the adversary is just which processor takes the next step, a *weak* adversary cannot observe the local states of processors or the contents of the shared memory. (Such an adversary is sometimes called *oblivious*.)

14.1.5 Impossibility of Uniform Algorithms

The randomized leader election algorithms considered so far have all been nonuniform, that is, they depend on n, the number of processors in the ring. Theorem 14.3 states that knowing n is necessary for electing a leader in an anonymous ring, even for a randomized algorithm. In fact, this impossibility result holds even if the algorithm only needs to guarantee termination in a single situation.

Theorem 14.3 *There is no uniform randomized algorithm for leader election in a synchronous anonymous ring that terminates in even a single execution for a single ring size.*

We only sketch the proof of this theorem. Assume there exists such a uniform randomized leader election algorithm A. Consider some execution, α, of A on an anonymous ring with n processors, $q_0, \ldots, q_{n-1}$, in which processor q_l is elected as leader. Such an execution exists by assumption. Note that (the description of) the execution includes the outcomes of the random choices of the processors.

Now consider another execution, β, of A on an anonymous ring with $2n$ processors, $p_0, \ldots, p_{2n-1}$, which is a "doubling" of α. That is, for any $i = 0, \ldots, n-1$, the events of p_i and p_{n+i} in β are the events of q_i in α, including the random choices, with the same interleaving (see Fig. 14.1, for $n = 4$). The execution β is possible because processors are anonymous, so we can have p_i and p_{n+i} take the same steps. However, in β, both p_l and p_{n+l} are elected as leaders. This violates the requirement that only a single leader is elected.

14.1.6 Summary of Probabilistic Definitions

In this subsection, we finish generalizing our definitions and summarize them all.

Our running example so far has been leader election in anonymous rings. Because any anonymous ring algorithm has a single initial configuration, we have seen no

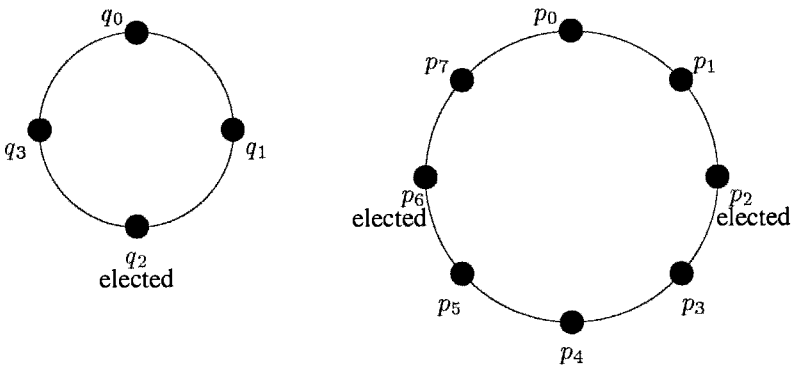

Fig. 14.1 Illustration for the proof of Theorem 14.3; $n = 4$.

impact of the initial configuration on the behavior of randomized algorithm. In the general case, though, an algorithm can have multiple initial configurations. The initial configuration is a third source of variability in the behavior of a randomized algorithm, in addition to the random choices and the adversary.

An execution of a specific algorithm is uniquely determined by an adversary $\mathcal{A}$, an initial configuration C_0, and an element $R \in \mathcal{R}$; it is denoted $exec(\mathcal{A}, C_0, R)$. ($\mathcal{R}$ is determined by the domain from which the random choices of the algorithm are drawn.)

Let P be some predicate on executions. For fixed adversary $\mathcal{A}$ and initial configuration C_0, $\Pr_{\mathcal{A},C_0}[P]$ is the probability of the event $\{R \in \mathcal{R} : exec(\mathcal{A}, C_0, R) \text{ satisfies } P\}$. $\mathcal{A}$ and C_0 may be omitted when they are clear from the context.

The expected value of any complexity measure is defined as follows. Let T be a random variable that, for a given execution, is the value of the complexity measure of interest for that run. For a fixed admissible adversary $\mathcal{A}$ and initial configuration C_0, let $E_{\mathcal{A},C_0}[T]$ be the expected value of T, taken over all $R \in \mathcal{R}$. That is,

$$\mathrm{E}_{\mathcal{A},C_0}[T] = \sum_{x \text{ is a value of } T} x \cdot \Pr_{\mathcal{A},C_0}[T = x]$$

Sometimes, we are interested in the probability that a complexity measure (especially the time complexity) is smaller than some quantity, that is, in $\Pr_{\mathcal{A},C_0}[T \leq x]$. In many cases (depending on the distribution of T), this quantity and the expected value of T are related.

Define the expected value of a complexity measure for an algorithm, $\mathrm{E}[T]$, to be the maximum, over all admissible adversaries $\mathcal{A}$ and initial configurations C_0, of $\mathrm{E}_{\mathcal{A},C_0}[T]$. Because the maximum is taken over all initial configurations, we get to pick the worst inputs for the algorithm for each adversary.

14.2 MUTUAL EXCLUSION WITH SMALL SHARED VARIABLES

In Chapter 4, we have seen that any deterministic mutual exclusion algorithm for n processors with k-bounded waiting requires a shared variable with at least $\Omega(\log n)$ bits (Theorem 4.4). A randomized algorithm can reduce the number of shared bits required for mutual exclusion while still being fair to all processors, by guaranteeing that a processor has probability $\Theta(1/n)$ of succeeding in each attempt to enter the critical section; the algorithm uses only a constant-size shared variable. The number of attempts is measured in terms of the number of steps a processor takes after moving into the entry section and until entering the critical section. This section outlines this algorithm and points out its intricacies, demonstrating the delicate interplay between the adversary and the algorithm's random choices.

The adversary does not have access to the contents of the shared variable or to the local states of the processors, but it can observe the interaction of the algorithm with the environment. That is, the adversary can tell which of the four sections, entry, critical, exit, or remainder, each processor is currently in.

The algorithm consists of phases, each happening between successive entries to the critical section. Each phase is partitioned into two stages, which determine which processor will enter the critical section next. While the critical section is not free, there is a *drawing* stage, during which each processor wishing to enter the critical section draws a ticket. The *winner* of the lottery in the drawing stage is the processor that draws the highest ticket; if more than one processor draws the highest lottery ticket, then the first one to draw this ticket (by the actual time of the drawing) is the winner. When the critical section becomes free, the winner of the lottery is discovered in a *notification* stage. The winning processor enters the critical section, and the drawing stage of the next phase begins.

Assume that each contending processor draws a ticket according to the lottery used in the previous section (for leader election): 1 with probability $1 - \frac{1}{n}$ and 2 with probability $\frac{1}{n}$. A calculation similar to the one in the previous section shows that the probability that a *specific* processor p_i is the only one that draws the maximal ticket is at least $\frac{c}{n}$, for the same constant $c > 0$. This fact can be used to bound the number of attempts a processor needs to execute until entering the critical section. Clearly, the range of lottery tickets is constant (independent of n), and the maximal ticket drawn so far can be kept in a shared variable of constant size.

It is possible that several processors see themselves as candidates; in particular, if 1 is drawn before 2 is drawn, then the first processor to draw 1 and the first processor to draw 2 are both candidates. The notification stage is used to pick the winner among candidates.

The argument bounding the chances of a specific processor to enter the critical section depends on having each processor draw a ticket at most once in each phase. One way to guarantee this property is to keep the phase number in the shared variable and have the winner of the phase (the processor entering the critical section) increment it. Then, a processor can limit itself to draw at most one ticket in each phase by drawing a ticket only if the current phase is larger than the previous phase

in which it participated. A processor can remember this phase number in a local variable.

The problem with this solution is that phase numbers are unbounded and cannot be kept in a constant-size shared variable. Instead, we employ a single random *phase bit*. This bit is set by the processor entering the critical section to be either 0 or 1 with equal probability. We modify the lottery scheme described above, so that only processors whose previous phase bit is not equal to the shared phase bit may draw a ticket. In the new lottery:

$$\Pr[p_i \text{ wins}] = \Pr[p_i \text{ wins} \mid p_i \text{ draws a ticket}] \cdot \Pr[p_i \text{ draws a ticket}]$$

and by the above argument,

$$\Pr[p_i \text{ wins}] \geq \frac{c}{n} \cdot \frac{1}{2}$$

for the same constant $c > 0$. Therefore, the probability that a processor p_i is the only processor that draws the maximal ticket is at least $\frac{c'}{n}$, for some constant $c' > 0$.

We can now suggest an algorithm that uses a shared read-modify-write variable with the following components:

Free: A flag indicating whether there is a processor in the critical section

Phase: The shared phase bit

Max: A two-bit variable containing the maximum ticket drawn (so far) in the current lottery: 0, 1 or 2.

In addition, each processor p_i has the following local variables:

last$_i$: The bit of the last phase the processor participated in

ticket$_i$: The last ticket the processor has drawn

The pseudocode appears in Algorithm 42. Processors use read-modify-write operations, which are executed atomically and without interference from other processors. Lines 1–10 are executed as a single atomic read-modify-write, in which a processor first decides whether to execute the drawing or the notification stage and then performs all the necessary assignments. When entering the critical section, a processor marks the critical section as not free, sets the phase bit and resets the maximal ticket for the next lottery, and initializes its own ticket (Lines 11–14).

Mutual exclusion is guaranteed by the *Free* component of the shared variable, which serves as a lock on the critical section (similarly to the shared variable V in Algorithm 7).

It may seem that there are at most two candidates in each phase (the first processor that draws 1 and the first processor that draws 2). However, because the algorithm uses only a single bit for the phase number and because losers are not always notified, there may be more than two candidates in a phase. Consider, for example, the following execution:

Algorithm 42 Mutual exclusion algorithm with small shared variables: code for processor p_i, $0 \leq i \leq n-1$.

```
⟨Entry⟩:                                                    // drawing stage
1:  if last ≠ Phase then                                    // didn't participate in this phase
2:      ticket := { 1  with probability 1 − 1/n
                  { 2  with probability 1/n                 // draw a ticket
3:  last := Phase                                           // remember this phase number
4:  if (ticket > Max) then                                  // p_i is a candidate in this phase
5:      Max := ticket
6:  else ticket := 0                                        // p_i lost
                                                            // notification stage
7:  wait until Free = true
8:  if ticket ≠ Max then                                    // p_i lost the lottery
9:      ticket := 0
10:     goto Line 1                                         // try again
                                                            // have the critical section, but clean up first
11: Free := false
12: Phase := { 0  with probability 1/2
             { 1  with probability 1/2
13: Max := 0
14: ticket := 0
⟨Critical Section⟩
⟨Exit⟩:
15: Free := true
```

In some phase with bit 0, p_i is the first to draw a 1 ticket and p_j is the first to draw a 2 ticket. In the notification stage, p_j is scheduled first, initializes a new phase, and enters the critical section. When p_j is scheduled next, the phase bit is again 0, and the critical section is free. If no processor drew a 2 ticket so far in this phase, then p_i will execute its notification stage and observes that its ticket is equal to *max*; therefore, p_i enters the critical section.

Although this kind of behavior is unpredictable, it does not violate the expected bounded waiting property. Scenarios of the type described before affect only phases in which the highest ticket drawn is 1; however, the analysis of the lottery considered only the case where the winner draws 2. Thus the behavior of a winner with a 1 ticket does not influence the above analysis.

There is, however, a possibility that the adversary can make the algorithm violate the bounded waiting property. If the adversary wishes to bias against some processor p_k, it can let p_k take steps only in phases in which there are processors with high lottery values. This decreases p_k's probability of winning the lottery and violates the bounded waiting property. The adversary could easily have done this if it had full knowledge of the execution so far, including the local states of processors and the contents of the shared register. Surprisingly, even without this knowledge, the

adversary can make deductions about information "hidden" from it; the adversary can "learn" whether a processor p_i has a high lottery value by observing the external behavior of other processors that participate with this processor in the lottery. The adversary can learn about the lottery value of a processor by a small "experiment":

The adversary waits for the critical section to be free. Then it schedules one step of processor p_i, then two steps of some other processor p_j, $j \neq i$, and then observes whether p_j enters the critical section or not. If p_j does not enter the critical section after these two steps, then p_j's lottery value must be 1 and p_i's lottery value is 2. Note that if p_j enters the critical section, the adversary waits for it to exit the critical section.

If the phase bit of p_j happens to be equal to the phase bit of p_k, then when p_k is scheduled to take its steps together with p_j, it has no chance of entering the critical section. Similar tricks can be used by the adversary to learn the exact phase bits, but this is slightly more complicated. Instead, note that the adversary can repeat the experiment, with other processors, and accumulate a set of processors, P, with lottery value 2, namely, the processors that play the role of p_i when p_j does not enter the critical section after two steps, in the scenario just described.

Note that the phase bits are random, so, with high probability, these processors have the same phase bit as p_k. Once we have this set of processors at our disposal, we can let p_k take steps always with processors with lottery value 2. Thus, p_k's probability of entering the critical section can be made very small.

This counterexample shows that arguing about the ability of an adversary is a delicate task. In this case, the adversary can deduce the values of the local variables and the shared variable, by scheduling processors and observing their interface with the environment.

This problem can be avoided by hiding the results of the lottery: The winner of the current lottery does not enter the critical section immediately, but waits for the next phase. The winner of the lottery in the previous phase enters the critical section in the current phase. In each phase, the algorithm executes the drawing stage of the current lottery, together with the notification stage of the previous lottery. Thus the behavior of the processors (as observed in the interface) does not indicate the results of the drawing stage in the current phase. The details of the implementation of this idea, as well as its correctness proof, are left out; the chapter notes discuss the relevant literature.

14.3 CONSENSUS

Our last, and perhaps most important, example of the utility of randomization in distributed computing is the consensus problem. For this problem, randomization allows us to circumvent inherent limitations—it allows us to solve consensus in asynchronous systems, and it allows us to achieve consensus in synchronous systems in fewer than $f + 1$ rounds, in the presence of f failures.

The randomized consensus problem is to achieve the following conditions:

Agreement: Nonfaulty processors do not decide on conflicting values.

Validity: If all the processors have the same input, then any value decided upon must be that common input.

Termination: All nonfaulty processors decide with some nonzero probability.

Agreement and Validity are the usual safety conditions (cf. the definition of the consensus problem for crash failures in Chapter 5). The Termination condition has been weakened; ideally, we would like to have termination with probability 1.

We present a randomized asynchronous algorithm for reaching randomized consensus that tolerates f crash failures, under the assumption $n \geq 2f + 1$. The algorithm has constant expected time complexity. This result does not contradict the impossibility result proved in Chapter 5 (Theorem 5.25), because the randomized algorithm has executions in which processors do not terminate; however, these executions occur with zero probability. In addition, if it is used in a synchronous system, the randomized algorithm is faster, on average, than the lower bound on the number of rounds proved in Chapter 5 (Theorem 5.3), for every set of inputs and pattern of failures.

The proof for Theorem 10.22 can be modified to show that $n \geq 2f + 1$ is a necessary condition for solving randomized consensus in an asynchronous systems, in the presence of f crash failures (Exercise 14.9). Thus the algorithm we present is optimal in the number of processors.

The algorithm has two parts: the first is a general phase-based voting scheme using individual processors' preferences to reach agreement (when possible), and the second is a common coin procedure used to break ties among these preferences. As described later, the general scheme can be adapted to tolerate Byzantine failures by using the asynchronous simulation of identical Byzantine failures (Algorithm 39). We also describe a simple common coin procedure that can tolerate Byzantine failures, which gives exponential expected time complexity.

Randomized consensus algorithms have been the subject of extensive research; some of the wide literature in this area is mentioned in the chapter notes.

14.3.1 The General Algorithm Scheme

The core of the algorithm is a phase-based *voting* scheme: In each phase, a processor votes on its (binary) preference for this phase by sending a message containing its preference. It calculates the outcome of the vote as the majority of all the preferences received; different processors may see different outcomes. If a processor sees a unanimous vote for some preference, it decides on this preference. In case some processors were not able to reach a decision by this point in the phase, processors exchange their outcomes from the vote. If all outcomes reported to a particular processor are the same, the processor sets its preference for the next phase to be this value; otherwise, it obtains its preference for the next phase by "flipping" a *common coin*.

Intuitively, a common coin procedure imitates the public tossing of a biased coin such that all processors see the coin landing on side v with probability at least ρ, for every v; there is a possibility that processors will not see the coin landing on the same value. Formally, an *f-resilient common coin* with *bias* ρ is a procedure (with no input) that returns a binary output. For every admissible adversary and initial configuration, all nonfaulty processors executing the procedure output v with probability at least ρ, for any value $v \in \{0, 1\}$.

A simple f-resilient common coin procedure for any $f < n$ is to have each processor output a random bit with uniform distribution; the bias of this common coin is small, 2^{-n}. Later, we present a common coin procedure with constant bias. As we shall see below, a larger bias decreases the expected number of rounds until processors decide.

The voting mechanism and the common coin procedure employ a simple information exchange procedure called get-core. This procedure guarantees the existence of a *core* set of $n - f$ processors whose values are observed by all (nonfaulty) processors; the procedure relies on the assumption that $n > 2f$ and that all processors execute it.

Algorithm 43 presents procedure get-core, which has three asynchronous phases; in every phase, a processor broadcasts to all processors and then collects information from $n - f$ processors. The algorithm relies on basic broadcast, which involves only sending a message to all processors and can be implemented in $O(1)$ time.

We first prove the properties of procedure get-core. The next lemma, whose proof is left to Exercise 14.10, shows that each nonfaulty processor eventually terminates its invocation of get-core. Recall that we assume that all $n > 2f$ processes execute get-core.

Lemma 14.4 *If processor p_i is nonfaulty, then p_i eventually returns from* get-core.

Next, we prove the existence of a large *core* group of processors whose values are seen by all nonfaulty processors.

Lemma 14.5 *There exists a set of processors, C, such that $|C| > \frac{n}{2}$ and, for every processor $p_i \in C$ and every processor p_j that returns V_j from* get-core, *V_j contains p_i's argument to* get-core.

Proof. Define a table T with n rows and n columns. For each i and j between 0 and $n - 1$, entry $T[i, j]$ contains a one if processor p_i receives a ⟨second⟩ message from p_j before sending its ⟨third⟩ message and a zero otherwise. If p_i never sends a ⟨third⟩ message, then $T[i, j]$ contains a one if and only if p_j sends its ⟨second⟩ message.

Each row contains at least $n - f$ ones, since a processor that sends a ⟨third⟩ message waits for $n - f$ ⟨second⟩ messages before doing so, and a processor that does not send a ⟨third⟩ message is the recipient of a ⟨second⟩ message from each of the $n - f$ (or more) nonfaulty processors. Thus the total number of ones in the table is at least $n(n - f)$. Since there are n columns it follows that some column ℓ contains at least $n - f$ ones. This means that the set P' of processors not receiving a ⟨second⟩ message from p_ℓ before sending their ⟨third⟩ message, contains at most f processors.

Algorithm 43 Procedure get-core: code for processor p_i, $0 \leq i \leq n-1$.

Initially *first-set*, *second-set*, *third-set* $= \emptyset$; *values*$[j] = \perp, 0 \leq j \leq n-1$

```
1:  when get-core(val) is invoked:
2:      values[i] := val
3:      bc-send(⟨first,val⟩,basic)

4:  when ⟨first,v⟩ is received from p_j:
5:      values[j] := v
6:      add j to first-set                    // track the senders of ⟨first⟩ messages
7:      if |first-set| = n − f then
            bc-send(⟨second,values⟩,basic)

8:  when ⟨second,V⟩ is received from p_j:
9:      if values[k] = ⊥ then values[k] := V[k], 0 ≤ k ≤ n − 1
                                              // merge with p_j's values
10:     add j to second-set                   // track the senders of ⟨second⟩ messages
11:     if |second-set| = n − f then
            bc-send(⟨third,values⟩,basic)

12: when ⟨third,V'⟩ is received from p_j:
13:     if values[k] = ⊥ then values[k] := V'[k], 0 ≤ k ≤ n − 1
                                              // merge with p_j's values
14:     add j to third-set                    // track the senders of ⟨third⟩ messages
15:     if |third-set| = n − f then return values
```

Let W be the set of values sent by p_ℓ in its ⟨second⟩ message. By the algorithm, $|W| \geq n - f > n/2$, that is, W contains the values of at least $n - f$ processors (the "core" processors).

Since $n - f > f$, every processor receives at least one ⟨third⟩ message from a processor not in P' before completing procedure get-core. This ⟨third⟩ message clearly includes W, the values of the core processors. This implies the lemma. □

The voting mechanism appears in Algorithm 44. It is deterministic—all the randomization is embedded in the common coin procedure.

Fix some admissible execution α of the algorithm. We say that processor p_i *prefers* v *in phase* r if *prefer*$_i$ equals v in Lines 1–5 of phase r. For phase 1 the preference is the input. For any phase $r > 1$, the preference for phase r is assigned either in Line 6 or in Line 7 of the previous phase; if the assignment is in Line 6 then p_i *deterministically prefers* v *in phase* r.

If all processors prefer v in some phase r, then they all vote for v in phase r. Therefore, all processors that reach Line 4 of phase r obtain $n - f$ votes for v and decide on v. This implies:

Algorithm 44 The consensus algorithm: code for processor p_i, $0 \leq i \leq n-1$.

Initially $r = 1$ and *prefer* = p_i's input x_i

```
1:  while true do                                              // phase r
2:      votes := get-core(⟨vote, prefer, r⟩)
3:      let v be the majority of phase r votes                 // default if no majority
4:      if all phase r votes are v then y := v                 // decide v
                                  // do not terminate, continue with the algorithm
5:      outcomes := get-core(⟨outcome, v, r⟩)
6:      if all phase r outcome values received are w then prefer := w
7:      else prefer := common-coin()
8:      r := r + 1
```

Lemma 14.6 *For any $r \geq 1$, if all processors reaching phase r prefer v in phase r, then all nonfaulty processors decide on v no later than phase r.*

This lemma already shows that the algorithm has the validity property; if all processors start with the same input v, then they all prefer v in the first phase and, therefore, decide on v in the first phase. To prove agreement, we show that if a processor decides on some value in some phase, then all other processors prefer this value in later phases. In particular, this implies that once a processor decides on v, it continues to prefer v in later phases.

For any binary value v, let $\bar{v}$ denote $1 - v$.

Lemma 14.7 *For any $r \geq 1$, if some processor decides on v in phase r, then all nonfaulty processors either decide on v in phase r or deterministically prefer v in phase $r + 1$.*

Proof. Assume that some processor p_i decides on v in phase r. This implies that p_i obtains only votes for v in phase r. By Lemma 14.5, every other processor obtains at least $n - f$ votes for v and at most f votes for $\bar{v}$, in phase r. Since $n \geq 2f + 1$, $n - f$ has majority over f, and thus all processors see a majority vote for v in phase r. Thus a processor either decides v or has outcome v in phase r. Since all ⟨outcome⟩ messages in round r have value v, every processor sets its *prefer* variable to v at the end of round r. □

The next lemma shows that if processors' preferences were picked deterministically then they agree.

Lemma 14.8 *For any $r \geq 1$, if some processor deterministically prefers v in phase $r + 1$, then no processor decides on $\bar{v}$ in phase r or deterministically prefers $\bar{v}$ in phase $r + 1$.*

Proof. Assume that some processor p_i deterministically prefers v in phase $r + 1$. By Lemma 14.7, no processor decides on $\bar{v}$ in phase r.

Assume, by way of contradiction, that some processor p_j deterministically prefers $\bar{v}$ in phase $r+1$. But then p_i sees $n-f$ processors with outcome value v, and p_j sees $n-f$ processors with outcome value $\bar{v}$. However, $2(n-f) > n$, since $n \geq 2f+1$; thus, some processor sent $\langle \text{outcome} \rangle$ messages for both v and $\bar{v}$, which is a contradiction. □

Assume that r_0 is the earliest phase in which some processor, say p_i, decides on some value, say v. By Lemma 14.8, no nonfaulty processor decides on $\bar{v}$ in phase r_0; by the minimality of r_0, no nonfaulty processor decides on $\bar{v}$ in any earlier phase. Lemma 14.7 implies that all nonfaulty processors prefer v in round r_0+1, and finally, Lemma 14.6 implies that all nonfaulty processors decide on v no later than round r_0+1. Thus the algorithm satisfies the *agreement* condition:

Lemma 14.9 *If some processor decides on v then all nonfaulty processors eventually decide on v.*

In fact, all nonfaulty processors decide on v no later than one phase later. The algorithm, as presented so far, requires processors to continue even after they decide; however, the last observation implies that a processor need only participate in the algorithm for one more phase after deciding. (See Exercise 14.12.)

To bound the expected time complexity of the algorithm, we prove that the probability of deciding in a certain phase is at least ρ, the bias of the common coin.

Lemma 14.10 *For any $r \geq 1$, the probability that all nonfaulty processors decide by phase r is at least ρ.*

Proof. We consider two cases.

Case 1: All nonfaulty processors set their preference for phase r to the return value of the common coin. The probability that all nonfaulty processors obtain the same value from the common coin is at least 2ρ—with probability ρ they all obtain 0 and with probability ρ they all obtain 1, and these two events are disjoint. In this case, all processors prefer v in phase r, so they decide at the end of phase r, by Lemma 14.6.

Case 2: Some processor deterministically prefers v in phase r. Then no processor deterministically prefers $\bar{v}$ in phase r, by Lemma 14.8. Thus each processor that reaches phase r either decides on v in phase $r-1$, or deterministically prefers v in phase r, or uses the common coin to set its preference for phase r. In the latter case, with probability at least ρ, the return value of the common coin of phase r is v for all processors. Thus, with probability at least ρ, all processors that reach phase r prefer v in phase r, and, by Lemma 14.6, all nonfaulty processors decide on v in phase r. □

Theorem 14.11 *If Algorithm 44 uses a common coin procedure with bias ρ, whose expected time complexity is T, then the expected time complexity of Algorithm 44 is $O(\rho^{-1}T)$.*

Algorithm 45 Procedure common-coin: code for processor p_i, $0 \leq i \leq n-1$.

```
1: when common-coin() is invoked:
2:     c := { 0  with probability 1/n
            { 1  with probability 1 - 1/n
3:     coins := get-core(⟨flip,c⟩)
4:     if there exists j such that coins[j] = 0 then return 0
5:     else return 1
```

Proof. First note that when $n \geq 2f+1$, procedure get-core takes $O(1)$ time.

By Lemma 14.10, the probability of terminating after one phase is at least ρ. Therefore, the probability of terminating after i phases is at least $(1-\rho)^{i-1}\rho$. Therefore, the number of phases until termination is a geometric random variable whose expected value is ρ^{-1}.

Clearly, the time complexity of the common coin procedure dominates the time complexity of a phase. Thus each phase requires $O(T)$ expected time and therefore, the expected time complexity of the algorithm is $O(\rho^{-1}T)$. □

14.3.2 A Common Coin with Constant Bias

As mentioned before, there is a simple f-resilient common coin algorithm with bias 2^{-n} and constant running time, for every $f < n$; each processor outputs a random bit with uniform distribution. Clearly, with probability 2^{-n} the value of the common coin for all processors is v, for any $v = 0, 1$. The expected time complexity of Algorithm 44 when using this coin is $O(2^n)$, by Theorem 14.11.

To get constant expected time complexity, we present an implementation for the procedure common-coin with constant bias. The different invocations of common-coin in the different phases of Algorithm 44 need to be distinguished from each other; this can be achieved by adding the phase number to the messages.

We present a common coin algorithm with bias $\frac{1}{4}$; the algorithm tolerates $\lceil \frac{n}{2} \rceil - 1$ crash failures. The key idea of the common coin algorithm is to base the value of the common coin on the local (independent) coin flips of a set of processors. Procedure get-core (from the previous section) is used to ensure that there is a large *core* set of processors whose local coin flips are used by all nonfaulty processors; this increases the probability that nonfaulty processors obtain the same return value for the common coin.

Processors begin the algorithm by randomly choosing 0 or 1 (with carefully selected asymmetric probabilities) and exchange their flips by using get-core. Processor p_i returns 0 for the common coin procedure if it received at least one 0 flip; otherwise, it returns 1. The pseudocode appears in Algorithm 45.

The correctness of the algorithm depends on all processors executing it; we can modify Algorithm 44 so that the procedure is executed in every phase, regardless of whether the processor needs the coin in that phase.

The common coin procedure assumes that the channels are secure and cannot be read by the adversary—only the recipient of a message can learn its contents. This assumption corresponds to a weak adversary, that is, one that takes as input the pattern of messages, but not their contents; because the message pattern is the same regardless of the random choices, the adversary can obtain no information at runtime about the random choices.

Lemma 14.12 *Algorithm 45 implements a* $(\lceil \frac{n}{2} \rceil - 1)$*-resilient coin with bias* $\frac{1}{4}$.

Proof. Fix any admissible adversary and initial configuration. All probabilities are calculated with respect to them.

First we show that the probability that all nonfaulty processors see the coin as 1 is at least $\frac{1}{4}$. This probability is at least the probability that every processor that executes Line 1 obtains 1. (Of course, there are other cases in which all nonfaulty processors terminate with 1, but it is sufficient to consider only this case.) The probability that an arbitrary processor obtains 1 in Line 1 is $1 - \frac{1}{n}$, and thus the probability that every processor that executes Line 1 obtains 1 is at least $(1 - \frac{1}{n})^n$, because the different processors' random choices are independent. For $n \geq 2$, the function $(1 - \frac{1}{n})^n$ is increasing up to its limit of e^{-1}. When $n = 2$, this function is $\frac{1}{4}$ and we are done.

To show the probability that all nonfaulty processors see the coin as 0 is at least $\frac{1}{4}$, consider the set C of core processors, whose existence is guaranteed by Lemma 14.5. Since every processor that returns from Algorithm 45 observes the random choices of all processors in C, all nonfaulty processors see the coin as 0 if some processor in C obtains 0 in Line 1.

The probability that some processor in C gets a 0 in Line 1 is $1 - (1 - \frac{1}{n})^{|C|}$, which is more than $1 - (1 - \frac{1}{n})^{n/2}$. To verify that this last expression is at least $\frac{1}{4}$, it suffices to verify that $(1 - \frac{1}{n})^n \leq (\frac{3}{4})^2$. Since $(1 - \frac{1}{n})^n$ is increasing up to its limit, we must verify that the limiting value e^{-1} is at most $(\frac{3}{4})^2$. This holds since $e^{-1} \approx 0.46$ and $(\frac{3}{4})^2 \approx 0.56$. □

Note that the time complexity of the common coin algorithm is $O(1)$. Because Algorithm 45 provides a $(\lceil \frac{n}{2} \rceil - 1)$-resilient common coin with bias $\frac{1}{4}$, Theorem 14.11 implies:

Theorem 14.13 *If* $n \geq 2f + 1$*, then there is a randomized consensus algorithm with* $O(1)$ *expected time complexity that can tolerate* f *crash failures in an asynchronous system.*

14.3.3 Tolerating Byzantine Failures

In this section, we describe modifications to the previous algorithm to tolerate Byzantine failures.

The first modification is to the broadcast primitive employed. Instead of using the basic broadcast from Chapter 8, we will use failure model simulations from Chapter 12. The first step is to mask the potential two-faced behavior of the Byzantine

processors with the asynchronous identical Byzantine failures simulation (Algorithm 39). On top of that, we will need a validation procedure to eliminate inappropriate messages and simulate asynchronous omission failures; the synchronous validate procedure (from Algorithm 37) can be modified to work in the asynchronous case.

The second modification is to use a common coin procedure that tolerates f Byzantine failures with bias ρ. The simple common coin algorithm, in which each processor flips a local coin, tolerates any number of Byzantine failures with an exponential bias. However, the exponentially small bias leads to exponentially large expected time complexity for the resulting algorithm. The running time is dramatically improved if a common coin with constant bias is used. Implementing a common coin with constant bias that tolerates Byzantine failures requires sophisticated techniques; for more details, see the chapter notes.

We leave it to the reader as an exercise to prove that the modified Algorithm 44 solves randomized consensus in the presence of f Byzantine failures. Moreover, if the expected time complexity of the common coin algorithm is T, then the expected time complexity of the randomized consensus algorithm is $\rho^{-1}T$. (See Exercise 14.13.)

The proof of the lower bound of Chapter 5 (Theorem 5.8) can be extended to prove that randomized consensus is possible only if $n \geq 3f + 1$. Thus the algorithm sketched in this section is optimal in the number of processors.

14.3.4 Shared Memory Systems

We now discuss how randomized consensus can be achieved in shared memory systems.

One possible way is to run the algorithms for message-passing systems by using some simple simulation, for example, the one described in Section 5.3.3, in a shared memory system. This approach suffers from two major shortcomings: First, the resulting algorithms are not wait-free, that is, they tolerate only $n/2$ failures at best; in message-passing systems we cannot hope to tolerate more failures, but this is not the case in shared memory systems. Second, because we only care about tolerating crash failures in shared memory systems, the algorithm can cope with stronger adversaries, such as those that have access to the local states and the contents of shared memory.

Interestingly, randomized consensus algorithms for shared memory systems have a general structure very similar to that of the message-passing algorithms.

Again, we have a deterministic algorithm that operates in (asynchronous) phases: A processor has a preference for decision at each phase; the processor announces its preference and it checks to see whether there is support for some decision value; if there is strong support for some value, the processor decides on it; if there is weak support for some value, the processor takes this value as preference to the next phase; if no single value has support, then the processor flips a coin (typically using some common coin procedure) to get a preference for the next phase.

Unlike the message-passing algorithms, because the algorithm has to be wait-free, we cannot count on the existence of a certain number of nonfaulty processors in each phase. Therefore, support level is not determined by the number of votes for a value,

but rather according to the largest phase in which some processor prefers this value. We do not discuss further details of this scheme here, and refer the reader to the chapter notes.

A simple common coin with exponential bias can be implemented by independent coin flips, as described for message-passing systems. More sophisticated techniques are required to obtain constant bias; they are also discussed in the chapter notes.

Exercises

14.1 Prove that every processor terminates Algorithm 41 after sending n messages.

14.2 Prove that at most one processor terminates Algorithm 41 as a leader.

14.3 Modify Algorithm 41 so that each message contains a single pseudo-identifier: The termination condition for the algorithm (Lines 6 and 7) needs to be modified.

14.4 Prove that both the one-shot and the iterated synchronous leader election algorithms (Algorithm 41 and its extension) have the same performance in the asynchronous case as they do in the synchronous.

14.5 Show that for synchronous anonymous ring algorithms, there is only a single adversary.

14.6 Complete the details of the proof of Theorem 14.3.

Try to prove a stronger result showing that there is no randomized leader election algorithm that knows n within a factor larger than 2. Formally, prove that there is no randomized leader election algorithm that works both for rings of size n and for rings of size $2n$.

14.7 Calculate the average message complexity of the randomized consensus algorithm (Algorithm 44), given a common coin with bias $\rho > 0$.

14.8 Calculate the average message complexity of the common coin procedure of Algorithm 45.

14.9 Prove that $n \geq 2f + 1$ is a necessary condition for solving randomized consensus in an asynchronous systems, in the presence of f crash failures.

Hint: Modify the proof for Theorem 10.22.

14.10 Prove Lemma 14.4.

14.11 Suppose we have a more restricted model, in which processors can choose random values only with uniform probability distributions (on a bounded range). Show how to pick 0 with probability $1/n$ and 1 with probability $1 - 1/n$, as needed for Algorithm 45.

14.12 Modify the pseudocode of Algorithm 44 so that processors terminate one phase after deciding. Prove that the modified algorithm solves randomized consensus.

14.13 Extend Algorithm 44 to tolerate $f > n/3$ Byzantine failures, using asynchronous identical Byzantine failures and a modified version of validate from Chapter 12. Assume the existence of a common coin procedure that tolerates f Byzantine failures, with bias ρ.

14.14 Prove that randomized consensus with probability 1 is possible only if $n \geq 3f + 1$, when there are Byzantine failures.

Hint: Extend the proof of Theorem 5.8, noting that every finite execution has at least one extension that terminates.

Chapter Notes

The book on randomized algorithms by Motwani and Raghavan [194] includes a chapter on distributed algorithms. The survey by Gupta, Smolka, and Bhaskar [127] covers randomized algorithms for dining philosophers, leader election, message routing, and consensus.

Itai and Rodeh [141] were the first to study randomized algorithms for leader election; the proof showing nonexistence of a uniform leader election algorithm in an anonymous ring is taken from their paper. In fact, they show that a necessary and sufficient condition for the existence of a randomized leader election algorithm for anonymous rings is knowing the number of processors on the ring within a factor of two (Exercise 14.6). Higham's thesis [136] describes improved algorithms and lower bounds for leader election and computation of other functions on rings.

Randomized leader election algorithms for systems with general topology were presented by Schieber and Snir [234] and by Afek and Matias [6].

An algorithm for mutual exclusion with small shared variables was suggested by Rabin [223]; this algorithm has average waiting time depending on m, the number of processors actually contending for the critical section, and requires a $O(\log \log n)$-bit shared variable. The algorithm we presented has average waiting time depending on n, the number of processors, and uses a constant-size shared variable; this algorithm appears in [223] and is credited to Ben-Or. Saias [232] has shown that the adversary can increase the expected waiting time of a processor, showing that the original algorithm of Rabin did not have the claimed properties. Kushilevitz and Rabin [151] have shown how to overcome this problem; Kushilevitz, Mansour, Rabin, and Zuckerman [150] proved lower bounds on the size of shared variables required for providing low expected bounded waiting.

The first randomized algorithm for consensus in an asynchronous message-passing system was suggested by Ben-Or [48]; this algorithm tolerated Byzantine failures and required $n > 5f$; its expected running time was exponential. A long sequence of randomized algorithms for consensus followed, solving the problem under different

assumptions about the adversary, the failure type, the ratio of faulty processors, and more. A survey by Chor and Dwork [80] describes this research prior to 1988. Aspnes [21] surveys later advances in this area.

In 1988, Feldman and Micali [104] presented the first algorithm that tolerates a linear fraction of Byzantine failures with constant expected time; they presented a synchronous algorithm requiring $n > 3f$ and an asynchronous algorithm requiring $n > 4f$. The best asynchronous algorithm known to date, due to Canetti and Rabin [65], tolerates Byzantine failures with constant expected time and only requires $n > 3f$.

Our presentation follows Canetti and Rabin's algorithm, simplified to handle only crash failures; the procedure for obtaining a core set of values was suggested by Eli Gafni. The deterministic voting scheme of Canetti and Rabin's algorithm is inspired by Bracha [60]. Bracha's algorithm uses a simulation of identical Byzantine failures together with a validation (as in Chapter 12); Bracha's paper includes the answer to Exercise 14.13. Canetti and Rabin, on the other hand, embed explicit validation in their algorithm.

There is no upper bound on the running time of the consensus algorithm presented here that holds for *all* executions. Goldreich and Petrank [123] showed how to obtain a synchronous consensus algorithm (for Byzantine failures) that has constant expected time complexity and is guaranteed to terminate within $f + 2$ rounds.

There are also many randomized consensus algorithms for shared memory systems. Although there are many similarities in the overall structure, the common coin is typically implemented using different techniques than those in message-passing systems. These algorithms are wait-free but do not have to tolerate Byzantine failures. Therefore, they can withstand a strong adversary, which can observe the local states and the shared memory. Against such an adversary, cryptographic techniques are not helpful.

The first randomized consensus algorithm for shared memory systems was given by Chor, Israeli, and Li [81]; this algorithm assumed an adversary that cannot read the local states of processors. Later, Abrahamson [1] presented the first algorithm that can withstand a strong adversary. This was followed by a number of algorithms; to date, the best algorithm that tolerates a strong adversary is due to Aspnes and Waarts [23]. Aspnes [20] proved a lower bound on the number of coin flips needed for solving consensus.

The algorithms presented here demonstrate a general methodology for using randomization safely. A deterministic mechanism guarantees the safety properties regardless of the adversary (mostly as a separate module), for example, having only a single leader or having agreement and validity; randomization enters the algorithm only to guarantee liveness properties, for example, termination or bounded waiting.

Distributed Computing: Fundamentals, Simulations and Advanced Topics, Second Edition
Hagit Attiya and Jennifer Welch
 ISBN: 0-471-45324-2

15

Wait-Free Simulations of Arbitrary Objects

The results of Chapter 5 indicate that read/write registers memory do not suffice for wait-free coordination among processors; for example, consensus cannot be solved by using only reads and writes. Indeed, most modern and proposed multiprocessors provide some set of "stronger" hardware primitives for coordination. But are these primitives sufficient? Can they be used to provide a wait-free (fault tolerant) simulation of any desired high-level object in software?

In Chapter 10, we saw how stronger shared objects can be simulated with simpler shared objects, in a manner that is wait-free. A closer look at the objects studied in that chapter reveals that the operations they support involve either reading or writing portions of the shared memory. They differ in their size (whether the values read are binary or can take arbitrary values), in their access pattern (whether a single processor or several processors can read or write a location), and in their read granularity (whether a single location or several locations are read in a single step). However, none of them allows a processor to read and write in an atomic step, as happens in a *read-modify-write* or *test&set* operation.

In this chapter, we show that this is inherent, that is, that some shared objects cannot be simulated with other shared objects, in a wait-free manner. We investigate the following question: Given two types of (linearizable) shared objects, X and Y, is there a wait-free simulation of object type Y using only objects of type X and read/write objects? It turns out that the answer depends on whether X and Y can be used to solve the consensus problem.

Slightly weaker than wait-free simulations are nonblocking simulations. The idea of a *nonblocking* simulation is that, starting at any point in an admissible execution in which some high-level operations are pending, there is a finite sequence of steps by a

single processor that will complete one of the pending operations. It is not necessarily the processor taking the steps whose operation is completed—it is possible that this processor will make progress on behalf of another processor while being unable to finish its own operation. However, this definition is not quite correct. For an operation to complete, technically speaking, the invoking processor has to do at least one step, the response. That is, another processor cannot totally complete an operation for another. To avoid this problem, we specify that an operation is *pending* if its response is not enabled.

The nonblocking condition is a global progress property that applies to the whole system, not to individual processors. The distinction between wait-free and nonblocking algorithms is similar to the distinction between no-lockout and no-deadlock algorithms for mutual exclusion.

We start this chapter by considering a specific object, the FIFO queue, and then extend the results to arbitrary objects. For convenience, we often identify an object type with the operations that can be applied to its instances.

15.1 EXAMPLE: A FIFO QUEUE

The approach we take is to compare the objects by their ability to support a wait-free consensus algorithm among a certain number of processors. To gain intuition, let us consider, first, a specific object, namely a FIFO queue, and see which objects cannot simulate it and which objects cannot be simulated with it. This example illustrates many of the general ideas we discuss later; furthermore, the FIFO queue is an important data structure widely used in operating systems software.

By the methodology of Chapter 7, the specification of a FIFO queue Q is as follows. The operations are [enq(Q, x),ack(Q)] and [deq(Q),return(Q, x)], where x can be any value that can be stored in the queue (x can be $\perp$ for the return). The set of allowable operation sequences consists of all sequences of operations that are consistent with the usual semantics of a sequential FIFO queue, namely, values are dequeued in the order in which they were enqueued. The events on FIFO queues are enq, deq, ack, and return.

Recall, from Chapter 5, that there is no wait-free consensus algorithm for any number of processors greater than one if we only have read/write objects. We now show that there is a wait-free consensus algorithm for two processors p_0 and p_1, using a FIFO queue and read/write objects.

The algorithm uses two read/write objects and one shared FIFO queue Q that initially holds 0. Processor p_i writes its input to a shared read/write object *Prefer*$[i]$ and then dequeues the queue. If the dequeued value is 0, then p_i accessed the queue before the other processor did and is the "winner," and it decides on its own input. Otherwise p_i "lost" to the other processor and it reads *Prefer*$[1 - i]$ and decides on that value (see pseudocode in Algorithm 46). The correctness of the algorithm is left as an exercise.

Lemma 15.1 *Algorithm 46 solves consensus for two processors.*

Algorithm 46 Consensus algorithm for two processors, using a FIFO queue: code for processor p_i, $i = 0, 1$.

Initially $Q = \langle 0 \rangle$ and $Prefer[i] = \perp, i = 0, 1$

```
1: Prefer[i] := x                          // write your input
2: val := deq(Q)
3: if val = 0 then y := x      // dequeued the first element, decide on your input
4: else y := Prefer[1 − i]                 // decide on other's input
```

If there was a wait-free simulation of FIFO queues with read/write objects, then there would be a wait-free consensus algorithm, for two processors, using only read/write objects. Because there is no such algorithm, an immediate implication is:

Theorem 15.2 *There is no wait-free simulation of a FIFO queue with read/write objects for any number of processors.*

This is a special case of a general theorem we present below (Theorem 15.5).

There are objects that admit wait-free consensus algorithms for any number of processors, for example, *compare&swap*. It is simplest to present the sequential specification of a compare&swap object by the following procedure:

```
compare&swap(X : memory address; old , new : value) returns value
    previous : = X
    if previous = old then X := new
    return previous
```

Algorithm 47 solves consensus for any number of processors, using a single compare&swap object. The object, called *First*, is initially $\perp$. Processor p_i performs compare&swap on *First*, trying to store its own input, x_i in it, if *First* is still initialized. If the operation is successful, that is, if it returns $\perp$, the processor decides on its own input; otherwise, the compare&swap operation returns the input value, v, of some processor, and processor p_i decides on v.

On the other hand, FIFO queue objects (and read/write objects) cannot support a wait-free consensus algorithm for three processors or more.

Theorem 15.3 *There is no n-processor wait-free consensus algorithm using only FIFO queues and read/write objects, if $n \geq 3$.*

Proof. We must show that there is no three-processor consensus algorithm using queues and read/write objects in an asynchronous system in which up to two processors can fail. The proof of this theorem has the same structure as the proof of Theorem 5.18.

Assume, in contradiction, that there is such an algorithm for three processors, p_0, p_1, and p_2. Because shared objects are linearizable, we shall simplify our notation by combining invocations and matching responses on a shared object into a single step by the processor.

Algorithm 47 Consensus algorithm for any number of processors, using compare&swap: code for processor p_i, $0 \leq i \leq n-1$.

Initially $First = \perp$

```
1:  v := compare&swap(First,⊥,x)
2:  if v = ⊥ then                    // this is the first compare&swap
3:      y := x                       // decide on your own input
4:  else y := v                      // decide on someone else's input
```

By Lemma 5.16, the algorithm has an initial bivalent configuration, B.

As in the proof of Theorem 5.18, we let p_0, p_1, and p_2 take steps until we reach a critical configuration, C, such that $p_0(C)$, $p_1(C)$, and $p_2(C)$ are all univalent. Note that they cannot all have the same value, or C would not be bivalent. Without loss of generality, assume that $p_0(C)$ is 0-valent and $p_1(C)$ is 1-valent (see Fig. 15.1).

What are p_0 and p_1 doing in their steps from C? If p_0 and p_1 access different variables or access the same read/write object, then we can employ arguments similar to those used in the proof of Theorem 5.18 to derive a contradiction. So, the interesting case is when p_0 and p_1 access the same FIFO queue object, Q. We consider three cases.

Case 1: p_0 and p_1 both dequeue from Q. In this case, $p_0(C) \overset{p_2}{\sim} p_1(C)$. Since $p_0(C)$ is 0-valent, Lemma 5.15 implies that $p_1(C)$ is also 0-valent, which is a contradiction.

Case 2: p_0 enqueues on Q, and p_1 dequeues from Q (or vice versa). If Q is not empty in C, then in $p_1(p_0(C))$ and $p_0(p_1(C))$ all objects and all processors are in the same state, which is a contradiction since $p_1(p_0(C))$ is 0-valent whereas $p_0(p_1(C))$ is 1-valent.

If Q is empty, then p_1's dequeue in C returns an empty indication, whereas p_1's dequeue in $p_0(C)$ does not return an empty indication; so the previous argument does not apply. However, $p_0(C) \overset{p_2}{\sim} p_0(p_1(C))$. Since $p_1(C)$ is 1-valent, $p_0(p_1(C))$ is also 1-valent, and by Lemma 5.15, $p_0(C)$ is 1-valent, which is a contradiction.

Case 3: p_0 and p_1 both enqueue on Q. Suppose p_0 enqueues a and p_1 enqueues b (see Fig. 15.2). Let $k-1$ be the number of values in Q in C. Thus in $p_0(C)$ the kth value is a, and in $p_1(C)$ the kth value is b.

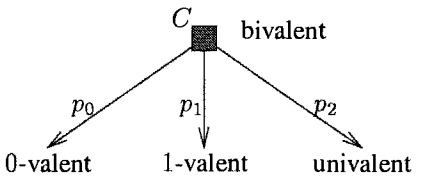

Fig. 15.1 The critical configuration, C.

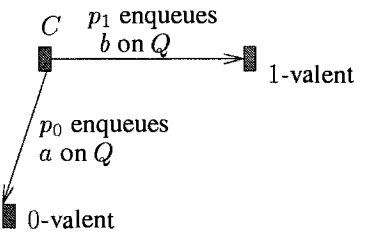

Fig. 15.2 Case 3 in the proof of Theorem 15.3.

We will show that we can run p_0 alone until it dequeues the a and then run p_1 alone until it dequeues the b. This means that even if a and b are enqueued in the opposite order, we can run p_0 alone until it dequeues the b and then run p_1 alone until it dequeues the a. Now p_2 cannot tell the difference between the two possible orders in which a and b are enqueued, which is a contradiction. The details follow.

Starting from $p_1(p_0(C))$, there is a finite p_0-only schedule, σ, that ends with p_0 deciding 0, since $p_0(C)$ is 0-valent and the algorithm is wait-free. We first argue that in σ, p_0 must dequeue the kth element. Suppose otherwise, that in σ, p_0 does fewer than k dequeues on Q, so that it never dequeues the a. Then, when we apply the schedule $p_0\sigma$ to $p_1(C)$, p_0 also decides 0, which contradicts the fact that $p_1(C)$ is 1-valent.

Thus p_0 must perform at least k dequeues on Q in σ. Let σ' be the longest prefix of σ that does not include p_0's kth dequeue on Q.

Starting from $\sigma'(p_1(p_0(C)))$, there is a finite p_1-only schedule τ that ends with p_1 deciding 0. We now argue that p_1 must dequeue from Q at some point in τ. Assume otherwise that p_1 never dequeues from Q in τ; namely, p_1 never dequeues b from the head of Q. Then when we apply the schedule τ to $\sigma'(p_0(p_1(C)))$, p_1 also decides 0, which contradicts the fact that $p_1(C)$ is 1-valent.

Thus it must be that p_1 dequeues from Q in τ. Let τ' be the longest prefix of τ that does not include p_1's dequeue on Q.

Consider two extensions from C (see Fig. 15.3):

First, consider the execution in which p_0 enqueues a on Q; p_1 enqueues b on Q; p_0 only takes steps in σ'; p_0 dequeues the a; p_1 only takes steps in τ'; p_1 dequeues the b. Let D_0 be the resulting configuration.

Second, consider the execution in which p_1 enqueues b on Q; p_0 enqueues a on Q; p_0 only takes steps in σ'; p_0 dequeues the b; p_1 only takes steps in τ'; p_1 dequeues the a. Let D_1 be the resulting configuration.

However, $D_0 \overset{p_2}{\sim} D_1$, and therefore, by Lemma 5.15, they must have the same valence. This is a contradiction since D_0 is reachable from $p_1(p_0(C))$, which is 0-valent, whereas D_1 is reachable from $p_0(p_1(C))$, which is 1-valent. □

This implies:

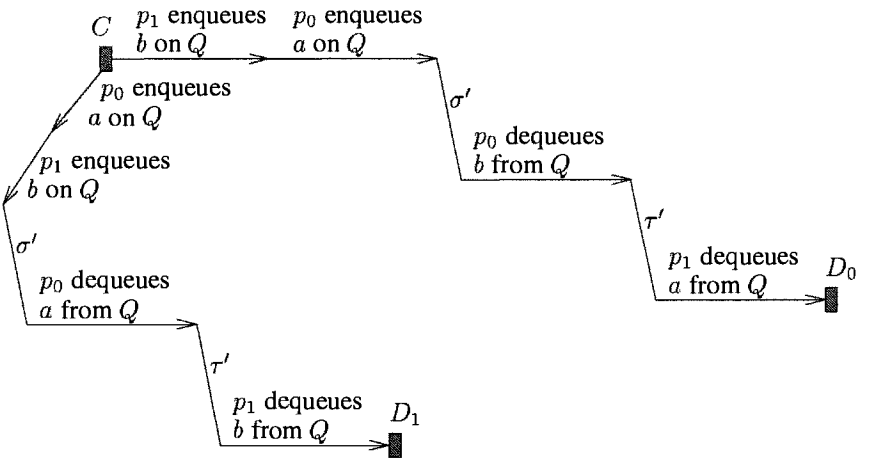

Fig. 15.3 Final stage in Case 3.

Theorem 15.4 *There is no wait-free simulation of a compare&swap object with FIFO queue objects and read/write objects, for three processors or more.*

15.2 THE WAIT-FREE HIERARCHY

The methods used for FIFO queues can be generalized into a criterion for the existence of wait-free simulations. The criterion is based on the ability of the objects to support a consensus algorithm for a certain number of processors.

Object type X *solves wait-free n-processor consensus* if there exists an asynchronous consensus algorithm for n processors, up to $n-1$ of which might fail (by crashing), using only shared objects of type X and read/write objects.

The *consensus number* of object type X is n, denoted $CN(X) = n$, if n is the largest value for which X solves wait-free n-processor consensus. The consensus number is infinity if X solves wait-free n-processor consensus for every n. Note that the consensus number of an object type X is at least 1, because any object trivially solves wait-free one-processor consensus.

For example, the consensus number of read/write objects is 1. There is a trivial algorithm for one-processor consensus using read/write objects, and back in Chapter 5 we showed that there is no wait-free algorithm for two-processor consensus, using only read/write objects.

As shown in Section 15.1, the consensus number of FIFO queues is 2. Other examples of objects with consensus number 2 are test&set, swap, fetch&add, and stacks.

As implied by Algorithm 15.2, the consensus number of compare&swap is infinity. Other objects with infinite consensus number are memory-to-memory move,

memory-to-memory swap, augmented queues, and fetch&cons (see Exercise 15.7). In fact, there is a hierarchy of object types, according to their consensus numbers; in particular, there are objects with consensus number m, for any m (see Exercise 15.8).

The consensus number of objects is interesting because of Theorem 15.5, which is the key result of this section.

Theorem 15.5 *If $CN(X) = m$ and $CN(Y) = n > m$, then there is no wait-free simulation of Y with X and read/write registers in a system with more than m processors.*

Proof. Suppose, in contradiction, that there is a wait-free simulation of Y with X and read/write registers in a system with $k > m$ processors. Denote $l = \min\{k, n\}$, and note that $l > m$. We argue that there exists a wait-free l-processor consensus algorithm using objects of type X and read/write objects.

Note that even if $l < k$, then there is also a wait-free simulation of Y with X in a system with l processors. Such a simulation can be achieved by employing $k - l$ "fictitious" processors, that never access the object Y.

Since $l \leq n$, there exists a wait-free l-processor consensus algorithm, A, which uses only objects of type Y and read/write objects. We can obtain another algorithm A' by replacing each type Y object with a wait-free simulation of it using objects of type X and read/write objects. Such a wait-free simulation of type Y objects exists, by assumption.

Then A' is a wait-free l-processor consensus algorithm using objects of type X and read/write objects. Therefore, X has consensus number at least $l > m$, which is a contradiction. □

Corollary 15.6 *There is no wait-free simulation of any object with consensus number greater than 1 using read/write objects.*

Corollary 15.7 *There is no wait-free simulation of any object with consensus number greater than 2 using FIFO queues and read/write objects.*

15.3 UNIVERSALITY

In Section 15.2, we have used the consensus number of objects to prove that certain objects cannot be wait-free simulated by other objects, that is, to show impossibility results. It turns out the consensus number can also be used to derive positive results, that is, wait-free simulations of objects, as shown in this section.

An object is *universal* if it, together with read/write registers, wait-free simulates any other object. In this section we show that any object X whose consensus number is n is universal in a system of at most n processors. Somewhat counter-intuitively, this does not imply, say, that X is universal in any system with $m > n$ processors; this and other anomalies of the notion of universality are discussed in the chapter notes.

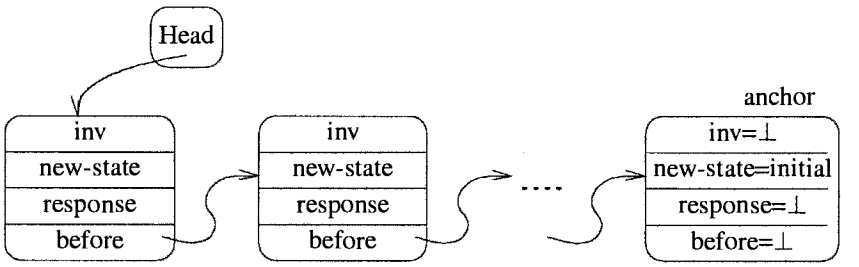

Fig. 15.4 Data structures for nonblocking simulation using compare&swap.

We prove this result by presenting a *universal* algorithm for wait-free simulating any object using only objects of type X and read/write registers. First, we present a universal algorithm for any object in a system of n processors using only n-processor *consensus objects* and read/write registers. Informally speaking, n-processor consensus objects are data structures that allow n processors to solve consensus; a more precise specification appears below. We then use X to simulate n-processor consensus objects.

15.3.1 A Nonblocking Simulation Using Compare&Swap

We start with a simple universal algorithm that is nonblocking but not wait-free. Furthermore, this algorithm uses a specific universal object, that is, compare&swap. This will introduce the basic ideas of the universal algorithm; later, we show how to use a generic consensus object and how to make the algorithm wait-free.

The idea is to represent an object as a shared linked list, which contains the ordered sequence of operations applied to the object. To apply an operation to the object, a processor has to thread it at the head of the linked list. A compare&swap object is used to manage the head of the list. Specifically, an operation is represented by a shared record of type *opr* with the following components:

inv:	The operation invocation, including its parameters
new-state:	The new state of the object, after applying the operation
response:	The response of the operation, including its return value
before:	A pointer to the record of the previous operation on the object

In addition, a compare&swap variable, called *Head*, points to the *opr* record of the last operation applied to the object. The initial value of the object is represented by a special anchor record, of type *opr*, with *new-state* set to the initial state of the object. Initially, *Head* points to the anchor record.

The *Head* pointer, the *opr* records, and their *before* pointers comprise the object's representation (see Fig. 15.4).

To perform an operation, a processor allocates an *opr* record, initializing it to the appropriate values, using the state information in the record at the *Head* of the list.

Algorithm 48 A nonblocking universal algorithm using compare&swap: code for processor p_i, $0 \leq i \leq n-1$.

Initially *Head* points to the anchor record

```
1:  when inv occurs:                    // operation invocation, including parameters
2:      allocate a new opr record pointed to by point with point.inv := inv
3:      repeat
4:          h := Head
5:          point.new-state, point.response := apply(inv,h.new-state)
6:          point.before := h
7:      until compare&swap(Head, h, point) = h
8:      enable the output indicated by point.response      // operation response
```

Then it tries to thread this record onto the linked list by applying compare&swap to *Head*.

In more detail, the compare&swap compares the current *Head* with the value of *Head* obtained by the processor when it updated the new *opr* record. If the compare finds that they are the same, implying that no new operation record has been threaded in the meantime, then the swap causes *Head* to point to the processor's *opr* record. Otherwise, the processor again reads *Head*, updates its new record, and tries the compare&swap.

The pseudocode appears in Algorithm 48. It uses the function apply, which calculates the result of applying an operation to the current state of the object. The notation $X.v$, where X is a pointer to a record, refers to the v field of the record pointed to by X.

Proving that Algorithm 48 simulates the object is straightforward. The desired linearization is derived from the ordering of operations in the linked list.

The algorithm is nonblocking—if a processor does not succeed in threading its operation on the linked list, it must be that some other processor's compare&swap operation succeeded, that is, another processor has threaded its operation on the list. Note that the algorithm is not wait-free because the same processor might succeed in applying its operation again and again, locking all other processors out of access to the shared object.

15.3.2 A Nonblocking Algorithm Using Consensus Objects

Algorithm 48 has three shortcomings: first, it uses compare&swap objects, rather than arbitrary consensus objects; second, it is nonblocking, rather than wait-free; and third, it uses an unbounded amount of memory. We now address each of these problems, incrementally presenting the algorithm.

First, we show how to replace the compare&swap operations with an arbitrary consensus object. A *consensus object Obj* provides a single operation [decide(Obj, in), return(Obj, out)], where *in* and *out* are taken from some domain of values. The set

of operation sequences consists of all sequences of operations in which all *out* values are equal to some *in* value. Consensus objects provide a data structure version of the consensus problem.

A first attempt might be to replace the *Head* pointer (which is a compare&swap object) with a consensus object. The consensus object will be used to decide which processor will thread its new operation on the list, that is, which operation will be applied next to the shared object being simulated. Note, however, that a consensus object can be used only once; after the first processor wins the consensus and threads its operation, the consensus object will always return the same value. Therefore, the consensus object cannot be used to thread additional records on the list.

The solution is to perform the competition (to decide which processor gets to thread its next operation) on a consensus object associated with the record at the head of the list. That is, we replace the *before* component of the record with a component called *after*, which is a consensus object pointing to the next operation applied to the object. To perform an operation, as in the previous algorithm, a processor creates an *opr* record and tries to thread it to the list. The attempt to thread is done by accessing the consensus object at the head of the list with the pointer to its *opr* record as preference. If it wins the consensus, then its operation has been threaded as the next operation on the simulated object, after the current head of the list. Note that the linked list is directed from the first to the latest entry now, instead of from the latest to the first.

There is one problem with the above idea, namely, how to locate the record at the head of the list. Note that we cannot hold a pointer in a simple read/write object (otherwise, we could wait-free simulate a queue), and a consensus object cannot change with time. This was not a problem in the previous algorithm, because we kept a pointer to the head of the list in a compare&swap object.

The way around this problem is to have each processor maintain a pointer to the last record it has seen at the head of the list. These pointers are kept in a shared array called *Head*. This information might be stale, that is, a processor might thread an operation to the list, save a pointer to the threaded operation in its entry in the *Head* array, and then be suspended for a long time. A processor following this pointer later might end up in the middle of the list because other processors may thread many operations in the meanwhile. So how can a processor know which of these records is the latest? It is not possible to check whether the consensus object *after* is already set, because the consensus object cannot be read without altering its value. Instead, we add sequence numbers to the records on the list. Sequence numbers are assigned so that later operations get higher sequence numbers. The record with the highest sequence number, to which an entry in the *Head* array is pointing, is the latest on the list.

The above ideas lead to the following algorithm, which is a nonblocking simulation of an arbitrary object (for n processors) using n-processor consensus objects. The algorithm augments the record type *opr* so it contains the following components (see Fig. 15.5):

seq: Sequence number (read/write)

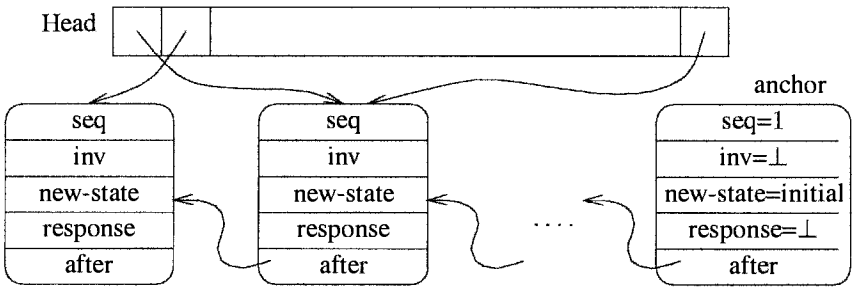

Fig. 15.5 Data structures for nonblocking simulation using consensus objects.

inv:	Operation type and parameters (read/write)
new-state:	The new state of the object (read/write)
response:	The value to be returned by the operation (read/write)
after:	Pointer to next record (consensus object)

In addition to the linked list of the operations applied to the object, the algorithm uses a shared array $Head[0..n-1]$; the ith entry, $Head[i]$, is a pointer to the last cell in the list that p_i has observed. Initially, all entries in *Head* point to the anchor record, which has sequence number 1.

The pseudocode appears in Algorithm 49.

Showing that Algorithm 49 simulates the object is rather straightforward. Given an admissible execution α of the algorithm, the desired linearization is derived from the sequence numbers of the records representing operations in the linked list. Clearly, this linearization preserves the semantics of the simulated object, and the relative order of non-overlapping operations.

We now study the progress properties of the algorithm.

For each configuration, C, in α, denote:

$$max\text{-}head(C) = \max\{Head[i].seq \mid 0 \leq i \leq n-1\}$$

This is the maximal sequence number of an entry in the *Head* array in the configuration C. We abuse terminology and refer to $Head[i].seq$ as the sequence number of the ith entry of the *Head* array.

Inspecting the pseudocode reveals that the sequence number of each entry of the *Head* array is monotonically nondecreasing during the execution. Furthermore, we have the following lemma, whose proof is left as an exercise to the reader (Exercise 15.12):

Lemma 15.8 *If a processor performs ℓ iterations of its repeat loop, then max-head increases at least by ℓ.*

Algorithm 49 A nonblocking universal algorithm using consensus objects: code for processor p_i, $0 \leq i \leq n-1$.

Initially $Head[j]$ points to anchor, for all j, $0 \leq j \leq n-1$

```
1:  when inv occurs:                          // operation invocation, including parameters
2:      allocate a new opr record pointed to by point with point.inv := inv
3:      for j := 0 to n − 1 do                // find record with highest sequence number
4:          if Head[j].seq > Head[i].seq then Head[i] := Head[j]
5:      repeat
6:          win := decide(Head[i].after,point)          // try to thread your record
7:          win.seq := Head[i].seq + 1
8:          win.new-state, win.response := apply(win.inv,Head[i].new-state)
9:          Head[i] := win                  // point to the record at the head of the list
10:     until win = point
11:     enable the output indicated by point.response          // operation response
```

Therefore, if processor p_i performs an unbounded number of steps, then *max-head* is not bounded. This implies that *max-head* is nondecreasing during the execution; furthermore, if α is infinite, then *max-head* is not bounded.

The above observation can be used to show that the algorithm is nonblocking. Assume that some processor, p_i, performs an unbounded number of steps, without threading its operation to the list. Then *max-head* increases without bound, and therefore, the sequence numbers are increasing. It follows that other processors succeed in threading their operations to the list.

The algorithm is still not wait-free because the same processor might succeed in applying its operation again and again, locking all other processors out of access to the shared object.

15.3.3 A Wait-Free Algorithm Using Consensus Objects

As mentioned, the algorithm of the previous section is nonblocking, and it is possible that a processor will never get to thread its *opr* record to the linked list. To make the algorithm wait-free, we use the method of *helping*. This method means performing the operations of other processors, not letting them be locked out of access to the shared object.

The key to this idea is to know which processors are trying to apply an operation to the object. This is done by keeping an additional shared array $Announce[0..n-1]$; the ith entry of this array, $Announce[i]$, is a pointer to the *opr* record that p_i is currently trying to thread onto the list (see Fig. 15.6). Initially, all entries in *Announce* point to the anchor record, because no processor is trying to thread an operation.

Given that it is known which processors are trying to apply an operation, the first question is how to choose the processor to help, in a way that guarantees that this processor will succeed in applying its operation.

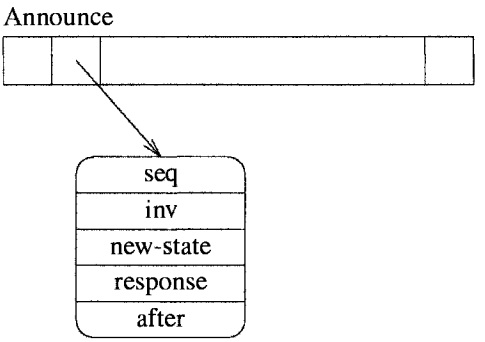

Fig. 15.6 Additional data structures for wait-free simulation using consensus objects.

The idea is to give priority, for each new sequence number, to some processor that has a pending operation. Priority is given in a round-robin manner, using the processors' ids; that is, if processor p_i has a pending operation, then it has priority in applying the kth operation, where $k = i \bmod n$. Thus if the sequence number at the head of the list is $k - 1$, where $k = i \bmod n$, then any competing processor checks whether p_i has a pending operation, and if so, it tries to thread p_i's operation to the list. Clearly, if all competing processors try to thread the same operation, they will succeed in doing so.

A final problem that arises is the issue of coordination between processors that concurrently try to thread the same record to the list. In the simple case in which operations are deterministic, the details of this coordination are relatively simple. In this case, it is safe for different processors to write the new state or the response, because they all write the same value (even if at different times). A processor detects that its operation has been applied to the object, that is, threaded onto the linked list, by seeing an assignment of a nonzero value to the sequence number of the record associated with its operation. Nondeterministic operations are considered in Section 15.3.5.

The pseudocode appears in Algorithm 50.

Fix some execution, α, of the algorithm. Proving that Algorithm 50 simulates the object is done as in the nonblocking algorithm.

Let us argue why the algorithm is wait-free. Assume that p_i tries to apply an operation to the object. Let C_1 be the first configuration after p_i announces its operation, in Line 2. As in the previous algorithm, for any configuration C of α, we denote by *max-head*(C) the maximal sequence number of an entry in the *Head* array, in C. As in the previous algorithm, *max-head* grows without bound, if some processor performs an unbounded number of steps in α.

Because *max-head* grows without bound, let C_2 be the first configuration after C_1 in which *max-head*(C_2) $= k = (i + 2) \bmod n$. We argue that p_i's operation is threaded by C_2. Assume not. Then when any processor p_j checks the if condition in Line 7 after configuration C_1, it finds that *Announce*[i].*seq* = 0. Therefore, all

Algorithm 50 A wait-free universal algorithm using consensus objects: code for processor p_i, $0 \leq i \leq n-1$.

Initially *Head*[j] and *Announce*[j] point to the anchor record, for all j, $0 \leq j \leq n-1$

```
1:  when inv occurs:                          // operation invocation, with parameters
2:      allocate a new opr record pointed to by Announce[i]
            with Announce[i].inv := inv and Announce[i].seq := 0
3:      for j := 0 to n − 1 do                // find highest sequence number
4:          if Head[j].seq > Head[i].seq then Head[i] := Head[j]
5:      while Announce[i].seq = 0 do
6:          priority := Head[i].seq+1 mod n   // id of processor with priority
7:          if Announce[priority].seq = 0     // check whether help is needed
8:              then point := Announce[priority]      // choose other record
9:              else point := Announce[i]             // choose your own record
10:         win := decide(Head[i].after,point)        // try to thread chosen record
11:         win.new-state, win.response := apply(win.inv,Head[i].new-state)
12:         win.seq := Head[i].seq + 1
13:         Head[i] := win                // point to the record at the head of the list

14:     enable the output indicated by win.response       // operation response
```

processors choose *Announce*[i] for the $(k+1)$st decision; by the agreement and validity properties of the consensus object, they all decide on *Announce*[i] and thread p_i's operation.

The above argument can be made more precise to bound the step complexity of the simulation. As in the algorithm of the previous section, after p_i performs n iterations of its while loop the value of *max-head* increases at least by n. Because each iteration of the while loop requires $O(1)$ steps by p_i, a configuration C_2 as above is reached after p_i performs $O(n)$ steps. This implies:

Theorem 15.9 *There exists a wait-free simulation of any object for n processors, using only n-processor consensus objects and read/write objects. Each processor completes any operation within $O(n)$ of its own steps, regardless of the behavior of other processors.*

The calculations in the above analysis charged one step for each invocation of the decide operation on the consensus object. A more detailed analysis of the cost should be based on the step complexity of the decide operation, in terms of operations on more basic objects.

This shows that Algorithm 50 is a wait-free simulation of an arbitrary object for n processors, using n-processor consensus objects. We can now state the general universality result.

Theorem 15.10 *Any object X with consensus number n is universal in a system with at most n processors.*

Proof. Algorithm 50 uses an arbitrary n-processor consensus object to wait-free simulate any object Y in a wait-free manner. We use objects of type X to wait-free simulate the n-processor consensus object. □

15.3.4 Bounding the Memory Requirements

We have presented a wait-free simulation of an arbitrary n-processor object using n-processor consensus objects and read/write objects. In this section, we show how to bound the memory requirements of the construction. Note that there are two types of memory unboundedness in this algorithm: The number of records used to represent an object as well as the values of the sequence numbers grow linearly, without bound, with the number of operations applied to the simulated object. Here, we describe how to control the first type of unboundedness—the number of records; handling the other type of unboundedness—the values of sequence numbers, is not treated here.

The basic idea is to *recycle* the records used for the representation of the object. That is, each processor maintains a pool of records belonging to it; for each operation, the processor takes some free record from its pool and uses it in the previous algorithm; eventually, a record can be reclaimed and reused for another operation. A record can be reclaimed if no processor is going to access it. The main difficulty is knowing which of the records already threaded on the list will not be accessed anymore and can be recycled.

To understand how this is done, we first inspect how records are accessed in Algorithm 50. Note that each of the records already threaded on the list belongs to some processor whose operation it represents. Such a record has an assigned sequence number, which remains fixed for the rest of the execution. Consider some record threaded on the list, belonging to processor p_i, with sequence number k; we refer to it as record number k.

Let p_j be a processor that may access record number k. It follows that $Head[j]$ after Line 4 is less than or equal to k, and therefore, p_j's record is threaded with sequence number $k + n$ or less. Note that p_j's record could be threaded by other processors helping it, yet p_j will not detect it (being inside the while loop) and will try to access record k on the list although its record is already threaded. However, once p_j detects that its record has been threaded, it will never access record k on the list anymore.

The above argument implies that the processors that may access record number k on the list are the processors whose records are threaded as numbers $k+1, \ldots, k+n$ on the list. These records do not necessarily belong to n different processors but may represent several operations by the same processor. Considered backwards, this means that if p_j's record is threaded as number k', p_j should release records number $k'-1, \ldots, k'-n$.

We add to the *opr* record type an array, $released[1, .., n]$, of binary variables. Before a record is used, all entries of the *released* array are set to *false*. If a record

Algorithm 51 A wait-free universal algorithm using consensus objects with bounded memory: code for processor p_i, $0 \leq i \leq n-1$.

Initially *Head*$[j]$ and *Announce*$[j]$ point to the anchor record, for all j, $0 \leq j < n$

```
1:  when inv occurs:                          // operation invocation, with parameters
2:      let point point to a record in Pool such that
            point.released[1] = ... =point.released[n] = true
            and set point.inv to inv
3:      for r := 1 to n do point.released[r] := false
4:      Announce[i] := point
5:      for j := 0 to n − 1 do                // find highest sequence number
6:          if Head[j].seq > Head[i].seq then Head[i] := Head[j]

7:      while Announce[i].seq = 0 do
8:          priority := Head[i].seq + 1 mod n     // id of processor with priority
9:          if Announce[priority].seq = 0         // check whether help is needed
10:             then point := Announce[priority] // choose other processor's record
11:             else point := Announce[i]        // choose your own record
12:         win := decide(Head[i].after,point)   // try to thread chosen record
13:         win.before := Head
14:         win.new-state, win.response := apply(win.inv, Head[i].new-state)
15:         win.seq := Head[i].seq + 1
16:         Head[i] := win                // point to the record at the head of the list

17:     temp := Announce[i].before
18:     for r := 1 to n do                        // go to n records before
19:         if temp ≠ anchor then
20:             before-temp := temp.before
21:             temp.released[r] := true          // release record
22:             temp := before-temp

23:     enable output indicated by Announce[i].response
```

has been threaded as number k on the list, then *released*$[r]$ = *true* means that the processor whose record was threaded as number $k + r$ on the list has completed its operation. When a processor's record is threaded as number k', it sets *released*$[r]$ = *true* in record $k' - r$, for $r = 1, \ldots, n$. When *released*$[r]$ = *true* for all $r = 1, \ldots, n$, then the record can be recycled.

To allow a processor to move backwards on the list of records, we restore the component *before* to the *opr* record type; when a record is threaded on the list, its *before* component points to the previous record threaded on the list. Now the list is doubly linked. The modified pseudocode appears as Algorithm 51.

We leave the correctness proof (linearizability and wait-freedom) to the reader as an exercise.

To calculate how many records a processor's pool should contain, we need to know how many unreleased records there may be, that is, records with *false* in some entry of their *released* array. Note that if *released*$[r] =$ *false* for record number k on the list, then the record number $k + r$ on the list belongs to an incomplete operation. Because each processor has at most one operation pending at a time, there are at most n records belonging to incomplete operations. Each of these records is responsible for at most n unreleased records. Thus, there are at most n^2 unreleased records. It is possible that all n^2 records belong to the same processor; therefore, each processor needs a pool of $n^2 + 1$ records, yielding a total of $O(n^3)$ records.

15.3.5 Handling Nondeterminism

The universal algorithms described so far assumed that operations on the simulated object are *deterministic*. That is, given the current state of the object and the invocation (the operation to be applied and its parameters), the next state of the object, as well as the return value of the operation, are unique. Many objects have this property, for example, queues, stacks, and read/write objects; however, there are object types with nondeterministic operations, for example, an object representing an unordered set with a *choose* operation returning an arbitrary element of the set. In this section, we outline how to modify the universal algorithm to simulate objects with nondeterministic operations.

For simplicity, we refer to the version of the algorithm that uses unbounded memory (Algorithm 50). Line 11 of this algorithm is where the new state and the response are calculated, based on the current state and the invocation. Because we assumed operations were deterministic, it is guaranteed that any processor applying the invocation to the current state will obtain the same new state and response. Thus it suffices to use read/write objects for the *new-state* and *response* fields of the *opr* record; even if processors write these fields at different times, they will write the same value.

When operations are nondeterministic, it is possible that different processors applying the invocation to the current state will obtain a different new state or response value. If we leave the *new-state* and *response* fields of the *opr* record as read/write objects, it is possible to get inconsistencies as different processors overwrite new (and different) values for the *new-state* or the *response* fields.

The solution is to reach consensus on the new state and the response. We modify the *opr* record type so that the new state and response value are stored jointly in a single consensus object. We replace the simple writing of the *new-state* and *response* fields (in Line 11) with a decide operation of the consensus object, using as input the local computation of a new state and response (using apply). The rest of the algorithm remains the same.

We leave the details of this algorithm, as well as its proof, to the reader as an exercise.

15.3.6 Employing Randomized Consensus

We can relax the Liveness condition in the definition of a linearizable shared memory (Section 9.1) to be probabilistic, that is, require operations to terminate only with high probability; this way we can define *randomized wait-free* simulations of a shared objects. Because randomized wait-free consensus algorithms can be implemented from read/write objects (see Chapter 14), they can replace the consensus objects in Algorithm 50. Thus there are randomized wait-free simulations of any object from read/write objects, and there is no hierarchy of objects if termination has to be guaranteed only with high probability.

Exercises

15.1 Prove that Algorithm 46 is a wait-free consensus algorithm for two processors. What happens if three processors (or more) use this algorithm?

15.2 Prove that Algorithm 47 is a wait-free consensus algorithm for any number of processors.

15.3 Prove that CN(test&set)$= 2$.

15.4 Prove that CN(stack)$= 2$.

15.5 Prove that CN(fetch&inc)$= 2$.

15.6 The wait-free consensus algorithm for two processors using a FIFO queue relies on the fact that the queue was nonempty initially. Present a two-processor wait-free consensus algorithm that uses two queues that are initially empty and read/write objects.

15.7 Show that the consensus number of an augmented queue, which allows peek operations, that is, reading the head of the queue without removing it, is infinite.

15.8 Show that for every integer $m \geq 1$, there exists an object with consensus number m.

Hint: Consider a variation of an augmented queue that can hold up to m values; once a processor attempts to enqueue the $(m+1)$st value, the queue "breaks" and returns a special $\perp$ response to every subsequent operation.

15.9 Show that consensus numbers also determine the existence of nonblocking simulations. That is, prove that if $CN(X) = m$ and $CN(Y) = n > m$, then there is no nonblocking simulation of Y by X in a system with more than m processors.

15.10 Prove the linearizability property of Algorithm 48.

15.11 Prove the linearizability property of Algorithm 49.

15.12 Prove Lemma 15.8.

15.13 Consider the following modification to Algorithm 50: First try to thread your own operation and only then try to help other processors. Show that the modified algorithm is not wait-free.

15.14 Consider the following modification to Algorithm 50: Add an iteration of the for loop (of Lines 3–4) inside the while loop (of Lines 5–13). What is the step complexity of the new algorithm? Are there situations in which this modification has improved step complexity?

15.15 Present a universal wait-free algorithm for simulating an n-processor object type with nondeterministic operations, using n-processor consensus objects; follow the outline in Section 15.3.5. Present the correctness proof for this algorithm.

15.16 Consider the same modification as in Exercise 15.14 to Algorithm 51: Add an iteration of the for loop (of Lines 5-6) inside the while loop (of Lines 7-16). What is the step complexity of the new algorithm? Are there situations in which this modification has improved step complexity?

15.17 Complete the correctness proof for Algorithm 51.

15.18 Show an execution of Algorithm 51, where n^2 records belonging to the same processor are not released.

15.19 A way to slightly reduce the memory requirements of Algorithm 51 is to have all processors use the same pool of records. Develop the details of this algorithm, which requires $O(n^2)$ records.

Chapter Notes

The results in this chapter indicate that there is a rich hierarchy of object types, and that some objects can be used to solve more problems than others. As mentioned in the introduction to this chapter, modern and proposed multiprocessors typically provide hardware primitives more powerful than just reading and writing. One would like to program at a higher level of abstraction than register operations, using shared objects of arbitrary types. As we have shown, if an object has a low consensus number, then it cannot be used, either alone or with additional read/write registers, to wait-free simulate an object with a high consensus number. Many common objects are universal and can be used to solve consensus among any number of processors, for example, compare&swap, or load-linked and store-conditional. Universal objects are desirable in hardware as they promise solutions for many other problems.

It is tempting to classify object types according to their consensus number. One way to do that, mentioned in passing before, is to organize object types into a *hierarchy*, where level n of the hierarchy contains exactly those objects whose consensus

number is n. This classification into levels of a hierarchy is meaningful only if object types at a higher level are somehow "stronger" than object types at lower levels.

The term "robust" has been applied to the notion of a meaningful hierarchy. A reasonable definition of robustness is that objects at a lower level cannot be used to simulate objects at a higher level. In more detail, a hierarchy is defined to be *robust* if any collection of object types $\mathcal{T}$, all at level k (or lower) of the hierarchy, *cannot* wait-free simulate an object type T that is at level $k+1$ (or higher) of the hierarchy. Jayanti [144] asked whether the wait-free hierarchy is robust and showed that if the consensus number is defined without allowing the use of read/write registers, or by allowing only a single object of the lower type, then the hierarchy is not robust. Our definition of the consensus number is consistent with this result, because it allows read/write registers and multiple objects of the lower type.

It turns out that the answer to the robustness question depends on additional aspects of the model. One aspect is whether operations of the object are deterministic or not; that is, whether the result of applying an operation to the object is unique, given the sequence of operations applied so far. Lo and Hadzilacos [172] showed that if objects are not deterministic, then the hierarchy is not robust. If the objects are deterministic, then the issue of robustness depends on some delicate assumptions concerning the relationships between the processors and the "ports" of the objects. Think of a port as a conduit through which operations can be applied to an object. An object may or may not be able to return different values, depending on which port is in use, and various rules concerning the binding of processors to ports are possible. The complete story has not yet been discovered; see a recent survey [105] for more information and references.

Another reasonable definition of robustness is that objects at a higher level can solve more problems than objects at a lower level. Specifically, consider two object types T and T', such that $CN(T) = m < CN(T') = n$. Under this interpretation, a robust hierarchy would mean that by using T' we can solve a strictly larger set of problems than by using T. However, such a hierarchy does not exist: It was shown by Rachman [224] that for any N, there exists an object type T, which can be accessed by $2N+1$ processors, whose consensus number is 1; furthermore, T can be used to solve 2-set consensus (defined in Section 16.1) among $2N+1$ processors. However, any number of N-consensus objects and read/write registers cannot be used to solve 2-set consensus among $2N+1$ processors. Thus T, an object type at level 1, can be used to solve the 2-set consensus problem, but an object at level N (N-consensus) cannot be used to solve the same problem. This anomalous behavior is possible because the number of processors is larger than the number for which N-consensus is universal.

Most of this chapter is based on Herlihy's work on impossibility and universality of shared objects [134]. Universality results for a specific consensus object (*sticky bits*) were presented by Plotkin [216]. Universality results using *load-linked* and *store-conditional* were presented by Herlihy [131]. Other general simulations were given by Prakash, Lee, and Johnson [219], by Shavit and Touitou [241], and by Turek, Shasha, and Prakash [254].

There have also been suggestions to modify the operating system in order to support more efficient nonblocking simulations of objects; interesting research in this direction was presented by Alemany and Felten [8] and by Bershad [51].

Distributed Computing: Fundamentals, Simulations and Advanced Topics, Second Edition
Hagit Attiya and Jennifer Welch
 ISBN: 0-471-45324-2

16

Problems Solvable in Asynchronous Systems

The impossibility result proved in Chapter 5 shows that consensus cannot be solved deterministically in failure-prone asynchronous systems. As shown in Chapter 15, the impossibility of solving consensus implies that many other important problems cannot be solved deterministically in failure-prone asynchronous systems. However, there *are* some interesting problems that can be solved in such systems.

In this chapter we survey several such problems. The first problem is *set consensus*, a weakening of the original consensus problem in which a fixed number of different decisions are allowed. We present a lower bound that relates the number of different decisions to the number of processor failures. The second problem is an alternative weakening of consensus, called *approximate agreement*, in which the decisions must lie in a sufficiently small range of each other. The third problem is *renaming*, in which processors must choose new identifiers for themselves. We also discuss *k-exclusion*, a fault-tolerant variant of mutual exclusion in which multiple processes can be in the critical section at a time. Solutions to the renaming problem can be used to solve a variant of k-exclusion, in which processes must each occupy a specific "place" in the critical section.

Throughout this chapter we assume the asynchronous shared memory model with crash failures. The maximum number of crash failures allowed in an admissible execution is denoted f. The simulation presented in Chapter 10 (Section 10.4) allows us to translate these results to the message-passing model, with $f < n/2$ crash failures.

16.1 K-SET CONSENSUS

The consensus problem for crash failures (studied in Chapter 5) required that all nonfaulty processors eventually decide on a single value, where that value is one of the original input values. We can loosen this problem statement to require only that the number of different values decided upon by nonfaulty processors be at most some quantity, say k, while still requiring that every decision value be some processor's original input. Obviously, this problem is only challenging to solve if the number of values, k, is less than the number of processors, n; otherwise, the trivial algorithm, in which every processor decides on its input value, is a solution to this problem that tolerates any number of failures. The original consensus problem can be viewed as a special case of this problem, where $k = 1$.

We now give a more formal definition of the *k-set consensus* problem. Each processor p_i has special state components x_i, the *input*, and y_i, the *output*, also called the *decision*. Initially x_i holds a value from some set of possible inputs and y_i is undefined. Any assignment to y_i is irreversible. A solution to the consensus problem must guarantee the following, in every admissible execution:

Termination: For every nonfaulty processor p_i, y_i is eventually assigned a value.

k-Agreement: $|\{y_i : p_i \text{ is nonfaulty}, 0 \leq i \leq n-1\}| \leq k$. That is, the set of decisions made by nonfaulty processors contains at most k values.

Validity: If y_i is assigned, then $y_i \in \{x_0, \ldots, x_{n-1}\}$, for every nonfaulty processor p_i. That is, the output of a nonfaulty processor is one of the inputs.

In this section, we show that the k-set consensus problem is solvable in an asynchronous system subject to crash failures as long as f, the number of failures to be tolerated, is at most $k-1$. Afterwards, we show that this bound is tight. In the wait-free case, where the number of failures to be tolerated is $n-1$, it follows that k-set consensus is possible only if $k = n$; in this case, the trivial algorithm, in which each processor decides on its own input, solves the problem.

We now describe a simple algorithm for k-set consensus. This algorithm, like most algorithms in this chapter, is presented for the shared memory model, and employs an atomic snapshot object, as defined in Chapter 10.

The snapshot object initially holds an empty indication in each segment. Each processor p_i writes its input x_i to its segment in the snapshot object; then, it repeatedly scans the snapshot object until it sees at least $n-f$ nonempty segments. Processor p_i decides on the minimum value contained in its last scan of the snapshot object. The pseudocode appears as Algorithm 52. The name of the atomic snapshot object is not explicitly included in the calls to the update and scan procedures.

Theorem 16.1 *Algorithm 52 solves k-set consensus in the presence of f crash failures, where $f \leq k-1$.*

Proof. We show that the above algorithm solves k-set consensus. Fix an admissible execution of the algorithm.

Algorithm 52 k-set consensus algorithm for $f < k$ failures: code for processor p_i, $0 \leq i \leq n-1$.

```
1: update_i(x)                          // write input value to snapshot segment
2: repeat
3:     values := scan_i()
4: until values contains at least n − f nonempty segments
5: y := min(values)                     // decide on smallest element of values
```

Consider any nonfaulty processor p_i. Termination follows since there are at most f crash failures. Because processors fail only by crashing, every value that is ever put into a processor's local variable *values* is the input value of some processor. Therefore, p_i decides on the input value of some processor, and the validity condition is satisfied.

Let S be the set of all values v such that v is the minimum value contained in the final scan of the snapshot object by some nonfaulty processor. We now prove that the set S contains at most $f+1$ distinct values. Suppose, in contradiction, that S contains at least $f+2$ distinct values, and let v be the largest among them. It follows that there are $f+1$ distinct input values that are strictly smaller than v, say $x_{i_0}, \ldots, x_{i_f}$ (the input values of $p_{i_0}, \ldots, p_{i_f}$). Let p_i be a nonfaulty processor such that v is the minimum value contained in its final scan. Thus p_i's final scan does not include the input values of $p_{i_0}, \ldots, p_{i_f}$, but then p_i's last scan missed at least $f+1$ input values, which contradicts the code.

Since the set S is exactly the set of decision values of nonfaulty processors, we have just shown that the number of nonfaulty decisions is at most $f+1$, which is at most k, which implies the k-agreement condition. □

The obvious next question is, what happens when the number of failures exceeds k? Is k-set consensus still solvable? As we shall see next, the answer is negative.

Lower Bound

We now show that there is no algorithm for solving k-set consensus in the presence of $f \geq k$ failures. For $k = 1$, this amounts to the impossibility of solving the consensus problem, proved in Chapter 5. We first prove the lower bound for $k = 2$ and $n = 3$ and later discuss how to extend the result for other values of k and n.

For the purpose of the lower bound, we assume that all communication is via an atomic snapshot object[1] with a single segment for each processor and that an algorithm consists of an alternating sequence of updates to the processor's segment

[1] Almost the same proof works if processors communicate by ordinary read and write operations; we assume a snapshot object because it slightly simplifies matters, and to be consistent with the other algorithms in this chapter.

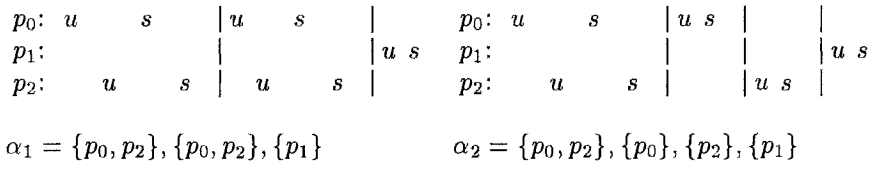

Fig. 16.1 Two block executions α_1 and α_2; u denotes an update operation and s denotes a scan operation; perpendicular lines separate blocks.

and scans of the snapshot object. Moreover, a processor maintains a local step counter that is written in each update.

A *block* B is an arbitrary nonempty set of processor ids, that is, B is a subset of $\{0, \ldots, n-1\}$. A block B induces a schedule fragment in which, first, each processor in B updates its segment of the snapshot object, and then each processor in B scans the snapshot object. For concreteness, we will assume that the updates are done in increasing order of processor id, as are the scans; however, the internal order of the update operations and the internal order of the scan operations are immaterial, as long as all update operations precede all scan operations.

Fix some k-set consensus algorithm A. A *block execution* of A is an execution whose schedule is a sequence of schedule fragments, each of which is induced by a block, and whose initial configuration has each processor starting with its own id as input, that is, p_i starts with i, $0 \leq i \leq n-1$. Thus a block execution, α, is completely characterized by the sequence of blocks $B_1, B_2, \ldots, B_l$ that induces its schedule. We abuse notation and write $\alpha = B_1, B_2, \ldots, B_l$. For notational convenience, we do not include steps of processors once they have decided.

The lower bound proof considers only admissible block executions of A in which there are no failures. Of course, A has to work correctly for all (admissible) executions, including those with failures, and all inputs; in fact, it is the requirement that A can tolerate failures that allows us to make crucial deductions about block executions with no failures.

Although they are very well-structured, block executions still contain uncertainty, because a processor does not know exactly which processors are in the last block. Specifically, if p_i is in B_k and observes an update by another processor p_j, p_i does not know whether p_j's update is in B_{k-1} or in B_k. Consider, for example, the block executions in Figure 16.1. First note that p_1 and p_2 do not distinguish between α_1 and α_2. However, because each processor increments a counter in each of its steps and includes it in its segment, p_0 distinguishes between α_1 and α_2 — in α_1, p_0 reads the second update of p_2, whereas in α_2, it reads the first update of p_2.

In Chapter 4 we defined the notion of similarity between configurations (Definition 4.1); here, we define an extended notion of similarity between block executions.

Let $\alpha = B_1, \ldots, B_l$; the *view of processor* p_i *after block* B_k is p_i's state and the state of all shared variables in the configuration after prefix of α that corresponds to $B_1, \ldots, B_k$. The *view of processor* p_i *in* α, denoted $\alpha|p_i$, is the sequence containing

p_i's view after each block in which p_i appears; $\alpha|p_i$ is empty if p_i does not appear in any block.

Two block executions, α and α', are *similar to some set of processors* P, denoted $\alpha \overset{P}{\sim} \alpha'$, if for any processor $p_i \in P$, $\alpha|p_i = \alpha'|p_i$. When $P = \{p_0, \ldots, p_{n-1}\} - \{p_j\}$, we use $\alpha \overset{\neg p_j}{\sim} \alpha'$ as a shorthand; for the nontrivial case $\alpha \neq \alpha'$, this means that p_j is the only processor distinguishing between α and α'.

We say that p_j is *unseen* in a block execution α if there exists $k \geq 1$, such that $p_j \notin B_r$ for every $r < k$ and $B_r = \{p_j\}$ for every $r \geq k$. Intuitively, this means that p_j's steps are taken after all other processors decide and none of them ever sees a step by p_j. Note that at most one processor is unseen in a block execution. For example, p_1 is unseen in executions α_1 and α_2 of Figure 16.1.

It is crucial to this proof to understand that it is *possible* to have an unseen processor p_i in an admissible execution of the algorithm A. The reason is that A is supposed to be able to tolerate the failure of (at least) one processor: If p_i takes no steps initially, the other processors must be prepared for the possibility that p_i has failed and thus they must decide without communicating with p_i. After the remaining processors have decided, it is possible for p_i to start taking steps.

Lemma 16.2 *If a processor p_j is unseen in a block execution α then there is no block execution $\alpha' \neq \alpha$ such that $\alpha \overset{\neg p_j}{\sim} \alpha'$.*

Proof. Let $\alpha = B_1, B_2, \ldots$. If p_j is unseen in α, then there exists some k such that $p_j \notin B_r$ for every $r < k$ and $B_r = \{p_j\}$ for every $r \geq k$. Assume, by way of contradiction, that there is a block execution $\alpha' = B'_1, B'_2, \ldots \neq \alpha$, such that $\alpha \overset{\neg p_j}{\sim} \alpha'$. Let l be the minimum index such that $B_l \neq B'_l$, that is, the first block in which α and α' differ.

If $l \geq k$ then $B_l = \{p_j\}$ in α, whereas in α', B'_l must include some other processor, p_i; however, p_i distinguishes between α and α', a contradiction. If $l < k$ then $p_j \notin B_l$. Since no processor in B_l distinguishes between α and α', then the same processors must be in B'_l, again contradicting the assumption that $B_l \neq B'_l$. □

If p_j is not unseen in α, then it is *seen*. Formally, a processor p_j is *seen in* k, if $p_j \in B_k$, and some processor $p_i \neq p_j$ is in B_r, for some $r \geq k$. If p_j is seen, then p_j is *last seen in* k, if k is the largest index in which p_j is seen. For example, p_0 and p_2 are seen in both α_1 and α_2 (Fig. 16.1); in α_1, p_0 is last seen in B_2.

Lemma 16.3 *If p_j is seen in a block execution α, then there is a unique block execution $\alpha' \neq \alpha$ such that $\alpha' \overset{\neg p_j}{\sim} \alpha$.*

Proof. Assume that in $\alpha = B_1, B_2, \ldots$, p_j is last seen in k. It is possible that p_j appears in blocks later than B_k, but in this case α can be written as $\alpha = B_1, \ldots, B_t, \{p_j\}, \ldots, \{p_j\}$. Define a block execution α' as follows:

1. If $B_k \neq \{p_j\}$, then take p_j into an earlier block by itself, that is,

$$\alpha' = B_1, \ldots, B_{k-1}, \{p_j\}, B_k - \{p_j\}, B_{k+1}, \ldots, B_t, \{p_j\}, \ldots, \{p_j\}$$

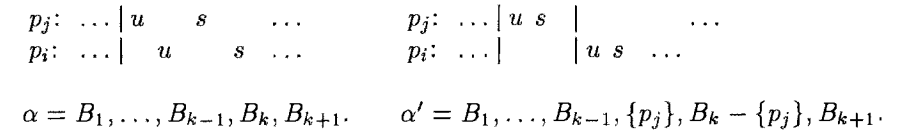

Fig. 16.2 Illustration for first case in proof of Lemma 16.3, $B_k = \{p_i, p_j\}$.

where the number of final blocks consisting only of p_j is sufficient for p_j to decide (see Fig. 16.2).

2. If $B_k = \{p_j\}$, then merge p_j with the next block, that is,

$$\alpha' = B_1, \ldots, B_{k-1}, \{p_j\} \cup B_{k+1}, B_{k+2}, \ldots, B_t, \{p_j\}, \ldots, \{p_j\}$$

where the number of final blocks consisting only of p_j is sufficient for p_j to decide.

Clearly, $\alpha' \overset{\neg p_j}{\sim} \alpha$. Since p_j distinguishes between α and α' (Exercise 16.2), we have that $\alpha \neq \alpha'$. We now show that α' is unique.

If $B_k \neq \{p_j\}$ (Case (1)), then clearly, there is another processor $p_i \in B_k$. If $B_k = \{p_j\}$ (Case (2)), then there is another processor $p_i \in B_{k+1}$: B_{k+1} is not empty and does not include p_j. Moreover, if $p_j \in B_r$, for some $r > k$, then $B_{r'} = \{p_j\}$ for every $r' \geq r$ (otherwise, p_j would be last seen in $k-1$ of α).

In both cases, for any block execution α'' that is neither α nor α', p_i distinguishes between α and α'', which proves the uniqueness of α'. □

Now construct a graph $\mathcal{B}_n$. The nodes of $\mathcal{B}_n$ correspond to block executions with n processors; since the algorithm is wait-free, there is a finite number of block executions and, therefore, the graph has a finite number of nodes (Exercise 16.5). There is an edge between two nodes corresponding to block executions α and α' if and only if $\alpha \overset{\neg p_j}{\sim} \alpha'$, for some processor p_j; the edge is labeled with p_j. For example, the part of $\mathcal{B}_3$ for the two executions of Figure 16.1 has two nodes, one for α_1 and one for α_2, and an edge between them labeled with p_0 (Fig. 16.3).

By Lemma 16.2, a node with degree $n-1$ corresponds to a block execution in which some processor p_i is unseen.

By Lemma 16.3, the degree of a node in $\mathcal{B}_n$ that corresponds to a block execution without an unseen processor must be n.

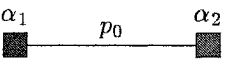

Fig. 16.3 Part of $\mathcal{B}_3$ for the executions of Figure 16.1.

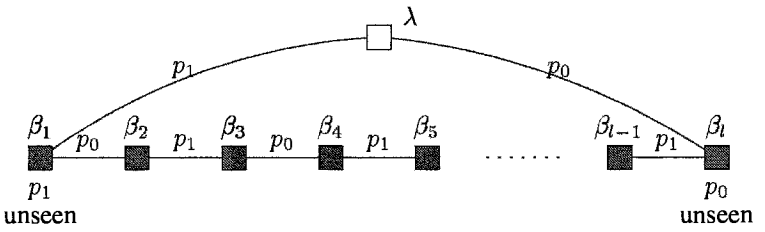

Fig. 16.4 The graph for block executions of two processors, $\hat{\mathcal{B}}_2$, with the imaginary node, λ.

We now color the edges of $\mathcal{B}_n$. If there is an edge between executions α and α' labeled with p_j then, by definition, $\alpha \overset{\neg p_j}{\sim} \alpha'$. Therefore, all processors other than p_j, namely, $p_0, \ldots, p_{j-1}, p_{j+1}, \ldots, p_{n-1}$, decide on the same values in α and in α'. The edge is colored with this set of decisions. Consider, for example, the executions of Figure 16.1; if p_1 decides on 1 and p_2 decides on 2 then the edge in Figure 16.3 is colored $\{1, 2\}$.

The discussion so far has applied to any number of processors; consider now the very simple case of a system with two processors. In this case, we can list all block executions; for example, if each processor (always) takes exactly two steps before deciding, then some of the block executions are:

$$
\begin{aligned}
& \beta_1 = \{p_0\}, \{p_0\}, \{p_1\}, \{p_1\} \\
\overset{\neg p_0}{\sim}\ & \beta_2 = \{p_0\}, \{p_0, p_1\}, \{p_1\} \\
\overset{\neg p_1}{\sim}\ & \beta_3 = \{p_0\}, \{p_1\}, \{p_0\}, \{p_1\} \\
\overset{\neg p_0}{\sim}\ & \beta_4 = \{p_0\}, \{p_1\}, \{p_0, p_1\} \\
\overset{\neg p_1}{\sim}\ & \beta_5 = \{p_0\}, \{p_1\}, \{p_1\}, \{p_0\} \\
& \vdots \\
\overset{\neg p_0}{\sim}\ & \beta_{l-1} = \{p_1\}, \{p_1, p_0\}, \{p_0\} \\
\overset{\neg p_1}{\sim}\ & \beta_l = \{p_1\}, \{p_1\}, \{p_0\}, \{p_0\}
\end{aligned}
$$

Note that there are two nodes with degree 1—one for the single execution in which p_0 is unseen and one for the single execution in which p_1 is unseen. We add an imaginary node, λ, with edges to these nodes, labeled with the unseen processor. Let $\hat{\mathcal{B}}_2$ be $\mathcal{B}_2$ with the imaginary node. Figure 16.4 shows $\hat{\mathcal{B}}_2$.

In $\mathcal{B}_2$, the color of an edge is a single value—the decision of a processor not distinguishing between the two incident executions. The coloring of edges extends naturally to the edges adjacent to λ; an edge labeled with p_i is colored with the decision of p_{1-i}, the other processor. We concentrate on edges in $\hat{\mathcal{B}}_2$ colored with $\{0\}$ and define the *restricted degree* of a node v to be the number of edges colored with $\{0\}$ that are incident to v.

Consider a non-imaginary node v with one incident edge colored with 0 and another incident edge colored with 1 (that is, the restricted degree of v is exactly 1). Node v corresponds to an execution in which one processor decides 0 and the other processor decides 1. That is:

Lemma 16.4 *If the restricted degree of a (non-imaginary) node is 1, then the node corresponds to an execution in which* $\{0, 1\}$ *are decided.*

Clearly, the edge incident to the imaginary node, λ, labeled with p_1, is colored with $\{0\}$; the other incident edge, labeled with p_0, is colored with $\{1\}$. Thus the restricted degree of λ is exactly 1. The sum of the restricted degrees of all nodes must be even, because each edge is counted twice in the summation. Thus there is an odd number of non-imaginary nodes with odd restricted degree, that is, there is an odd number of non-imaginary nodes with restricted degree 1. That is:

Lemma 16.5 *There is an odd number of executions in which* $\{0, 1\}$ *are decided.*

Therefore, there is at least one execution in which $\{0, 1\}$ are decided. This provides an alternative proof for the impossibility of consensus; moreover, this can be used to prove the impossibility of wait-free three-processor algorithms for 2-set consensus, as shown next.

Let us now consider the case of three processors, p_0, p_1 and p_2. Recall that in block executions p_i starts with input i, namely, 0, 1, or 2, and by the problem definition, processors must decide on at most two different values and the decisions must be a subset of the inputs.

We consider the graph $\mathcal{B}_3$, defined as above, and add an imaginary node λ, with an edge to each node with degree 2, corresponding to a block execution with an unseen processor; the edge is labeled with the unseen processor. The extended graph is denoted $\hat{\mathcal{B}}_3$. After λ is added, the degree of each non-imaginary node is exactly 3; each of its adjacent edges is labeled with a different processor. In the same manner, the additional edges are colored with the set containing the decisions of all processors but the unseen processor.

For three processors, we concentrate on edges colored with the pair $\{0, 1\}$; this means that the pair of processors that do *not* distinguish between the two adjacent executions decide on 0 and on 1 (in both executions). Similarly to the two-processor case, the *restricted degree* of a node v is the number of edges colored with $\{0, 1\}$ that are incident to v. Exercise 16.6 asks you to show that a (non-imaginary) node cannot have restricted degree 3, that is, it cannot have three incident edges colored with $\{0, 1\}$.

Therefore, the restricted degree of a node is at most 2. As in the case of two processors, the restricted degree is interesting because we can prove:

Lemma 16.6 *If the restricted degree of a (non-imaginary) node is 1, then the node corresponds to an execution in which* $\{0, 1, 2\}$ *are decided.*

Proof. Let α be the block execution corresponding to a node v. Without loss of generality, assume that the single incident edge colored with $\{0, 1\}$ is labeled by

processor p_0 and that p_1 decides 0 and p_2 decides 1 in α. We argue that p_0 decides 2 in α.

If p_0 decides 0 in α, then consider the edge incident to v labeled with p_1. This edge must exist since each node has n an adjacent edges, each labeled with a different processor. It must be colored with $\{0, 1\}$ (the decisions of p_0 and p_2). Similarly, if p_0 decides 1 in α, then consider the edge incident to v and labeled with p_2; it must be colored with $\{0, 1\}$ (the decisions of p_0 and p_1). In both cases, the restricted degree of v must be 2. □

What is the restricted degree of the imaginary node? This is the number of block executions with an unseen processor in which $\{0, 1\}$ are decided by the seen processors. By the validity condition, these can only be executions in which p_2 is unseen. Thus these are block executions in which p_0 and p_1 run on their own, not seeing p_2 at all, and p_2 runs after they decide. It can be shown (Exercise 16.8) that these executions have a one-to-one correspondence with all two-processor block executions, as captured by $\mathcal{B}_2$. By Lemma 16.5, there is an odd number of two-processor block executions in which $\{0, 1\}$ are decided. Therefore, the restricted degree of λ is odd.

Because the sum of restricted degrees of nodes in the extended graph $\hat{\mathcal{B}}_3$ (including λ) is even, there must be an odd number of non-imaginary nodes with odd restricted degree. Because the restricted degree of a node is at most two, it follows that an odd number of nodes have restricted degree 1. Therefore, at least one node has restricted degree 1. By Lemma 16.6, this node corresponds to an execution in which $\{0, 1, 2\}$ are decided, which proves:

Theorem 16.7 *There is no wait-free algorithm for solving the 2-set consensus problem in an asynchronous shared memory system with three processors.*

For wait-free algorithms, the lower bound for any value of k is proved by considering $\mathcal{B}_k$, colored as before. Define the *restricted degree* of a node to be the number of edges colored with $\{0, \ldots, k-2\}$ that are incident on the node. The above combinatorial argument can be extended, by induction, to show that an odd number of nodes has restricted degree 1 (Exercise 16.10). The lower bound then follows from the natural extension of Lemma 16.6 for $\mathcal{B}_k$ (Exercise 16.11).

The lower bound can be extended to any number of failures $f \geq k$, in the same manner that the impossibility of consensus is extended from wait-free algorithms to any number of failures (Chapter 5). The simulation of Section 5.3.2 needs to be extended to work for any number of simulating processors (not just two); more details appear in Exercise 16.12 and the chapter notes.

Finally, the simulation of atomic snapshot objects from read/write registers (Algorithm 30), together with the simulation of read/write registers in message-passing systems (described in Section 5.3.3), imply that the same lower bounds hold for asynchronous message-passing systems, as long as $f > n/2$. If $f \leq n/2$, this problem, as well as all the other problems in this chapter, cannot be solved in asynchronous message-passing systems; the proof is similar to proof of Theorem 10.22.

16.2 APPROXIMATE AGREEMENT

The approximate agreement problem is another weakening of the standard consensus problem, which, like k-set consensus, admits fault-tolerant solutions in asynchronous systems. Instead of allowing limited disagreement in terms of the number of different values decided, a *range* is specified in which decision values must fall.

Processor p_i's input value is denoted x_i and its output value is denoted y_i, $0 \leq i \leq n-1$. Input and output values are real numbers. The following conditions should be satisfied in every admissible execution by a solution to the *ϵ-approximate agreement* problem, for some positive real number ϵ:

Termination: For every nonfaulty processor p_i, y_i is eventually assigned a value.

ϵ-Agreement: For all nonfaulty processors p_i and p_j, $|y_i - y_j| \leq \epsilon$. That is, all nonfaulty decisions are within ϵ of each other.

Validity: For every nonfaulty processor p_i, there exist processors p_j and p_k such that $x_j \leq y_i \leq x_k$. That is, every nonfaulty decision is within the range of the input values.

Below, we present two wait-free algorithms for approximate agreement: a simple algorithm that depends on knowing the range of possible input values and an adaptive algorithm that does not require this knowledge.

16.2.1 Known Input Range

We now present an algorithm that solves the approximate agreement problem for up to $n-1$ failures. The algorithm proceeds in a series of asynchronous rounds. In each round, processors exchange values and apply an averaging function (specifically, computing the midpoint) to the values exchanged in order to compute new values, which are used in the next round. Values are exchanged by having each processor first update its segment of an atomic snapshot object and then scan the snapshot object. The exchange of values is asymmetric in that a fast process might see many fewer values in the snapshot object than a slow one; however, it is guaranteed to see at least one (its own).

As will be shown, each round reduces the spread of the values held by processors by a factor of 2. The number of rounds required until the spread is within the specified ϵ is the log of the range of the inputs divided by ϵ. The intuition behind this calculation is that the number of factor-of-2 reductions required to shrink the spread from its original range to ϵ is the log (base 2) of the ratio of the old and new ranges. Later, we discuss how to modify the algorithm to work with an unknown input range.

For simplicity of presentation, the algorithm uses a separate snapshot object for each round r, ASO_r. Initially each segment in ASO_r holds an empty indication. The pseudocode appears as Algorithm 53. Given a nonempty set X, the function range(X) returns the interval $[\min(X), \max(X)]$, and the function spread(X) returns the length of this interval, that is, $\max(X) - \min(X)$; the function midpoint(X) returns the middle of range(X), that is, $\frac{1}{2}(\min(X) + \max(X))$.

Algorithm 53 Asynchronous round $r \geq 1$ of wait-free ϵ-approximate agreement algorithm for known input range: code for processor p_i, $0 \leq i \leq n-1$.

Initially $v = x$ and $maxRound = \lceil \log_2 \frac{D}{\epsilon} \rceil + 1$,
where D is the maximal spread of inputs

1: update$_i$(ASO$_r$,v)
2: $values[r]$:= scan$_i$(ASO$_r$)
3: v := midpoint($values[r]$)
4: if $r = maxRound$ then $y := v$ and terminate // decide

The algorithm uses a separate snapshot object for each round, but they can be replaced with a single snapshot object, in which each processor writes the concatenation of its values for all rounds so far (see Exercise 16.13).

The following lemmas are with respect to an arbitrary admissible execution of the algorithm.

For each $r \geq 1$ and each processor p_i, denote by V_i^r the value of p_i's local variable $values[r]$ after Line 2 of asynchronous round r. For each $r \geq 1$, denote by U^r the set of all values ever written to ASO$_r$; this can be by either faulty or nonfaulty processors. Denote by U^0 the set of input values of all processes.

Let M be the value of *maxRound*. Note that U^r is not empty, for every r, $0 \leq r \leq M$. The key for the correctness of the algorithm is the next lemma.

Lemma 16.8 *For every asynchronous round r, $0 \leq r < M$, there exists a value $u \in range(U^r)$, such that $\min(U^{r+1}) \geq (\min(U^r) + u)/2$ and $\max(U^{r+1}) \leq (\max(U^r) + u)/2$ (see Fig. 16.5).*

Proof. Let u be the first value written (in Line 1) with round number r, by some processor p_j. We argue that u satisfies the claim of the lemma.

By the properties of the atomic snapshot object, each processor that participates in round $r+1$ reads u when calculating its value for round $r+1$. This holds since if p_j overwrites u, it is with a round number larger than r, so scans that are linearized after p_j overwrites u will be used for a round strictly bigger than $r+1$. Thus $u \in V_i^r$, for any processor p_i calculating a value v_i^{r+1} for round $r+1$. The lemma follows by proving that $(\min(U^r) + u)/2 \leq v_i^{r+1} \leq (\max(U^r) + u)/2$, which we leave as an exercise to the reader (see Exercise 16.16). □

As Figure 16.5 makes obvious, the above lemma implies:

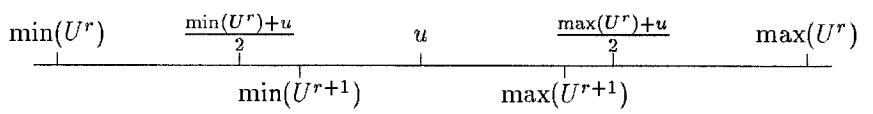

Fig. 16.5 Illustration for Lemma 16.8.

Lemma 16.9 *For every* r, $0 \leq r < M$, $\text{range}(U^{r+1}) \subseteq \text{range}(U^r)$.

Moreover, the spread of values is reduced by a factor of 2 at each asynchronous round.

Lemma 16.10 *For every* r, $0 \leq r < M$, $\text{spread}(U^{r+1}) \leq \frac{1}{2}\text{spread}(U^r)$.

Theorem 16.11 *Algorithm 53 solves wait-free ϵ-approximate agreement when the input range is known.*

Proof. Fix an admissible execution of the algorithm.

Termination follows since each processor performs at most *maxRound* asynchronous rounds and each asynchronous round completes within a finite number of steps.

To prove validity, consider some nonfaulty processor p_i. By repeated application of Lemma 16.9, its decision y_i is in the range of all the input values.

We now consider ϵ-agreement. Consider any two nonfaulty processors p_i and p_j. By definition, $maxRound = \lceil \log_2(D/\epsilon) \rceil$; clearly, $maxRound \geq \log_2(\text{spread}(U^0)/\epsilon)$.

By repeated application of Lemma 16.10,

$$\text{spread}(U^{maxRound}) \leq \text{spread}(U^0) \cdot 2^{-maxRound}$$

Substituting the above lower bound on *maxRound* shows that

$$\text{spread}(U^{maxRound}) \leq \epsilon$$

By the code, p_i's decision, y_i, and p_j's decision, y_j, are in $U^{maxRound}$. Hence, $|y_i - y_j| \leq \epsilon$. □

We can state an explicit bound on the number of steps taken by a nonfaulty processor.

Theorem 16.12 *A nonfaulty processor performs $O(\lceil \log_2(D/\epsilon) \rceil)$ scan and update operations on a snapshot object before deciding, in any admissible execution of Algorithm 53.*

16.2.2 Unknown Input Range

Algorithm 53 depends on knowing D, an upper bound on the spread of input values. Such a bound is not always available, and even when it is available, the bound can be very large compared with the actual spread of input values in the execution. We now describe an algorithm that does not rely on knowing D.

A close look at the algorithm reveals that what we really need is a bound on the spread of inputs in the execution. A first idea would be to modify the current algorithm so it calculates the number of rounds dynamically at each round, based on the spread of the inputs of processors that have started the execution so far. However, consider the case where some processor, say p_0, takes a solo execution in which it

Algorithm 54 Wait-free ϵ-approximate agreement algorithm for unknown input range: code for processor p_i, $0 \leq i \leq n-1$.

```
1: update_i(⟨x, 1, x⟩)
2: repeat
3:     ⟨x_0, r_0, v_0⟩, ..., ⟨x_{n-1}, r_{n-1}, v_{n-1}⟩ := scan_i()
4:     maxRound := log_2(spread(x_0, ..., x_{n-1})/ε)  // assume that log_2 0 = -∞
5:     r_max := max{r_0, ..., r_{n-1}}
6:     values := {v_j | r_j = r_max, 0 ≤ j ≤ n-1}
7:     update_i(⟨x, r_max + 1, midpoint(values)⟩)
8: until r_max ≥ maxRound
9: y := midpoint(values);                                   // decide
```

writes its input, executes some number of rounds, K, and then decides, without p_1 taking any steps. It can be shown that, in this case, p_0 must decide on its input, x_0 (see Exercise 16.15). Suppose after p_0 decides, processor p_1 starts a solo execution with an input x_1, such that $|x_0 - x_1| > \epsilon \cdot 2^{K+2}$; p_0 writes its input, reads x_0, x_1 and calculates a number of rounds to execute, $K' > K + 2$. Note that by asynchronous round $K+1$, p_1's preference is still more than ϵ away from x_0. Later, when executing asynchronous round $K + 2$, p_1 reads only its own preference for round $K + 1$, so p_1's preference remains the same until it decides. Thus p_1 decides on a value that is more than ϵ away from x_0, contradicting the ϵ-agreement property.

To avoid this problem, a processor does not go from one round to the next round, but rather skips to one more than the maximal round r that it observes in a scan, taking the midpoint of the values already written for this round r (and possibly ignoring its own input value).

The pseudocode appears as Algorithm 54.

The correctness proof assumes an arbitrary admissible execution of the algorithm.

We use the same notation of U^r and V_i^r as in the proof of the previous approximate agreement algorithm, but with slightly modified definitions because space is reused. For $r \geq 1$, U^r is the set of all values ever written (in Line 1 or Line 7) to the atomic snapshot object with round number r (middle element of the triple). U^0 is defined to be the set of input values of all processors. For $r \geq 1$, V_i^r is the value of p_i's variable *values* after executing Line 6 with $r_{\max} = r$. (If p_i never has its $r_{\max}$ variable equal to r, then V_i^r is undefined.)

Let M be the largest r such that U^r is not empty. Convince yourself that U^r is not empty, for every r, $1 \leq r \leq M$. The proof of Lemma 16.8 remains almost exactly the same (except that u can be written either in Line 1 *or* in Line 7 of the algorithm). Therefore, we can derive Lemma 16.9 and Lemma 16.10.

Theorem 16.13 *Algorithm 54 solves wait-free ϵ-approximate agreement when the input range is unknown.*

Proof. Fix an admissible execution of the algorithm.

To show that termination holds, we must show that no processor can keep increasing *maxRound* forever. *maxRound* only can increase if another processor starts

executing the algorithm, thus increasing the spread of observable inputs. Since there is a finite number of processes, *maxRound* can only be increased finitely many times.

Validity follows in a manner similar to the proof for Theorem 16.11.

We consider ϵ-agreement. Let R be the smallest round in which some nonfaulty processor p_i decides. We claim that spread$(U^R) \leq \epsilon$. By Lemma 16.9, for any round R' such that $R < R' \leq M$, range$(U^{R'}) \subseteq$ range(U^R), which, together with the claim, implies ϵ-agreement.

We now verify the claim that spread$(U^R) \leq \epsilon$. Consider some value $v_j^r \in U^R$ written for round R by p_j. If p_j writes its input value (i.e., performs its first update) before p_i computes *maxRound* for the last time, then the claim follows as in the proof of Theorem 16.11. If, on the other hand, p_j writes its input value after p_i computes *maxRound* for the last time, then the maximal round number seen by p_j in its first scan is at least R. (In this case, p_j ignores its input.) Thus, the value written by p_j for any round greater than or equal to R is in the range of the values written so far for round R, which proves the claim. □

16.3 RENAMING

The coordination problems considered so far in this chapter—k-set consensus and approximate agreement—require processors to decide on values that are close together. We now present a problem in which processors should decide on *distinct* values, but in a small range.

The *renaming problem* considers a situation in which processors start with unique names from a large domain (the *original names*) and, for some reason, they need to shrink it. Thus each processor should pick a *new name* from some small name space $[1..M]$. Denote by y_i the new name chosen by processor p_i; the main requirements of the renaming problem are:

Termination: For every nonfaulty processor p_i, y_i is eventually assigned a value.

Uniqueness: For all distinct nonfaulty processors p_i and p_j, $y_i \neq y_j$.

The goal is to minimize M, the size of the new name space. A superficial solution is to let processors choose their index, that is, processor p_i takes i as its new name; the new name space is of size n. Yet this solution is not good if the indices are larger than the actual number of processors. To rule out this solution, we make the following additional requirement:

Anonymity: The code executed by processor p_i with original name x is exactly the same as the code executed by processor p_j with original name x.

The uniqueness condition implies that M must be at least n. Here we show renaming algorithms with $M = n + f$, where f is the number of crash failures to be tolerated.

Algorithm 55 Wait-free renaming algorithm: code for processor p_i, $0 \leq i \leq n-1$.

```
1:  s := 1
2:  while true do
3:      update_i(⟨x, s⟩)
4:      (⟨x_0, s_0⟩, ..., ⟨x_{n-1}, s_{n-1}⟩) := scan_i()
5:      if s = s_j for some j ≠ i, then
6:          let r be the rank of x in {x_j ≠ ⊥ | 0 ≤ j ≤ n − 1}
7:          let s be the rth integer not in {s_j ≠ ⊥ | 0 ≤ j ≠ i ≤ n − 1}
8:      else
9:          y := s                                  // decide on s as new name
10:         terminate
```

16.3.1 The Wait-Free Case

We start with the wait-free case, namely, $f = n - 1$. For this case, there is a renaming algorithm whose output domain contains $n + f = 2n - 1$ names, namely, $M = 2n - 1$. This algorithm is simpler than the algorithm for arbitrary f, because it uses a larger name space.

The idea of the algorithm is quite simple; processors communicate using some atomic snapshot object containing for each processor its original name and a new name it suggests for itself. Each processor, p_i, starts the algorithm by writing its original name to its segment in the snapshot object. Then p_i scans the snapshot object and picks some new name that has not been suggested yet by another processor (the exact rule will be defined later). Processor p_i suggests this name by writing it to its segment in the snapshot object and scans the snapshot object again. If no other processor suggests this name, p_i decides on it; otherwise, it picks another new name and suggests it again.

The pseudocode appears in Algorithm 55. In this algorithm, the rule for picking a new name is to choose the rth ranked integer from the free (not suggested) numbers in the range $[1..2n-1]$, where r is the rank of the processor's original name among all the original names of participating processors. The algorithm uses a single atomic snapshot object, whose name is left implicit in the calls to the update and scan procedures. The ith segment of the snapshot object contains a pair of values: p_i's original name x_i and p_i's current suggestion s_i for its new name.

The following lemmas are with respect to an arbitrary admissible execution α of the algorithm. Obviously, anonymity is obeyed.

Lemma 16.14 (Uniqueness) *No two processors decide on the same name.*

Proof. Assume by way of contradiction that two processors, p_i and p_j, decide on the same name, say y. Let $(\langle x_0, s_0\rangle, \ldots, \langle x_{n-1}, s_{n-1}\rangle)$ be the view returned the last time p_i executes Line 4 before deciding. By the code, $s_i = y$, since p_i writes its suggested name before its last scan. Similarly, let $(\langle x'_0, s'_0\rangle, \ldots, \langle x'_{n-1}, s'_{n-1}\rangle)$ be the view returned the last time p_j executes Line 4 before deciding, and again, $s'_j = y$.

Without loss of generality, we may assume that p_i's scan precedes p_j's scan, by the linearizability property of the atomic snapshot object. Also, p_i does not change its suggestion after it decides, and thus $s'_i = y$. This violates the condition for deciding, and yields a contradiction. □

Note that the lemma does not depend on the specific rule used for picking the suggested name, as it does not consider Lines 6–7, where the new name is picked. This rule is important only for bounding the size of the new names and for guaranteeing termination of nonfaulty processors, as done in the next two lemmas.

The rank of a processor is at most n, and at most $n - 1$ integers are already suggested by other processors, so the highest integer a processor may suggest is $2n - 1$. Because a processor decides only on a name it has previously suggested:

Lemma 16.15 *The new names are in the range* $[1..2n - 1]$.

The delicate part of the correctness proof is arguing termination; that is, proving that a processor cannot take an infinite number of steps without deciding, regardless of the behavior of other processors.

Lemma 16.16 (Termination) *Any processor either takes a finite number of steps or decides.*

Proof. Assume, by way of contradiction, that some processor takes an infinite number of steps in the execution α without deciding; we say that such a processor is *trying*. Consider a finite prefix of α such that all trying processors have already executed Line 3 at least once and all other processors have either decided or taken all their steps. Denote by α' the remaining suffix of α; note that only trying processors take steps in α'. Let p_i be the trying processor with smallest original name; we argue that p_i decides in α', which is a contradiction.

Let *NF* (for "not free") be the set of suggested names appearing in the atomic snapshot object at the beginning of α' in the segments of processors that are not trying; note that this set remains fixed in α'. Let F (for "free") be all the remaining names, that is, $F = [1..2n - 1] - NF$; assume that $F = \{z_1, z_2, \ldots\}$, where $z_1 < z_2 < \ldots$.

Consider a point in α' where all trying processors have written a suggestion (for a new name) based on a view returned by a scan that started in α'. Since no processor performs Line 3 for the first time in α', it follows that all views contain the same set of original names; therefore, each processor gets a distinct rank.

Let r be the rank of p_i's original name x_i in this view. By choice of p_i, r is the smallest rank among all the trying processors.

We first argue that no trying processor other than p_i ever suggests a name in $\{z_1, \ldots, z_r\}$ once every trying processor has done an update based on a scan that started in α'. Consider another trying processor p_j. When p_j performs a scan in α', it sees every name in *NF* in use and possibly some other names as well. Thus the free names from p_j's perspective form a set $F' \subseteq F$. Since p_j's original name has rank greater than r, p_j suggests a name greater than z_r.

Algorithm 56 A renaming algorithm resilient to f failures: code for processor p_i, $0 \leq i \leq n-1$.

```
1:  s := ⊥
2:  repeat
3:      update_i(⟨x, s, false⟩)
4:      (⟨x_0, s_0, d_0⟩, ..., ⟨x_{n-1}, s_{n-1}, d_{n-1}⟩) := scan_i()
5:      if (s = ⊥) or (s = s_j for some j ≠ i) then
6:          let r be the rank of x in {x_j ≠ ⊥ | d_j = false, 0 ≤ j ≤ n-1}
7:          if r ≤ f + 1 then
                let s be the rth integer not in {s_j ≠ ⊥ | 0 ≤ j ≠ i ≤ n-1}
8:      else
9:          update_i(⟨x, s, true⟩)                    // indicate decided
10:         y := s                                    // decide on s as new name
11:         terminate
12: until false
```

We now argue that p_i will eventually suggest z_r in α'. If not, then p_i always sees z_r as someone else's suggestion. By definition, z_r is not a member of NF. Thus it is continually suggested by other trying processors. But by the previous claim, every other trying processor will eventually reach a point in time after which it only suggests higher names. Thus eventually p_i will stop seeing z_r as someone else's suggestion.

Thus eventually p_i will suggest z_r, see no conflicting suggestion of z_r, and decide z_r. □

16.3.2 The General Case

Let us now consider the general case of an arbitrary $f < n$; for this case, we present a renaming algorithm with $n + f$ new names. Although the wait-free algorithm will obviously work in this case as well, we pay a price in terms of an unnecessarily large name space when f is smaller than $n - 1$. Thus we are interested in more efficient algorithms for smaller numbers of failures.

The algorithm extends the idea of the previous algorithm by restricting the number of processors that are proposing names at the same time. A processor suggests a name only if its original name is among the $f + 1$ lowest names of processors that have not decided yet.

The only addition to the data in the snapshot object is a bit, where the processor announces that it has decided. The pseudocode is very similar and appears as Algorithm 56.

The uniqueness property follows by the same arguments as in Lemma 16.14.

Clearly, at most $n-1$ integers are suggested or chosen by processors other than p_i itself, in any view returned in Line 4. A processor suggests a name only if its rank is at most $f+1$; thus the name suggested by a processor is at most $(n-1)+(f+1) = n+f$.

Since a processor decides only on a name it has previously suggested, its new name must be in the range $[1..n+f]$.

The next lemma claims termination and is proved along the same lines as Lemma 16.16; the proof is left as an exercise for the reader (Exercise 16.21).

Lemma 16.17 *A processor either takes a finite number of steps or decides.*

16.3.3 Long-Lived Renaming

An interesting aspect of the renaming problem is in a *long-lived* setting, where processors request and release new names dynamically. For example, assume that we have a large potential set of processors $p_0, \ldots, p_{n-1}$, with original names $0, \ldots, n-1$; however, at each point in the execution at most k of them are interested in participating in the algorithm. There are reasons to reassign names to the participating processors, for example, to use small data structures, whose size depends on the number of participating processors, k, rather than on the total number of processors, n.

To specify the *long-lived renaming problem with k participants*, we need to use the tools from the layered model of Chapter 7; there are inputs and outputs to deal with and later in this chapter we describe an algorithm that layers two algorithms, one of which is an algorithm for long-lived renaming. The inputs for the specification are *request-name*$_i$ and *release-name*$_i$, $0 \leq i \leq n-1$; the outputs are *new-name*$_i(y)$, $0 \leq i \leq n-1$, where y is a potential name. The allowable sequences of inputs and outputs are those satisfying the following properties:

Correct Interaction: The subsequence of inputs and outputs for each i is a prefix of *request-name*$_i$, *new-name*$_i$, *release-name*$_i$, repeated forever.

To define the next properties, we need a notion of a *participating* process: p_i is participating after a prefix of a sequence of inputs and outputs if the most recent input for i that has occurred is *request-name*. We also define a participating processor p_i to be *named* y if *new-name*$_i(y)$ has occurred since the most recent *request-name*$_i$.

k-Participants: After every prefix of the sequence, the number of processors that are participating is at most k.

The uniqueness and termination properties need to be slightly reworded:

Uniqueness: If participating processor p_i is named y_i and participating processor p_j $(j \neq i)$ is named y_j after any prefix of the sequence, then $y_i \neq y_j$.

Termination: The subsequence of inputs and outputs for each i does not end with *request-name*.

The anonymity requirement from before is modified to require that the range of the new names will only depend on k, the maximum number of concurrent participants, rather than on n, the total number of potential participants.

Algorithm 57 A long-lived renaming algorithm for k participating processors: code for processor p_i.

```
1:  upon request-name event:
2:      s := ⊥
3:      repeat
4:          update_i(⟨x, s, false⟩)
5:          (⟨x_0, s_0, d_0⟩, ..., ⟨x_{n-1}, s_{n-1}, d_{n-1}⟩) := scan_i()
6:          if (s = ⊥) or (s = s_j for some j ≠ i) then
7:              let r be the rank of x in {x_j ≠ ⊥ | d_j = false, 0 ≤ j ≤ n − 1}
8:              if r ≤ k then
                let s be the rth integer not in {s_j ≠ ⊥ | 0 ≤ j ≠ i ≤ n − 1}
9:          else
10:             update_i(⟨x, s, true⟩)                    // indicate decided
11:             new-name(s) and exit
12:     until false

13: upon release-name event:
14:     update_i(⟨⊥, ⊥, ⊥⟩)
```

Bounded Name Space: For each *new-name*$_i(y)$ output, y is in $\{1, \ldots, M(k)\}$, for some function M.

Simple modifications to Algorithm 56 give an algorithm that solves long-lived renaming for k participants with new name space $2k - 1$. See Algorithm 57.

The properties of the algorithm, uniqueness, $(2k-1)$ name space, and termination are proved along the lines of the general renaming algorithm. Only the last property, termination, depends on assuming that at most k processors participate concurrently: If more than k processors participate, we are still guaranteed that new names taken concurrently by participating processors are unique and in the range $[1..2k - 1]$, but it is possible that some processor(s) will not terminate (see Exercise 16.24 to Exercise 16.26).

After translating the scan and update procedures into their constituent reads and writes, the number of steps performed by a processor between a *request-name* event and *new-name* event depends on n, the total number of processors; the chapter notes discuss algorithms whose step complexity depends on k, the number of participating processors.

16.4 K-EXCLUSION AND K-ASSIGNMENT

The section presents two problems that extend the mutual exclusion problem (studied in Chapter 4). In the first problem, *k-exclusion*, some number of processors (specified by the parameter k) are allowed to be inside the critical section concurrently. The

k-exclusion problem is a natural way to incorporate fault tolerance into the mutual exclusion problem, as will be discussed shortly.

The second problem, *k-assignment with m slots*, extends k-exclusion even further and requires each processor inside the critical section to have a unique number between 1 and m (a *slot*). The k-assignment problem is an abstraction of the situation when there is a pool of identical resources, each of which must be used by only one processor at a time; for instance, suppose a user needs to print a file but does not care which of the several available printers is used, as long as it gets exclusive access to the printer. The parameter k indicates the number of processors that can be using resources in the pool simultaneously, whereas m is the number of resources in the pool. In the absence of failures, k and m would be the same, but, as we shall see, the potential for failures indicates that m should be larger than k.

After defining the problems more precisely, we present an algorithm that solves k-exclusion in the presence of less than k failures. Then we show that k-assignment with m slots can be solved by combining a k-exclusion algorithm with a long-lived renaming algorithm for k participants whose new name space has size m.

The properties we require for k-exclusion are extensions of the properties required in the mutual exclusion problem.

k-Exclusion: No more than k processors are concurrently in the critical section.

k-Lockout avoidance: If at most $f < k$ processors are faulty, then any nonfaulty processor wishing to enter the critical section eventually does so.

Note that k-lockout avoidance (or even the weaker property of k-deadlock avoidance) cannot be attained if processors may get stuck inside the critical section. Thus, we assume that nonfaulty processors take only a finite number of steps in the critical section and eventually transfer to the exit and the remainder section. Faulty processors may fail inside the critical section, but because fewer than k processors can fail, this allows an additional nonfaulty processor to make progress and enter the critical section.

For the k-assignment problem with m slots we also require that processors in the critical section have a slot, denoted s_i for p_i, where s_i is an integer between 1 and m.

Uniqueness: If p_i and p_j are concurrently in the critical section then $s_i \neq s_j$.

As in the renaming problem, we would like to reduce the range of values held in s_i variables, that is, make m as small as possible.

16.4.1 An Algorithm for k-Exclusion

The algorithm is similar to the bakery algorithm for mutual exclusion (Algorithm 10) from Chapter 4 in that it uses tickets to order the requests by processes. (The chapter notes discuss how the tickets can be bounded.) When starting the entry section, a processor obtains a ticket, ordering itself among the processors competing for the critical section. Then it checks the tickets of the competing processors; if fewer than

Algorithm 58 A k-exclusion algorithm: code for processor p_i, $0 \leq i \leq n-1$.

```
⟨Entry⟩:
1: ticket_0, ..., ticket_{n-1} := scan_i()
2: update_i(max(ticket_j ≠ ∞ | j = 0, ..., n − 1) + 1)
3: repeat
4:     ticket_0, ..., ticket_{n-1} := scan_i()
5: until |{⟨ticket_j, j⟩ : ⟨ticket_j, j⟩ < ⟨ticket_i, i⟩}| < k      // lexicographic order
⟨Critical Section⟩
⟨Exit⟩:
6: update_i(∞)
⟨Remainder⟩
```

k of them have "older" tickets, it enters the critical section. In the exit section, a processor sets its ticket to ∞.

The algorithm uses a single atomic snapshot object, whose name is left implicit in the calls to the update and scan procedures. The ith segment of the snapshot object contains the current ticket of processor p_i. If p_i is not interested in the critical section, then its ticket is set to ∞, which is also the initial value. The pseudocode appears as Algorithm 58.

We now discuss why the algorithm provides k-exclusion and k-lockout avoidance. Fix an execution of the algorithm.

Lemma 16.18 *Algorithm 58 provides k-exclusion.*

Proof. Assume in contradiction there is a configuration in which more than k processors are in the critical section. Before entering the critical section, each of these processors executed a scan, stored a new ticket, and scanned again (at least once).

Let p_i be the processor who stored its ticket latest in the execution. The linearizability property of snapshots implies that the scan of p_i contains the tickets of all other processors. That is, p_i must see at least k smaller tickets, which contradicts the condition on Line 5. □

The proof of k-lockout avoidance considers some processor p_i that gets stuck in the entry section. Eventually, all nonfaulty processors that enter the critical section before p_i exit; in addition, all nonfaulty processors that scanned before the update of p_i (in the linearizability order of the snapshot) enter the critical section as well and exit. In both cases, if these processors ever move to the entry section, they will get a ticket larger than p_i's ticket. Thus eventually the only processors with tickets smaller than p_i's are the faulty ones; because fewer than k processors are faulty, the condition in Line 5 will be satisfied and p_i will enter the critical section. This proves:

Lemma 16.19 *Algorithm 58 provides k-lockout avoidance.*

Together, the last two lemmas prove:

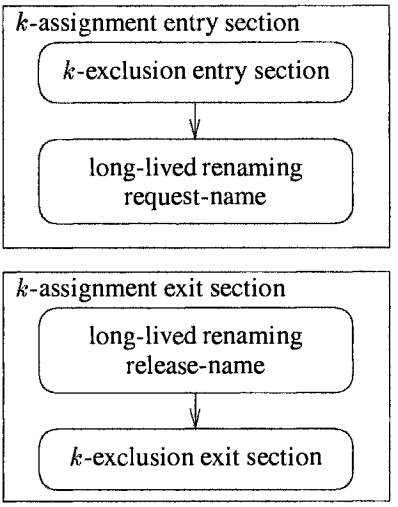

Fig. 16.6 A schematic view of the k-assignment algorithm.

Theorem 16.20 *Algorithm 58 solves the k-exclusion problem, if fewer than k processors are faulty.*

16.4.2 An Algorithm for k-Assignment

The reader might be tempted to suggest solving k-assignment with $2k-1$ slots using the algorithm for long-lived renaming for k participants. The slot assignments are obvious because the renaming algorithm uses $2k-1$ names, and indeed, as discussed at the end of Section 16.3.3, if too many processors wish to enter the critical section, some of them will be stuck. Thus the protocol provides k-exclusion as well. It is the lockout avoidance property that is not provided by long-lived renaming; when more than k processors are trying to obtain a new name, a processor in the entry section may be overtaken infinitely many times (see Exercise 16.27).

The solution, however, is simple; we encompass a long-lived renaming algorithm for k participants (using $2k-1$ names) with a k-exclusion algorithm (see Fig. 16.6). Formally proving the properties of the algorithm is left to the reader.

Exercises

16.1 Prove that in a solo execution of any k-set consensus algorithm, the processor must decide on its input value.

16.2 For both cases considered in the proof of Lemma 16.3, prove that p_j distinguishes between α and α'.

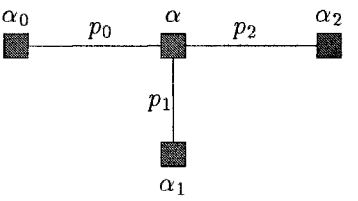

Fig. 16.7 Illustration for Exercise 16.6.

16.3 Consider the following block execution for four processors:

$$\alpha = \{p_0, p_2, p_3\}, \{p_2, p_3\}, \{p_0\}, \{p_1\}, \{p_1\}$$

1. Which processor is unseen in α?
2. Is p_0 seen in block 1 of α?
3. Is p_3 last seen in block 3 of α?
4. Construct the unique execution $\alpha' \overset{\neg p_i}{\sim} \alpha$, for every $i = 0, 1, 2, 3$.

16.4 Find a formula for the number of block executions with two processors, assuming that each processor always takes exactly s steps before deciding.

16.5 Prove that for a system with n processors there is a finite number of block executions.

16.6 Prove that the restricted degree of a non-imaginary node in $\hat{\mathcal{B}}_3$ cannot be 3.

Hint: Consider a non-imaginary node and its adjacent nodes, as described in Figure 16.7, and try to determine the decisions of all three processors in the four block executions corresponding to these nodes.

16.7 Extend the proof of Exercise 16.6 to show that in $\hat{\mathcal{B}}_k$ there is no non-imaginary node whose restricted degree is 3.

16.8 Let A be a k-set consensus algorithm for three processors. Show a one-to-one correspondence between the block executions of A in which p_2 is unseen and the block executions of a k-set consensus algorithm A' for two processors.

Hint: A' behaves like A behaves in block executions that do not contain any steps of p_2.

16.9 Explain how the definition of block executions should be modified so that the proof of Theorem 16.7 holds also when processors communicate by ordinary read and write operations, rather than update and scan operations.

16.10 Prove, by induction on $k \geq 2$, that an odd number of nodes in $\mathcal{B}_k$ have restricted degree 1.

Hint: Add an imaginary node connected to all nodes corresponding to block executions with an unseen processor; use the inductive hypothesis on k to prove that the restricted degree of the imaginary node is odd; complete the proof as was done for $k = 3$.

16.11 Extend Lemma 16.6 to any $k \geq 2$. That is, prove that if the restricted degree of a non-imaginary node in $\mathcal{B}_k$ is 1, then it corresponds to an execution in which $\{0, \ldots, k-1\}$ are decided.

16.12 Extend the simulation of Section 5.3.2 to prove: If there is a k-set consensus algorithm for a system of $n > k$ processors that tolerates the failure of k processors, then there is a wait-free k-set consensus algorithm for a system of k processors.

16.13 Modify the code of the approximate agreement algorithm to use only a single snapshot object, with possibly unbounded number of values in each segment.

16.14 Prove Theorem 16.12.

16.15 Prove that in a solo execution of any approximate agreement algorithm, the processor must decide on its input value.

16.16 Complete the proof of Lemma 16.8.

16.17 Use Lemma 16.8 to prove Lemma 16.9 and Lemma 16.10.

16.18 Prove that Algorithm 53 is correct even if processors use only an array of atomic single-writer multi-reader registers.

16.19 Modify the approximate agreement algorithm (Algorithm 54) and its correctness proof so that *maxRound* is calculated only in the first asynchronous round.

16.20 Present an execution of the renaming algorithm (Algorithm 55) in which some process takes an exponential number of steps before deciding.

16.21 Prove Lemma 16.17.

16.22 Describe an execution of Algorithm 56 that uses name $n + f$.

16.23 Prove that there is no wait-free renaming algorithm with new name space of size n.

Hint: Follow the ideas used to prove that there is no wait-free algorithm for solving consensus (Theorem 5.18).

16.24 Prove the uniqueness property of Algorithm 57, even when more than k processors participate concurrently.

16.25 Prove that $[1..2k+1]$ is the new name space used by Algorithm 57, even when more than k processors participate concurrently.

16.26 Prove the termination property of Algorithm 57, when k processors or less participate concurrently.

16.27 Show an execution of Algorithm 57 with more than k participating processors (concurrently), in which some processor does not terminate.

16.28 Show an execution of Algorithm 57 with $2k$ participating processors (concurrently), in which all processors terminate.

16.29 Explain why Algorithm 58 needs a flag.

Hint: Assume that many processors (more than k) exit the critical section and are now in the remainder; now consider what happens when some processor wishes to enter the critical section.

16.30 Specify the k-exclusion and k-assignment problems using the model of Chapter 7. Then describe and prove correct the k-assignment algorithm mentioned in Section 16.4.2 using the layered model.

Chapter Notes

The study of solvable problems has been a prolific research area in the last five years, and our presentation has only touched it; here, we try to describe some of the key developments.

The k-set consensus problem was first presented by Chaudhuri [73], who also presented an f-resilient algorithm for k-set consensus, for any $f < k$. The lower bound showing that k-set consensus cannot be solved in the presence of $f \geq k$ failures was proved concurrently by Borowsky and Gafni [58], by Herlihy and Shavit [132], and by Saks and Zaharoglou [233].

Our proof of the lower bound for k-set consensus combines an operational argument about block executions and their similarity (and non-similarity) structure with a combinatorial argument about a coloring of a graph representing this structure. Block executions were defined by Saks and Zaharoglou [233] and by Borowsky and Gafni [58], who called them *immediate snapshot executions* because they correspond to executions in which restricted snapshot objects are employed. Lemma 16.2 and Lemma 16.3 were stated and proved by Attiya and Rajsbaum [32]. The graph representation of the similarity structure of block executions ($\mathcal{B}_n$) is new; it is inspired, in part, by the lower bound on the number of rounds for solving k-set consensus in the synchronous model, presented by Chaudhuri, Herlihy, Lynch, and Tuttle [75]. The combinatorial argument follows a graph-theoretic proof of Sperner's lemma given by Tompkins [253]; see also Bondy and Murty's standard text [57, pp. 21–23].

Approximate agreement was first presented by Dolev, Lynch, Pinter, Stark, and Weihl [96]; they considered message-passing models with crash and Byzantine failures. The algorithms we presented are based on an algorithm of Moran [190].

The renaming problem was first presented by Attiya, Bar-Noy, Dolev, Peleg, and Reischuk [26] for message-passing systems. The algorithms we presented here

(Algorithm 55 for the wait-free case and Algorithm 56 for the general case) are adaptations of the algorithm of Attiya et al. to shared memory systems.

Moir and Anderson [188] presented algorithms for long-lived renaming that are *fast* in the sense discussed for mutual exclusion, that is, in the absence of contention, a processor will pick a new name in a constant number of steps. Better algorithms for long-lived renaming were presented by Moir and Garay [189].

The k-exclusion problem was introduced by Fischer, Lynch, Burns, and Borodin [108]; we have presented an algorithm of Afek, Dolev, Gafni, Merritt, and Shavit [5], who also showed that the number of times a processor is overtaken is bounded in this algorithm. As for the algorithms of Chapter 10, a bounded timestamp system can be used instead of explicit tickets, to bound the memory requirements of this algorithm.

The k-assignment problem was introduced in message-passing systems under the name *slotted k-exclusion* by Attiya, Bar-Noy, Dolev, Peleg, and Reischuk [26]; the term *k-assignment* was coined by Burns and Peterson [64], who were the first to study this problem in shared memory systems. Burns and Peterson also showed that at least $2k + 1$ "slots" are required to solve this problem, by a proof that extends the methods of Fischer, Lynch, and Paterson [110].

All algorithms for variants of the renaming problem require the number of new names to be at least $n + f$, where f is the number of failures to be tolerated. An obvious question is whether the number of new names can be reduced. Attiya, Bar-Noy, Dolev, Peleg, and Reischuk [26] showed that at least $n + 1$ new names are needed (Exercise 16.23). Much later, Herlihy and Shavit proved that the number of new names must be at least $n + f$ [132].

Herlihy and Shavit derive the lower bounds for k-set consensus and renaming from a more general theorem characterizing the problems that can be solved by an f-resilient algorithm with only read and write operations [133]. This theorem uses techniques of algebraic topology. A simpler characterization theorem, relying only on graph-theoretic concepts, was proved by Biran, Moran, and Zaks [52] when only a single processor may fail. Their theorem is stated for message-passing systems, but simulations such as those presented in Chapter 5 and Chapter 10 can be used to translate it to shared memory systems.

Distributed Computing: Fundamentals, Simulations and Advanced Topics, Second Edition
Hagit Attiya and Jennifer Welch
 ISBN: 0-471-45324-2

17

Solving Consensus in Eventually Stable Systems

We have seen that fault-tolerant consensus is impossible to solve in an asynchronous system in which processors communicate via message passing or shared read-write registers. The key difficulty in trying to tolerate failures in an asynchronous system is distinguishing between a crashed processor and a slow processor. However, the assumption of complete asynchrony is often an overly pessimistic view of practical systems. If there are upper bounds on processor step time and message delays, synchrony can be used to detect failed processors, for instance, by having the processors exchange 'I'm alive' messages periodically.

A more abstract approach to detecting failures is to assume a service that does so but whose inner workings are not known to the users. This *failure detector* service could then be used in any kind of system, including an asynchronous one. The motivation for this approach is that there could be some other, and better, way to detect failures rather than imposing more stringent timing assumptions on the systems. This approach ignores the *operational features* of a failure detector and concentrates on the *properties* needed to solve consensus.

Strengthening the system assumptions, for instance, with failure detectors or stronger synchrony, is one strategy for circumventing the impossibility of consensus presented in Chapter 5. Recall from Chapter 5 that guaranteeing both safety (agreement and validity) and termination is impossible when the system is asynchronous.

A system may be poorly behaved for arbitrarily long periods, yet safety should nonetheless be preserved during these periods; that is, processors should never decide on conflicting or invalid values. We describe a simple mechanism to guarantee safety requirements, whereas termination is achieved only when the environment is well-behaved. Later, we describe how failure detectors encapsulate the treatment of the

environment's behavior. A similar approach is taken in Chapter 14, where termination relied on lucky rolling of a dice.

The chapter starts with the mechanism for guaranteeing safety, which may terminate under fortunate circumstances. We present a formal model for failure detectors and define three types of failure detectors and show how they can be combined with the safety preserving algorithm to solve consensus, in both shared memory and message-passing systems. Possible implementations of failure detectors are discussed, as well as an application of these algorithms to state-machine replication.

17.1 PRESERVING SAFETY IN SHARED MEMORY SYSTEMS

This section presents a basic algorithm for guaranteeing safety; the algorithm is presented for asynchronous shared memory systems.

The algorithm consists of many invocations of a procedure called safe-phase. Each invocation has an associated *phase number* as a parameter. A processor may have several (non-overlapping) invocations of safe-phase; the numbers passed as parameters to the invocations of a particular processor are strictly increasing over the duration of the execution. Different processors may execute phases concurrently, but it is assumed that a *separate* set of phase numbers is used by each processor, for example, by appending the processor id as the "lower" bits of the phase number.

Calls to safe-phase also take a *value* parameter and return a value subject to the following conditions:

Validity: If an invocation of safe-phase returns $v \neq \perp$, then v is the *value* parameter in some invocation of safe-phase that begins before the return of v

Agreement: If an invocation returns $v \neq \perp$, then no invocation returns a value other than v or $\perp$.

Conditional termination: If there is an invocation of safe-phase with phase number r such that every other invocation that begins before this one ends has a smaller phase number, then the invocation returns a value $v \neq \perp$.

Each processor maintains its current *suggestion* regarding a return value in a shared register. Each phase consists of two stages: In the first stage, the processor chooses a value v, and in the second stage it tries to decide on v as its return value. Specifically, in the first stage, a processor writes its new phase number, and reads all the registers. If some other processor is observed to have reached a larger phase, the processor ends the phase without choosing. Otherwise, the processor chooses the observed value with the largest phase number and writes this value, tagged with its own current phase number, as its suggestion to its register. If no value has yet been suggested, then the processor suggests the value that was its input parameter. In the second stage, the processor reads all the registers; if all processors are still in a smaller phase, the processor decides on its chosen value.

In more detail, processor p_i has a single-writer multi-reader register R_i with the following fields:

Algorithm 59 safe-phase procedure for processor p_i.

```
procedure safe-phase(value x, integer r)
                                    // Stage 1: choose that value with largest phase tag
1:  R_i.phase := r          // other fields of R_i are written with their current values
2:  maxPhase := 0
3:  chosenVal := x
4:  for j := 0 to n − 1 do
5:      if R_j.phase-tag > r then return ⊥
6:      if R_j.val ≠ ⊥ then
7:          if R_j.phase-tag > maxPhase then
8:              maxPhase := R_j.phase-tag
9:              chosenVal := R_j.val
10: R_i := ⟨r,chosenVal,r⟩
                          // Stage 2: check that no other processor started a larger phase
11: for j := 0 to n − 1 do
12:     if R_j.phase > r then return ⊥
13: return chosenVal
```

phase The current phase number, initially 0.

val The value that p tried to commit in its last phase, initially $\perp$.

phase-tag The phase in which *val* was written.

Algorithm 59 presents the pseudocode.

Note that *chosenVal*, initially set to the argument x, is overwritten unless every other processor has a smaller phase and a $\perp$ value in its shared variable.

We first prove that the algorithm satisfies the validity property. The decision value is either processor p_i's input or a value read from some other processor's register. Simple induction shows this register contains only input values, which implies the validity of the algorithm.

Next, we prove the agreement property. Let p_i be the processor that decides with the smallest phase number, r_i, on some value v. This processor is well-defined because processors use distinct phase numbers.

We first argue that all processors that write a suggestion for a later phase (greater than r_i) suggest v. Otherwise, let p_j be the processor that first writes (in Line 10) a conflicting suggestion $v' \neq v$ for a phase $r_j > r_i$. (Again, this processor is well-defined because processors use distinct phase numbers and writes are atomic.) Note that p_i's second read from $R_j.phase$ (in Line 12) during phase r_i does not see r_j or a larger phase number, otherwise p_i would return $\perp$, and not v, from safe-phase. The invocations of safe-phase by p_i have increasing phase numbers and thus, p_j's first write to $R_j.phase$ (in Line 1) for phase r_j follows p_i's write of v to $R_i.val$ (in Line 10) for phase r_i. Hence p_j reads v from $R_i.val$ with phase number r_i during phase r_j. Because p_j is the earliest processor to write a conflicting value with phase number larger than r_i, p_j only sees the value v associated with phase numbers that

are at least r_i during the for loop in Lines 11–12. Thus p_j suggests v, not v', for phase r_j.

Because processors decide only on values they suggest, no processor decides on a value different than v.

Finally, we verify conditional termination. Inspecting the code reveals that if a nonfaulty processor p_i executes an entire phase with the largest phase number, then p_i decides.

17.2 FAILURE DETECTORS

A way to capture stability properties of an asynchronous system is with the concept of a *failure detector*. When combined with the safety-preserving mechanism in Algorithm 59, failure detectors allow termination to be achieved. This section augments the formal model of computation given in Chapters 2 and 5 to model failure detectors.

Every processor p_i has a failure detector component called $suspect_i$ that contains a set of processor ids. The state transition function of a processor uses the current value of $suspect_i$ as one of its arguments (in addition to the current accessible state of p_i).

The state transition function does not change $suspect_i$. Instead, $suspect_i$ is updated by a *failure detector algorithm*, which we are not explicitly modeling. In each configuration of the system, the value of each $suspect_i$ component contains a set of processor ids. If p_j is in $suspect_i$, then we say that p_i *suspects* p_j (of being faulty). Below we put constraints on the values of the *suspect* variables in admissible executions to reflect the workings of a specific kind of failure detector.

What do we want a failure detector to do? First, it should tell us that a failed processor has failed. Such a condition is called "completeness." In particular, we shall insist that *eventually* every processor that crashes is permanently suspected by every nonfaulty processor. To rule out unhelpful failure detectors that would simply suspect everyone, we also would like a failure detector not to tell us that an operational processor has failed. Such a condition is called "accuracy." We first consider a fairly weak form of accuracy, in which eventually *some* nonfaulty processor is never suspected by any nonfaulty processor. Note that the failure detector is allowed to make some mistakes for a while before settling down to good behavior. Incorporating the above discussion into the formal model, we get the following definition.

Definition 17.1 *A failure detector is* eventually strong, *denoted* $\diamond\mathcal{S}$, *if every admissible execution satisfies the following two properties:*

- *For every nonfaulty processor p_i and faulty processor p_j, there is a suffix of the execution in which j is in $suspect_i$ in every configuration of the suffix.*
- *There exists a nonfaulty processor p_i and a suffix of the execution such that for every nonfaulty processor p_j, i is not in $suspect_j$ in any configuration of the suffix.*

The accuracy property can be strengthened so that there is some nonfaulty processor that is *never* suspected.

Definition 17.2 *A failure detector is* strong, *denoted $\mathcal{S}$, if every admissible execution satisfies the following two properties:*

- *For every nonfaulty processor p_i and faulty processor p_j, there is a suffix of the execution in which j is in $suspect_i$ in every configuration of the suffix.*
- *There exists a nonfaulty processor p_i such that for every nonfaulty processor p_j, i is not in $suspect_j$ in any configuration of the execution.*

A complementary approach equips every processor p_i with a component called $trust_i$ that contains a single processor id, instead of $suspect_i$. The indicated processor can be considered a leader. In a similar manner, the state transition function of a processor uses the current value of $trust_i$ as one of its arguments; $trust_i$ is updated by a *failure detection algorithm*, which we are not explicitly modeling; this algorithm can also be viewed as a leader election algorithm. If $trust_i$ is p_j, then we say that p_i *trusts* p_j (as being nonfaulty).

Analogously to the $\diamond\mathcal{S}$ failure detector, we allow this failure detector, called Ω, to be initially unreliable; it can fail either by electing a faulty processor or by causing different processors to trust different leaders. More formally:

Definition 17.3 *A failure detector is a* leader elector, *denoted Ω, if every admissible execution satisfies the following property:*

- *Eventually $trust_i$ at every nonfaulty processor p_i holds the id of the same nonfaulty processor.*

17.3 SOLVING CONSENSUS USING FAILURE DETECTORS

17.3.1 Solving Consensus with $\diamond\mathcal{S}$

A "rotating coordinator" paradigm is employed to solve consensus with $\diamond\mathcal{S}$. The *coordinator* of phase r is the processor p_c, where $c = r \bmod n$, and it calls safe-phase with phase number r. A processor that is not the coordinator of phase r simply waits until either the coordinator completes phase r, or it suspects the coordinator. Then the processor increases its phase number and repeats.

The pseudocode appears in Algorithm 60.

We now show that, in any admissible execution of Algorithm 60, the hypothesis for conditional termination of safe-phase holds, that is, eventually there will be a processor that executes an entire invocation of safe-phase with the largest phase number. Consider a point in the execution when all the faulty processors have crashed and henceforth some nonfaulty processor p_c is never suspected. Let $r_{\max}$ be the maximum phase (according to $R_i.phase$) of any nonfaulty processor at that point. Let r be the smallest multiple of c that is larger than $r_{\max}$. Clearly, no nonfaulty

Algorithm 60 Consensus with $\diamond\mathcal{S}$: code for processor p_i.

```
1:  r := 0
2:  while true
3:      c := r mod n
4:      if i = c then
5:          ans := safe-phase(r,x)                    // x is p_i's input
6:          if ans ≠ ⊥ then y := ans                  // and halt, y is p_i's output
7:      else wait until c ∈ suspect or R_c.phase ≥ r   // not a coordinator
8:      r := r + 1                                    // and repeat
```

processor gets stuck at any phase number unless it decides. Thus eventually p_c executes safe-phase with phase number r and is the coordinator of that phase.

Suppose, in contradiction, that there is another processor p_i that executes safe-phase with a phase number $r' \geq r$ concurrently with p_c's execution of safe-phase with phase number r. Because different processors use different phase numbers, $r' \neq r$, and thus $r' > r$. Because processors do not skip phases, p_i executed phase r at some earlier time. How did p_i finish phase r? Obviously p_c had not finished phase r yet, so p_i must have suspected p_c. But p_i executes phase r after the time when p_c is no longer suspected, because $r > r_{\max}$, which is a contradiction.

As stated, the conditional termination property ensures only that processor p_c terminates. It is possible to make all processors terminate in this case, by having a processor write its decision to a shared register once it has decided. A processor begins each iteration of the while loop by checking the decision registers of the other processors; if anyone else has decided, the processor decides the same value. This modification ensures that after p_c decides, eventually every nonfaulty processor decides as well.

Theorem 17.1 *Consensus can be solved in shared memory systems using $\diamond\mathcal{S}$, for any number of crash failures.*

In contrast to the wait-freedom possible in shared memory, $n > 2f$ is necessary for solving consensus with $\diamond\mathcal{S}$ when processors communicate by message passing. This result is shown using a familiar partitioning argument (cf. Theorem 10.22 in Chapter 10).

Theorem 17.2 *Consensus cannot be solved using the $\diamond\mathcal{S}$ failure detector in an asynchronous system if $n \leq n/2$.*

Proof. Suppose in contradiction that there is an algorithm A that solves consensus using the $\diamond\mathcal{S}$ failure detector in an asynchronous system with $n \leq 2f$. Partition the set of processors into two sets S_0 and S_1 with $|S_0| = \lceil n/2 \rceil$ and $|S_1| = \lfloor n/2 \rfloor$.

Consider an admissible execution α_0 of A in which all inputs are 0, all processors in S_0 are nonfaulty, and all processors in S_1 crash initially. Furthermore, suppose the failure detector behavior is such that every processor in S_0 permanently suspects

every processor in S_1 and never suspects any processor in S_0. By the termination and validity conditions of A, some processor p_i in S_0 decides 0 at some time t_0.

Consider an analogous admissible execution α_1 of A in which all inputs are 1, all processors in S_1 are nonfaulty, and all processors in S_0 crash initially. Suppose the failure detector behavior is such that every processor in S_1 permanently suspects every processor in S_0 and never suspects any processor in S_1. Again, by the termination and validity conditions of A, some processor p_j in S_1 decides 1 at some time t_1.

Finally, consider an admissible execution α_2 of A that is a "merger" of α_0 and α_1. In more detail, all processors in S_0 have input 0, all processors in S_1 have input 1, and there are no faulty processors, but messages between S_0 and S_1 are delayed until time $t_2 = \max\{t_0, t_1\}$. Suppose the $\diamond\mathcal{S}$ failure detector behaves as follows: Every processor in S_0 suspects every processor in S_1 until time t_2, and then it suspects no one. Every processor in S_1 suspects every processor in S_0 until time t_2, and then it suspects no one.

Executions α_0 and α_2 are indistinguishable to p_i until time t_2, so p_i decides 0 at time t_0 in α_2. But executions α_1 and α_2 are indistinguishable to p_j until time t_2, so p_j decides 1 at time t_1 in α_2. Thus α_2 violates the agreement condition for consensus. □

The simulation of shared memory in a message-passing system (Section 10.4) can be applied to the algorithm of Theorem 17.1 to obtain Theorem 17.3.

Theorem 17.3 *Consensus can be solved in message-passing systems using $\diamond\mathcal{S}$, assuming that $n > 2f$.*

17.3.2 Solving Consensus with $\mathcal{S}$

In shared memory systems, the same algorithm used with $\diamond\mathcal{S}$ (Algorithm 60) can be used to solve consensus with $\mathcal{S}$. Because $\mathcal{S}$ guarantees that some nonfaulty processor p_j is never suspected by nonfaulty processors, the algorithm terminates when p_j takes on the role of coordinator. That is, the algorithm terminates within n phases.

When message-passing systems are considered, Theorem 17.2 cannot be extended to show that $n > 2f$ is required for solving consensus with $\mathcal{S}$. In fact, consensus can be solved for any value of n and f when $\mathcal{S}$ can be employed.

One way to derive this algorithm (Exercise 17.3 explores another way) is to use $\mathcal{S}$ in order to simulate shared memory on top of message passing, without requiring that $n > 2f$.

Specifically, we modify the simulation of shared memory in a message-passing system (Section 10.4) so that processor p_i waits for responses from all processors not in $suspect_i$.

In more detail, when the writer wants to write a value to a register, it sends a message containing the new value and an incremented sequence number to all the processors. Each recipient updates a local variable with the value, if the sequence number is larger than what it currently has; in any event, the recipient sends back an acknowledgment. Once the writer receives acknowledgments from all processors it

Algorithm 61 Consensus with Ω: code for processor p_i.

```
1:  r := 0
2:  while true do
3:      if i = trust then
4:          ans := safe-phase(r + i, x)          // x is p_i's input
5:          if ans ≠ ⊥ then y := ans             // and halt, y is p_i's output
6:      r := r + n                               // and repeat
```

does not suspect, it finishes the write. The properties of $\mathcal{S}$ guarantee that the writer eventually receives responses from all processors it does not suspect.

In a similar manner, in order to read the register, a reader sends a message to all the processors. Each recipient sends back a message with the value it currently has. Once the reader receives a responses from all processors it does not suspect, it returns the value with the largest sequence number among those received.

Failure detector $\mathcal{S}$ guarantees that there is some nonfaulty processor, say p_j, that is never suspected by any processor. Clearly, p_j is in the intersection of the set of processors sending an acknowledgment to the writer and the set of processors sending a response value to the reader. Exercise 17.4 asks you to show that the algorithm correctly simulates a shared register, following the proof of Theorem 10.21.

17.3.3 Solving Consensus with Ω

An alternative approach to solving consensus relies on the failure detector Ω. Algorithm 59 is executed together with failure detector Ω, which provides termination. Different processors use different phase numbers, in particular, processor p_i uses phase numbers $i, n+i, 2n+i, 3n+i, \ldots$. As long as processor p_i is a leader and has not yet decided, p_i calls safe-phase with increasing phase numbers.

The pseudocode appears in Algorithm 61.

To see why Algorithm 61 is correct, consider any admissible execution. Eventually, at some point in the execution, Ω ensures that every processor continuously trusts the same nonfaulty processor, call it p_c. All invocations of safe-phase that are in progress at that time are completed by some later time, after which no processor other than p_c invokes safe-phase any more. Processor p_c, however, continues to invoke safe-phase and thus eventually does so with a phase number that is larger than the phase number of any other invocation. This invocation satisfies the hypotheses for the conditional termination condition of safe-phase and thus it returns a non-$\perp$ value, causing p_c to decide. Termination of other processors can be handled in the same manner as for Algorithm 60.

For message-passing systems, it can be shown (Exercise 17.5) that $n > 2f$ is required for solving consensus, even when the system is augmented with the Ω failure detector. Under the assumption that $n > 2f$, Algorithm 61 can be executed in a message-passing system, using the simulation of Section 10.4.

17.4 IMPLEMENTING FAILURE DETECTORS

Because $\diamond\mathcal{S}$ can be used to solve consensus, it is not possible to implement $\diamond\mathcal{S}$ in an asynchronous system subject to crash failures. If it were, then consensus could be solved, contradicting the impossibility of doing so shown in Chapter 5. Obviously, the same observation holds for any failure detector that can be used to solve consensus, for example, $\mathcal{S}$ and Ω.

However, this observation is a theoretical one, which shows that for any proposed algorithm, there is a particularly adversarial execution in which it will fail. More practically speaking, there is a simple implementation of $\diamond\mathcal{S}$ based on timeouts. Each processor periodically sends an 'I'm alive' message to all the other processors. If a processor p_i does not hear from another processor p_j for some length of time (called the timeout interval for p_j), then p_i puts p_j in its suspect list. If p_i hears from p_j while p_i suspects p_j, then p_i removes p_j from its suspect list and increases the timeout interval for p_j.

Clearly this scheme satisfies the completeness property of $\diamond\mathcal{S}$: Because a processor that has failed never sends any more 'I'm alive' messages, this processor will eventually be put in every nonfaulty processor's suspect list and never removed. What about the accuracy property of $\diamond\mathcal{S}$? It is possible that a nonfaulty processor p_j will continually be added to and removed from the suspect list of another nonfaulty processor. This behavior, which violates the accuracy property, occurs if the messages from p_j to p_i always have delay longer than the current timeout interval for p_j being maintained by p_i. However, such a pattern of message delays is highly unlikely to occur.

A similar scheme implements $\mathcal{S}$ if some nonfaulty process *never* violates the timing assumptions.

Ω can be built on top of another failure detector, in particular, $\diamond\mathcal{S}$, or implemented directly using timing assumptions.

17.5 STATE MACHINE REPLICATION WITH FAILURE DETECTORS

Recall the state machine approach for implementing distributed systems, described in Section 8.4.2. In this approach, a system is described as a state machine, whose transitions are initiated by client requests and return responses. If the state machine is deterministic, then the key issue is *ordering* (or *sequencing*) the clients' requests.

A simple way to order the requests uses a single coordinator (server), who receives requests from clients, applies them to the state machine, computes responses, and sends them back to the clients. Obviously, in this scheme, the server is a single point of failure. Fault tolerance can be improved by replacing the single server with a collection of servers.

One approach to state machine replication has one of the servers act as the coordinator in normal execution mode; when the coordinator fails, one of the remaining servers is elected to replace it as the new coordinator. When the system is asynchronous, leader election must rely on timing assumptions or other mechanisms for

detecting failures; the leader election mechanism may fail to converge, when the system does not obey the timing assumptions. This behavior can be encapsulated within failure detector Ω.

Ω may produce erroneous results for a while, causing several servers to consider themselves coordinators until Ω stabilizes and elects a single leader. If each (self-proclaimed) coordinator orders clients' request on its own, different processors' views of the state machine transitions will diverge and become inconsistent. Instead, agreement among coordinators must be used to order clients' requests. Servers that consider themselves coordinators invoke a copy of Algorithm 61 for each state machine transition; their input is the next client request they wish to commit. The coordinators invoke the algorithm for transition $\ell + 1$ only after transitions $1, \ldots, \ell$ are decided, and the state of the machine after the first ℓ transitions is fixed.

Note that even when the system is stable, with a unique nonfaulty coordinating server, the coordinator still calls Algorithm 61 for every transition of the state machine. The reason is that other processors may erroneously suspect the coordinator is faulty and try to replace it.

Exercises

17.1 Modify the proof of Theorem 17.2 to show that nonfaulty processors must continue to send messages in order to solve consensus with $\diamond\mathcal{S}$.

17.2 Show optimizations to the message and time complexity of the simulation of shared memory by message passing in the context of Theorem 17.3.

17.3 Directly derive a consensus algorithm for message-passing systems, with any number of faulty processors, using $\mathcal{S}$.

Hint: Follow Algorithm 15.

17.4 Expand the ideas presented in Section 17.3.2 to show a simulation of a shared register in a message-passing system, with any number of failures, assuming failure detector $\mathcal{S}$.

Hint: Follow Theorem 10.21.

17.5 Modify the proof of Theorem 17.2, to show that consensus cannot be solved using the Ω failure detector in an asynchronous system if $n \leq 2f$.

17.6 Suppose that the completeness property of the $\diamond\mathcal{S}$ failure detector is weakened to require that eventually every crashed processor is suspected by *some* nonfaulty processor (instead of every one). Either show how to convert this weaker failure detector into $\diamond\mathcal{S}$ in an asynchronous system with $n > 2f$, or prove that it is impossible.

Can this weaker failure detector be used to solve consensus in an asynchronous system?

17.7 Suppose that the accuracy property of the $\diamond\mathcal{S}$ failure detector is strengthened to require that eventually *no* nonfaulty processor is suspected by any nonfaulty processor. Either show that consensus cannot be solved in an asynchronous system with this stronger failure detector when $n \leq 2f$ or describe an algorithm using this failure detector that works when $n \leq 2f$.

17.8 $\diamond\mathcal{P}$ is a failure detector that guarantees that eventually each processor's suspected list contains exactly the faulty processors. Can you use $\diamond\mathcal{P}$ to simulate Ω?
Can you use Ω to simulate $\diamond\mathcal{P}$?
Either give algorithms or give impossibility proofs. Assume a message passing system with crash failures.

Chapter Notes

The original paper introducing failure detectors was by Chandra and Toueg [67], who proposed a variety of completeness and accuracy conditions. They also presented consensus algorithms for message-passing systems, using various failure detectors. Lo and Hadzilacos [171] studied failure detectors in shared-memory systems, and presented consensus algorithms using $\diamond\mathcal{S}$ and $\mathcal{S}$.

Our presentation is inspired by the work of Delporte-Gallet, Fauconnier, and Guerraoui [89], who studied shared memory simulations using failure detectors. (The solution to Exercise 17.4 can be found in this paper.) Algorithm 59 is based on algorithms presented by Lo, and Hadzilacos [171] and by Gafni and Lamport [116].

The failure detector in Exercise 17.6 is $\diamond\mathcal{W}$; Chandra, Hadzilacos, and Toueg [66] showed that no weaker failure detector can solve consensus.

In the crash failure model, a failed processor is indistinguishable from a slow processor. In contrast, in the *failstop model*, described in Chapter 8, it is possible to tell whether a processor has failed. Sabel and Marzullo [230] use failure detector $\diamond\mathcal{W}$ to simulate failstop processors in the presence of crash failures.

Failure detectors have been applied to several other problems, including various kinds of broadcasts [67], atomic commitment [126], leader election [231], and group membership [41, 79, 114, 169]. Additional work has addressed the relationships between different failure detector specifications [67, 83], and failure detectors in shared memory [171, 197].

Algorithm 61 is the shared memory version of the *Paxos* algorithm for message-passing systems, presented by Lamport [161], originally in 1989. Our description is inspired by the so-called *Disk Paxos* algorithm of Gafni and Lamport [116]. De Prisco, Lampson, and Lynch [220] describe an alternative way to derive Algorithm 61, by embedding the leader election into the algorithm. Their algorithm uses processor ids to break ties when there are several conflicting proposals for the same phase and terminates when certain timing assumptions hold.

In the state machine replication scheme described in Section 17.5, Algorithm 61 is invoked for each transition of the state machine, because it is not clear whether

the system is stable or a new coordinator is being elected. Lamport [161] optimizes the normal case, where the system is stable and there is a single coordinator. The coordinator performs the first stage of safe-phase for several transitions at once. Thus, ideally, a leader will be able to commit several waiting transitions fairly quickly.

It is worth comparing the Paxos approach to state machine replication, discussed in this chapter, with state machine replication using totally order broadcast, described in Chapter 8. The latter approach is more flexible because applications can trade off weaker semantics of the broadcast service for better performance; this allows the development of applications in an incremental manner, first prototyping using strong semantics, then gradually weakening the semantics provided by the broadcast service in order to improve performance, while preserving correctness. On the other hand, the safety mechanism embedded in the Paxos approach, when translated to message passing, requires a processor to communicate only with a majority of the processors, whereas broadcast-based replication requires all group members to acknowledge each operation.

Distributed Computing: Fundamentals, Simulations and Advanced Topics, Second Edition
Hagit Attiya and Jennifer Welch
 ISBN: 0-471-45324-2

References

1. Karl Abrahamson. On achieving consensus using a shared memory. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing*, pages 291–302. ACM, 1988.

2. Sarita Adve and Mark Hill. Weak ordering—A new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, 1990.

3. Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873–890, September 1993.

4. Yehuda Afek, Geoffrey Brown, and Michael Merritt. Lazy caching. *ACM Transactions on Programming Languages and Systems*, 15(1):182–205, January 1993.

5. Yehuda Afek, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. A bounded first-in, first-enabled solution to the l-exclusion problem. *ACM Transactions on Programming Languages and Systems*, 16(3):939–953, May 1994.

6. Yehuda Afek and Yossi Matias. Elections in anonymous networks. *Information and Computation*, 113(2):312–330, September 1994.

7. Marcos Kawazoe Aguilera and Sam Toueg. A simple bivalency proof that t-resilient consensus requires $t + 1$ rounds. *Information Processing Letters*, 71(3-4):155–158, 1999.

8. J. Alemany and E. W. Felten. Performance issues in non-blocking synchronization on shared-memory multiprocessors. In *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing*, pages 125–134, 1992.

9. Y. Amir, D. Dolev, S. Kramer, and D. Malki. *Total Ordering of Messages in Broadcast Domains*. Technical Report CS92-9, Dept. of Computer Science, The Hebrew University of Jerusalem, 1992.

10. Y. Amir, L. E. Moser, P. M. Melliar-Smith, D.A. Agarwal, and P. Ciarfella. The totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems*, 13(4):311–342, 1995.

11. Yair Amir, Danny Dolev, Shlomo Kramer, and Dalia Malki. Transis: A communication sub-system for high availability. In *Proceedings of the 22nd Annual International Symposium on Fault-Tolerant Computing*, pages 76–84, 1992.

12. Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

13. James Anderson. Composite registers. *Distributed Computing*, 6(3):141–154, April 1993.

14. James Anderson. Multi-writer composite registers. *Distributed Computing*, 7(4):175–196, May 1994.

15. James H. Anderson, Yong-Jik Kim, and Ted Herman. Shared-memory mutual exclusion: major research trends since 1986. *Distributed Computing*, 16(2–3):75–110, 2003.

16. Thomas E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

17. Dana Angluin. Local and global properties in networks of processors. In *Proceedings of the 12th ACM Symposium on Theory of Computing*, pages 82–93, 1980.

18. ANSI/IEEE. *Local Area Networks: Token Ring Access Method and physical Layer Specifications, Std 802.5*. Technical report, 1989.

19. Eshrat Arjomandi, Michael J. Fischer, and Nancy A. Lynch. Efficiency of synchronous versus asynchronous distributed systems. *Journal of the ACM*, 30(3):449–456, July 1983.

20. James Aspnes. Lower bounds for distributed coin-flipping and randomized consensus. *Journal of the ACM*, 45(3):415–450, 1998.

21. James Aspnes. Randomized protocols for asynchronous consensus. *Distributed Computing*, 16(2–3):165–175, 2003.

22. James Aspnes and Maurice Herlihy. Wait-free data structures in the asynchronous PRAM model. In *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 340–349, 1990.

23. James Aspnes and Orli Waarts. Randomized consensus in expected $O(N \log^2 N)$ operations per processor. *SIAM Journal on Computing*, 25(5):1024–1044, October 1996.

24. Hagit Attiya. Efficient and robust sharing of memory in message-passing systems. *Journal of Algorithms*, 34(1):109–127, 2000.

25. Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):121–132, January 1995.

26. Hagit Attiya, Amotz Bar-Noy, Danny Dolev, David Peleg, and Rudiger Reischuk. Renaming in an asynchronous environment. *Journal of the ACM*, 37(3):524–548, July 1990.

27. Hagit Attiya, Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Bounds on the time to reach agreement in the presence of timing uncertainty. *Journal of the ACM*, 41(1):122–152, January 1994.

28. Hagit Attiya and Roy Friedman. A correctness condition for high-performance multiprocessors. *SIAM Journal on Computing*, 27(6):1637–1670, 1998.

29. Hagit Attiya, Amir Herzberg, and Sergio Rajsbaum. Clock synchronization under different delay assumptions. *SIAM Journal on Computing*, 25(2):369–389, April 1996.

30. Hagit Attiya and Marios Mavronicolas. Efficiency of semisynchronous versus asynchronous networks. *Mathematical Systems Theory*, 27(6):547–571, Nov./Dec. 1994.

31. Hagit Attiya and Ophir Rachman. Atomic snapshots in $O(n \log n)$ operations. *SIAM Journal on Computing*, 27(2):319–340, March 1998.

32. Hagit Attiya and Sergio Rajsbaum. The combinatorial structure of wait-free solvable tasks. *SIAM Journal on Computing*, 31(4):1286–1313, 2002.

33. Hagit Attiya, Marc Snir, and Manfred Warmuth. Computing on an anonymous ring. *Journal of the ACM*, 35(4):845–876, October 1988.

34. Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems*, 12(2):91–122, May 1994.

35. Hagit Attiya and Jennifer L. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. McGraw-Hill Publishing Company, May 1998.

36. Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32(4):804–823, October 1985.

37. Baruch Awerbuch. New distributed depth-first-search algorithm. *Information Processing Letters*, 20(3):147–150, April 1985.

38. Baruch Awerbuch. Reducing complexities of distributed maximum flow and breadth-first-search algorithms by means of network synchronization. *Networks*, 15(4):425–437, Winter 1985.

39. Baruch Awerbuch. Optimal distributed algorithms for minmum weight spanning tree, counting, leader election and related problems. In *Proceedings of the 19th ACM Symposium on Theory of Computing*, pages 230–240. ACM, 1987.

40. Baruch Awerbuch and David Peleg. Network synchronization with polylogarithmic overhead. In *Proceedings of the 31th IEEE Symposium on Foundations of Computer Science*, volume II, pages 514–522, 1990.

41. Ozalp Babaoglu, Renzo Davoli, and Alberto Montresor. Group communication in partitionable systems: Specification and algorithms. *IEEE Transactions on Software Engineering*, 27(4):308–336, 2001.

42. Ozalp Babaoglu and Keith Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In Sape Mullender, editor, *Distributed Systems*, chapter 4. Addison-Wesley Publishing Company, Wokingham, 2nd edition, 1993.

43. Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):180–205, March 1992.

44. Amotz Bar-Noy, Danny Dolev, Cynthia Dwork, and H. Raymond Strong. Shifting gears: Changing algorithms on the fly to expedite Byzantine agreement. *Information and Computation*, 97(2):205–233, April 1992.

45. Valmir Barbosa. *An Introduction to Distributed Algorithms*. MIT Press, 1996.

46. Rida A. Bazzi and Gil Neiger. The complexity of almost-optimal coordination. *Algorithmica*, 17(3):308–321, March 1997.

47. Rida Adnan Bazzi. *Automatically Improving the Fault-Tolerance in Distributed Systems*. PhD thesis, College of Computing, Georgia Institute of Technology, 1994. Technical Report GIT-CC-94-62.

48. Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing*, pages 27–30, 1983.

49. J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the 2nd*

Annual ACM Symposium on Principles and Practice of Parallel Processing, pages 168–176, 1990.

50. Pioter Berman and Juan Garay. Cloture votes: $n/4$-resilient distributed consensus in $t + 1$ rounds. *Mathematical Systems Theory*, 26(1):3–19, 1993.

51. B. N. Bershad. Practical considerations for non-blocking concurrent objects. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 264–274, 1993.

52. Ofer Biran, Shlomo Moran, and Shmuel Zaks. A combinatorial characterization of the distributed 1-solvable tasks. *Journal of Algorithms*, 11(3):420–440, September 1990.

53. Ken Birman and Tommy Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

54. Ken Birman, Andre Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

55. Ken Birman and Robert van Renesse (eds.). *Reliable Distributed Programming with the Isis Toolkit*. IEEE Computer Society Press, 1993.

56. R. Bisiani and M. Ravishankar. PLUS: A distributed shared-memory system. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 115–124, 1990.

57. J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications*. MacMillan, London and Basingstoke, 1976.

58. Elizabeth Borowsky and Eli Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *Proceedings of the 25th ACM Symposium on Theory of Computing*, pages 91–100, New-York, 1993.

59. Elizabeth Borowsky, Eli Gafni, Nancy Lynch, and Sergio Rajsbaum. The BG distributed simulation algorithm. *Distributed Computing*, 14(3):127–146, 2001.

60. Gabriel Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation*, 75(2):130–143, November 1987.

61. James E. Burns. *A Formal Model for Message Passing Systems*. Technical Report 91, Indiana University, September 1980.

62. James E. Burns, Paul Jackson, Nancy A. Lynch, Michael J. Fischer, and Gary L. Peterson. Data requirements for implementation of n-process mutual exclusion using a single shared variable. *Journal of the ACM*, 29(1):183–205, January 1982.

63. James E. Burns and Nancy A. Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation*, 107(2):171–184, December 1993.

64. James E. Burns and Gary L. Peterson. The ambiguity of choosing. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, pages 145–158, 1989.

65. R. Canetti and T. Rabin. Fast asynchronous Byzantine agreement with optimal resilience. In *Proceedings of the 25th ACM Symposium on Theory of Computing*, pages 42–51, 1993.

66. Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, 1996.

67. Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

68. K. Mani Chandy. *Essence of Distributed Snapshots*. Technical Report CS-TR-89-05, California Institute of Technology, 1989.

69. K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

70. Ernest Chang and Rosemary Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Communications of the ACM*, 22(5):281–283, May 1979.

71. Jo-Mei Chang and N. F. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.

72. Bernadette Charron-Bost. Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, 39:11–16, July 1991.

73. Soma Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Information and Computation*, 103(1):132–158, July 1993.

74. Soma Chaudhuri, Rainer Gawlick, and Nancy Lynch. Designing algorithms for distributed systems with partially synchronized clocks. In *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing*, pages 121–132, 1993.

75. Soma Chaudhuri, Maurice Herlihy, Nancy A. Lynch, and Mark R. Tuttle. Tight bounds for k-set agreement. *Journal of the ACM*, 47(5):912–943, 2000.

76. Soma Chaudhuri, Martha J. Kosa, and Jennifer L. Welch. One-write algorithms for multivalued regular and atomic registers. *Acta Informatica*, 37(3):161–192, 2000.

77. D. R. Cheriton and W. Zwaenepoel. Distributed process groups in the V kernel. *ACM Transactions on Computer Systems*, 2(3):77–107, May 1985.

78. To-Yat Cheung. Graph traversal techniques and the maximum flow problem in distributed computation. *IEEE Transactions on Software Engineering*, 9(4):504–512, July 1983.

79. Gregory Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 33(4):427–469, December 2001.

80. B. Chor and C. Dwork. Randomization in Byzantine agreement. In *Advances in Computing Research 5: Randomness and Computation*, pages 443–497. JAI Press, 1989.

81. Benny Chor, Amos Israeli, and Ming Li. Wait-free consensus using asynchronous hardware. *SIAM Journal on Computing*, 23(4):701–712, August 1994.

82. Randy Chow and Theodore Johnson. *Distributed Operating Systems and Algorithms*. Addison-Wesley Publishing Company, 1997.

83. Francis Chu. Reducing Ω to $\diamond\mathcal{W}$. *Information Processing Letters*, 67:289–293, 1998.

84. Brian A. Coan. A compiler that increases the fault-tolerance of asynchronous protocols. *IEEE Transactions on Computers*, 37(12):1541–1553, December 1988.

85. G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems, Concepts and Designs*. Addison-Wesley Publishing Company, 2nd edition, 1994.

86. F. Cristian, R. Beijer, and S. Mishra. A performance comparison of asynchronous atomic broadcast protocols. *Distributed Systems Engineering Journal*, 1(4):177–201, 1994.

87. Flaviu Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3(3):146–158, July 1989.

88. Flaviu Cristian, H. Aghili, Ray Strong, and Danny Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. *Information and Computation*, 118(1):158–179, April 1995.

89. Carole Delporte-Gallet, Hugues Fauconnier, and Rachid Guerraoui. Failure detection lower bounds on registers and consensus. In *Proceedings of the 16th International Conference on Distributed Computing*, pages 237–251, 2002.

90. E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

91. Danny Dolev. The Byzantine generals strike again. *Journal of Algorithms*, 3(1):14–30, March 1982.

92. Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

93. Danny Dolev, Joseph Y. Halpern, Barbara Simons, and H. Raymond Strong. Dynamic fault-tolerant clock synchronization. *Journal of the ACM*, 42(1):143–185, January 1995.

94. Danny Dolev, Joseph Y. Halpern, and H. Raymond Strong. On the possibility and impossibility of achieving clock synchronization. *Journal of Computer and System Sciences*, 32(2):230–250, April 1986.

95. Danny Dolev, Maria Klawe, and Michael Rodeh. An $O(n \log n)$ unidirectional distributed algorithm for extrema finding in a circle. *Journal of Algorithms*, 3(3):245–260, September 1982.

96. Danny Dolev, Nancy A. Lynch, Shlomit S. Pinter, Eugene W. Stark, and William E. Weihl. Reaching approximate agreement in the presence of faults. *Journal of the ACM*, 33(3):499–516, July 1986.

97. Danny Dolev, Dalia Malki, and H. Raymond Strong. *An asynchronous membership protocol that tolerates partitions*. Technical Report CS94-6, Institute of Computer Science, The Hebrew University, 1994.

98. Danny Dolev and Nir Shavit. Bounded concurrent time-stamping. *SIAM Journal on Computing*, 26(2):418–455, April 1997.

99. Danny Dolev and H. Raymond Strong. Authenticated algorithms for Byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, November 1983.

100. M. Dubois and C. Scheurich. Memory access dependencies in shared-memory multiprocessors. *IEEE Transactions on Software Engineering*, 16(6):660–673, June 1990.

101. Cynthia Dwork and Yoram Moses. Knowledge and common knowledge in a Byzantine environment: Crash failures. *Information and Computation*, 88(2):156–186, October 1990.

102. Cynthia Dwork and Orli Waarts. Simple and efficient bounded concurrent timestamping and the traceable use abstraction. *Journal of the ACM*, 46(5):633–666, September 1999.

103. E. Allen Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 16, pages 996–1072. Elsevier Science Publisher B. V., Amsterdam, 1990.

104. P. Feldman and S. Micali. Optimal algorithms for Byzantine agreement. In *Proceedings of the 20th ACM Symposium on Theory of Computing*, pages 162–172, 1988.

105. Faith E. Fich and Eric Ruppert. Hundreds of impossibility results for distributed computing. *Distributed Computing*, 16(2–3):121–163, 2003.

106. C. Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24(8):28, August 1991.

107. Michael J. Fischer and Nancy A. Lynch. A lower bound for the time to assure interactive consistency. *Information Processing Letters*, 14(4):183–186, June 1982.

108. Michael J. Fischer, Nancy A. Lynch, James E. Burns, and Allan Borodin. Distributed FIFO allocation of identical resources using small shared space. *ACM Transactions on Programming Languages and Systems*, 11(1):90–114, January 1989.

109. Michael J. Fischer, Nancy A. Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1(1):26–39, January 1986.

110. Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty processor. *Journal of the ACM*, 32(2):374–382, April 1985.

111. Greg N. Frederickson and Nancy Lynch. Electing a leader in a synchronous ring. *Journal of the ACM*, 34(1):98–115, January 1987.

112. Roy Friedman. *Consistency Conditions for Distributed Shared Memories*. PhD thesis, Department of Computer Science, The Technion, 1994.

113. Roy Friedman. Implementing hybrid consistency with high-level synchronization operations. *Distributed Computing*, 9(3):119–129, December 1995.

114. Roy Friedman and Robbert van Renesse. Strong and weak synchrony in horus. In *Proceedings of the 16th IEEE Symposium On Reliable Distributed Systems*, 1996.

115. Eli Gafni. Perspectives on distributed network protocols: A case for building blocks. In *Proceedings IEEE MILCOM '86*, 1986.

116. Eli Gafni and Leslie Lamport. Disk paxos. *Distributed Computing*, 16(1):1–20, 2003.

117. Robert Gallager. Finding a leader in a network with $o(e)+o(n \log n)$ messages. MIT Internal Memorandum, 1977.

118. Robert Gallager, Pierre Humblet, and Philip Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and Systems*, 5(1):66–77, January 1983.

119. Juan Garay and Yoram Moses. Fully polynomial Byzantine agreement for $n > 3t$ processors in $t + 1$ rounds. *SIAM Journal on Computing*, 27(1):247–290, February 1998.

120. Hector Garcia-Molina and Annemarie Spauster. Ordered and reliable multicast communication. *ACM Transactions on Programming Languages and Systems*, 9:242–271, August 1991.

121. Vijay K. Garg and Brian Waldecker. Detection of weak unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):299–307, March 1994.

122. K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, 1990.

123. Oded Goldreich and Erez Petrank. The best of both worlds: Guaranteeing termination in fast randomized Byzantine agreement protocols. *Information Processing Letters*, 36(1):45–49, October 1990.

124. Ajei Gopal and Sam Toueg. Inconsistency and contamination. In *Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing*, pages 257–272, 1991.

125. Gary Graunke and Shreekant Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 23(6):60–69, June 1990.

126. Rachid Guerraoui. Non-blocking atomic commit in asynchronous distributed systems with failure detectors. *Distributed Computing*, 15(1):17–25, 2002.

127. Rajiv Gupta, Scott A. Smolka, and Shaji Bhaskar. On randomization in sequential and distributed algorithms. *ACM Computing Surveys*, 26(1):7–86, March 1994.

128. Vassos Hadzilacos. *Issues of Fault Tolerance in Concurrent Computations*. PhD thesis, Aiken Computation Laboratory, Harvard University, June 1984.

129. Vassos Hadzilacos and Sam Toueg. *A modular approach to fault-tolerant broadcasts and related problems*. Technical Report TR 94-1425, Cornell University, Dept. of Computer Science, Cornell University, Ithaca, NY 14853, May 1994.

130. Joseph Y. Halpern, Nimrod Megiddo, and A. A. Munshi. Optimal precision in the presence of uncertainty. *Journal of Complexity*, 1(2):170–196, December 1985.

131. Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

132. Maurice Herlihy and Nir Shavit. The asynchronous computability theorem for t-resilient tasks. In *Proceedings of the 25th ACM Symposium on Theory of Computing*, pages 111–120, 1993.

133. Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(6):858–923, 1999.

134. Maurice P. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

135. Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

136. Lisa Higham. *Randomized Distributed Computing on Rings*. PhD thesis, Department of Computer Science, University of British Columbia, 1989. Technical Report 89-05.

137. Lisa Higham and Teresa Przytycka. A simple, efficient algorithm for maximum finding on rings. *Information Processing Letters*, 58(6):319–324, 1996.

138. Lisa Higham and Jolanta Warpechowska-Gruca. *Notes on Atomic Broadcast*. Technical Report 95/562/14, University of Calgary, Department of Computer Science, 1995.

139. D. S. Hirschberg and J. B. Sinclair. Decentralized extrema-finding in circular configurations of processors. *Communications of the ACM*, 23(11):627–628, November 1980.

140. Amos Israeli and Ming Li. Bounded time-stamps. *Distributed Computing*, 6(4):205–209, July 1993.

141. Alon Itai and Michael Rodeh. Symmetry breaking in distributed networks. *Information and Computation*, 88(1):60–87, September 1990.

142. Joseph JaJa. *An Introduction to Parallel Algorithms*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1992.

143. B. Janssens and W. K. Fuchs. Relaxing consistency in recoverable distributed shared memory. In *Proceedings of the 23rd Annual International Symposium on Fault-Tolerant Computing*, pages 155–165, 1993.

144. Prasad Jayanti. Robust wait-free hierarchies. *Journal of the ACM*, 44(4):592–614, July 1997.

145. David B. Johnson and Willy Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. *Journal of Algorithms*, 11(3):462–491, September 1990.

146. Frans Kaashoek, Andy Tanenbaum, S. Hummel, and Henri Bal. An efficient reliable broadcast protocol. *Operating Systems Review*, 23(4):5–19, October 1989.

147. Sundar Kanthadai and Jennifer L. Welch. Implementation of recoverable distributed shared memory by logging writes. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 116–124, 1996.

148. J-H. Kim and N. H. Vaidya. Recoverable distributed shared memory using the competitive update protocol. In *Proceedings of the 1995 Pacific Rim International Conference on Fault-Tolerant Systems*, pages 152–157, 1995.

149. Martha J. Kosa. Making operations of concurrent data types fast. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, pages 32–41, 1994.

150. Eyal Kushilevitz, Yishay Mansour, Michael O. Rabin, and David Zuckerman. Lower bounds for randomized mutual exclusion. *SIAM Journal on Computing*, 27(6):1550–1563, 1998.

151. Eyal Kushilevitz and Michael O. Rabin. Randomized mutual exclusion algorithms revisited. In *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing*, pages 275–284, 1992.

152. T. H. Lai and T. H. Yang. On distributed snapshots. In Zhonghua Yang and T. Anthony Marsland, editors, *Global States and Time in Distributed Systems*. IEEE Computer Society Press, 1994.

153. Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 18(8):453–455, August 1974.

154. Leslie Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2:95–114, 1978.

155. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–564, July 1978.

156. Leslie Lamport. How to make a multiprocessor that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

157. Leslie Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions on Programming Languages and Systems*, 6(2):254–280, April 1984.

158. Leslie Lamport. On interprocess communication, Part I: Basic formalism. *Distributed Computing*, 1(2):77–85, 1986.

159. Leslie Lamport. On interprocess communication, Part II: Algorithms. *Distributed Computing*, 1(2):86–101, 1986.

160. Leslie Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, February 1987.

161. Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

162. Leslie Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, January 1985.

163. Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

164. F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, 1992.

165. Gérard LeLann. Distributed systems, towards a formal approach. In *IFIP Congress Proceedings*, pages 155–160, 1977.

166. Kai Li. *Shared Virtual Memory on Loosely-Coupled Processors*. PhD thesis, Yale University, New Haven, Connecticut, September 1986. Number: YALEU/DCS/RR-492.

167. Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Programming Languages and Systems*, 7(4):321–359, November 1989.

168. Ming Li, John Tromp, and Paul M. B. Vitányi. How to share concurrent wait-free variables. *Journal of the ACM*, 43(4):723–746, July 1996.

169. Kal Lin and Vassos Hadzilacos. Asynchronous group membership with oracles. In *Proceedings of the 13th International Conference on Distributed Computing*, pages 79–93, 1999.

170. Richard Lipton and John Sandberg. *PRAM: A Scalable Shared Memory*. Technical Report CS-TR-180-88, Computer Science Department, Princeton University, September 1988.

171. Wai-Kau Lo and Vassos Hadzilacos. Using failure detectors to solve consensus in asynchronous sharde-memory systems. In *Proceedings of the 8th International Workshop on Distributed Algorithms*, pages 280–295, 1994.

172. Wai-Kau Lo and Vassos Hadzilacos. All of us are smarter than any of us: Nondeterministic wait-free hierarchies are not robust. *SIAM Journal on Computing*, 30(3):689–728, 2000.

173. Michael C. Loui and Hosame H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. In *Advances in Computing Research, Vol. 4*, pages 163–183. JAI Press, Inc., 1987.

174. Jennifer Lundelius and Nancy Lynch. An upper and lower bound for clock synchronization. *Information and Control*, 62(2/3):190–204, Aug./Sept. 1984.

175. Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

176. Nancy Lynch and Michael Fischer. On describing the behavior and implementation of distributed systems. *Theoretical Computer Science*, 13(1):17–43, January 1981.

177. Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 137–151. ACM, 1987. A full version is available as MIT Technical Report MIT/LCS/TR–387.

178. Stephen R. Mahaney and Fred B. Schneider. Inexact agreement: Accuracy, precision, and graceful degradation. In *Proceedings of the 4th Annual ACM Symposium on Principles of Distributed Computing*, pages 237–249, 1985.

179. Dahlia Malkhi and Michael K. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

180. Keith Marzullo and Susan S. Owicki. Maintaining the time in a distributed system. *Operating Systems Review*, 19(3):44–54, 1985.

181. Friedemann Mattern. Virtual time and global states of distributed systems. In M. Cosnard et. al, editor, *Parallel and Distributed Algorithms: Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B. V., 1989.

182. Marios Mavronicolas and Dan Roth. Linearizable read/write objects. *Theoretical Computer Science*, 220(1):267–319, 1999.

183. P. M. Melliar-Smith, Louise E. Moser, and Vivek Agrawala. Broadcast protocols for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):17–25, January 1990.

184. John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

185. Michael Merritt. Notes on the Dolev-Strong lower bound for Byzantine agreement. Unpublished manuscript, 1985.

186. David L. Mills. Internet time synchronization: The Network Time Protocol. *IEEE Transactions on Communications*, 39(10):1482–1493, October 1991.

187. David L. Mills. Improved algorithms for synchronizing computer network clocks. *IEEE/ACM Transactions on Networking*, 3(3):245–254, June 1995.

188. Mark Moir and James H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Science of Computer Programming*, 25(1):1–39, October 1995.

189. Mark Moir and Juan A. Garay. Fast long-lived renaming improved and simplified. In *Proceedings of the 10th International Workshop on Distributed Algorithms*, volume 1151 of *Lecture Notes in Computer Science*, pages 287–303. Springer-Verlag, 1996.

190. Shlomo Moran. Using approximate agreement to obtain complete disagreement: The output structure of input free asynchronous computations. In *Proceedings of the 3rd Israel Symposium on Theory of Computing and Systems*, pages 251–257, 1995.

191. Carol Morgan. Global and logical time in distributed systems. *Information Processing Letters*, 20(4):189–194, May 1985.

192. L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended virtual synchrony. In *Proceedings of the 14th International Conference on Distributed Computing Systems*, pages 56–65, 1994.

193. Yoram Moses and Mark R. Tuttle. Programming simultaneous actions using common knowledge. *Algorithmica*, 3(1):121–169, 1988.

194. Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

195. Sape Mullender, editor. *Distributed Systems*. Addison-Wesley Publishing Company, 2nd edition, 1993.

196. Gil Neiger. Distributed consensus revisited. *Information Processing Letters*, 49(4):195–201, February 1994.

197. Gil Neiger. Failure detectors and the wait-free hierarchy. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 100–109, 1995.

198. Gil Neiger and Sam Toueg. Automatically increasing the fault-tolerance of distributed algorithms. *Journal of Algorithms*, 11(3):374–419, September 1990.

199. Gil Neiger and Sam Toueg. Simulating synchronized clocks and common knowledge in distributed systems. *Journal of the ACM*, 40(2):334–367, April 1993.

200. Nuno Neves, Miguel Castro, and Paulo Guedes. A checkpoint protocol for an entry consistent shared memory system. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, pages 121–129, 1994.

201. Bill Nitzberg and Virginia Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE Computer*, 24(8):52–60, August 1991.

202. Gary Nutt. *Centralized and Distributed Operating Systems*. Prentice-Hall, Inc., 1992.

203. Susan Owicki and David Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(4):319–340, 1976.

204. Susan Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, July 1982.

205. J. Pachl, E. Korach, and D. Rotem. Lower bounds for distributed maximum-finding algorithms. *Journal of the ACM*, 31(4):905–918, October 1984.

206. Boaz Patt-Shamir and Sergio Rajsbaum. A theory of clock synchronization. In *Proceedings of the 26th ACM Symposium on Theory of Computing*, pages 810–819, 1994.

207. Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

208. David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. SIAM Mongraphs on Discrete Mathematics and Applications. Philadelphia, PA, 2000.

209. David Peleg and Jeffrey D. Ullman. An optimal synchronizer for the hypercube. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 77–85, 1987.

210. Kenneth J. Perry and Sam Toueg. Distributed agreement in the presence of processor and communication faults. *IEEE Transactions on Software Engineering*, 12(3):477–482, March 1986.

211. Gary L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12:115–116, June 1981.

212. Gary L. Peterson. An $O(n \log n)$ unidirectional algorithm for the circular extrema problem. *ACM Transactions on Programming Languages and Systems*, 4(4):758–762, October 1982.

213. Gary L. Peterson. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 5(1):46–55, January 1983.

214. Gary L. Peterson and Michael J. Fischer. Economical solutions for the critical section problem in a distributed system. In *Proceedings of the 9th ACM Symposium on Theory of Computing*, pages 91–97, 1977.

215. Larry L. Peterson, Nick C. Buchholz, and Richard D. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–246, August 1989.

216. S. A. Plotkin. Sticky bits and universality of consensus. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, pages 159–175, 1989.

217. Athanassios S. Poulakidas and Ambuj K. Singh. Online replication of shared variables. In *Proceedings of the 17th International Conference on on Distributed Computing Systems*, pages 500–507, 1997.

218. David Powell, Peter Barrett, Gottfried Bonn, Marc Chérèque, Douglas Seaton, and Paulo Veríssimo. The Delta-4 distributed fault-tolerant architecture. In Thomas L. Casavant and Mukesh Singhal, editors, *Readings in Distributed Systems*. IEEE Computer Society Press, 1994.

219. S. Prakash, Y. Lee, and T. Johnson. A nonblocking algorithm for shared queues using compare-and-swap. *IEEE Transactions on Computers*, 43:548–559, May 1994.

220. Roberto De Prisco, Butler W. Lampson, and Nancy A. Lynch. Revisiting the PAXOS algorithm. *Theoretical Computer Science*, 243(1-2):35–91, 2002.

221. J. Protic, M. Tomasevic, and V. Milutinovic. A survey of distributed shared memory systems. In *Proceedings of the 28th Hawaii Conference on System Sciences*, volume I, pages 74–84, 1995.

222. Jelica Protic, Milo Tomasevic, and Veljko Milutinovic, editors. *Distributed Shared Memory: Concepts and Systems*. IEEE Computer Society Press, August 1997.

223. Michael O. Rabin. N-process mutual exclusion with bounded waiting by $4 \cdot \log n$-valued shared variable. *Journal of Computer and System Sciences*, 25(1):66–75, August 1982.

224. Ophir Rachman. Anomalies in the wait-free hierarchy. In *Proceedings of the 8th International Workshop on Distributed Algorithms*, volume 857 of *Lecture Notes in Computer Science*, pages 156–163. Springer-Verlag, 1994.

225. M. Raynal. *Algorithms for Mutual Exclusion*. MIT Press, 1986.

226. Michel Raynal. *Networks and Distributed Computation: Concepts, Tools, and Algorithms*. MIT Press, 1988.

227. Michel Raynal, Andre Schiper, and Sam Toueg. The causal ordering abstraction and a simple way to implement it. *Information Processing Letters*, 39(6):343–350, September 1991.

228. Michael K. Reiter. Distributing trust with the Rampart toolkit. *Communications of the ACM*, 39(4):71–74, April 1996.

229. G. G. Richard III and M. Singhal. Using logging and asynchronous checkpointing to implement recoverable distributed shared memory. In *Proceedings of the 12th IEEE Symposium on Reliable Distributed Systems*, pages 58–67, 1993.

230. Laura Sabel and Keith Marzullo. Simulating fail-stop in asynchronous distributed systems (brief announcement). In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, pages 399–399, 1994.

231. Laura Sabel and Keith Marzullo. *Election vs. Consensus in Asynchronous Systems*. Technical Report TR95-411, UC San-Diego, 1995.

232. Isaac Saias. Proving probabilistic correctness statements: The case of Rabin's algorithm for mutual exclusion. In *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing*, pages 263–274, 1992.

233. Michael Saks and Fotios Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. In *Proceedings of the 25th ACM Symposium on Theory of Computing*, pages 101–110, 1993.

234. Baruch Schieber and Marc Snir. Calling names on nameless networks. *Information and Computation*, 113(1):80–101, August 1994.

235. Andre Schiper, Jorge Eggli, and Alain Sandoz. A new algorithm to implement causal ordering. In *Proceedings of the 3rd International Workshop on Distributed Algorithms*, number 392 in Lecture Notes in Computer Science, pages 219–232. Springer-Verlag, 1989.

236. F. B. Schneider, D. Gries, and R. D. Schlichting. Fault-tolerant broadcasts. *Science of Computer Programming*, 4(1):1–15, April 1984.

237. Fred B. Schneider. Synchronization in distributed programs. *ACM Transactions on Programming Languages and Systems*, 4(2):125–148, April 1982.

238. Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

239. Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7(3):149–174, 1994.

240. Adrian Segall. Distributed network protocols. *IEEE Transactions on Information Theory*, IT-29(1):23–35, January 1983.

241. Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, February 1997.

242. Barbara Simons, Jennifer Welch, and Nancy Lynch. An overview of clock synchronization. In *Fault-Tolerant Distributed Computing*, number 448 in

Lecture Notes in Computer Science, pages 84–96. Springer-Verlag, 1984. Also IBM Technical Report RJ 6505, October 1988.

243. Ambuj K. Singh, James H. Anderson, and Mohamed G. Gouda. The elusive atomic register. *Journal of the ACM*, 41(2):311–339, March 1994.

244. A. Prasad Sistla and Jennifer L. Welch. Efficient distributed recovery using message logging. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, pages 223–238, 1989.

245. T. K. Srikanth and Sam Toueg. Optimal clock synchronization. *Journal of the ACM*, 34(3):626–645, July 1987.

246. T. K. Srikanth and Sam Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2(2):80–94, August 1987.

247. Robert E. Strom and Shaula A. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.

248. Michael Stumm and Songnian Zhou. Fault-tolerant distributed shared memory algorithms. In *Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing*, pages 719–724, 1990.

249. Andrew Tanenbaum. *Modern Operating Systems*. Prentice-Hall, Inc., 1992.

250. Andrew Tanenbaum. *Distributed Operating Systems*. Prentice-Hall, Inc., 1995.

251. Andrew Tanenbaum. *Computer Networks*. Prentice-Hall, Inc., 5th edition, 1996.

252. Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 1994.

253. C. B. Tompkins. Sperner's lemma and some extensions. In E. F. Beckenbach, editor, *Applied Combinatorial Mathematics*, chapter 15, pages 416–455. John Wiley and Sons, Inc., New York, 1964.

254. John Turek, Dennis Shasha, and Sundeep Prakash. Locking without blocking: Making lock based concurrent data structure algorithms nonblocking. In *Proceedings of the 11th Annual ACM Symposium on Principles of Database Systems*, pages 212–222, 1992.

255. Russel Turpin and Brian Coan. Extending binary Byzantine agreement to multivalued Byzantine agreement. *Information Processing Letters*, 18(2):73–76, February 1984.

256. Robbert van Renesse, Kenneth P. Birman, and Silvano Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, April 1996.

257. K. Vidyasankar. Converting Lamport's regular register to atomic register. *Information Processing Letters*, 28:287–290, August 1988.

258. Paul M. B. Vitányi and Baruch Awerbuch. Atomic shared register access by asynchronous hardware. In *Proceedings of the 27th IEEE Symposium on Foundations of Computer Science*, pages 233–243. IEEE, 1986.

259. Jennifer Lundelius Welch. Simulating synchronous processors. *Information and Computation*, 74(2):159–171, August 1987.

260. Jennifer Lundelius Welch and Nancy A. Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation*, 77(1):1–36, April 1988.

261. J. H. Wensley et al. SIFT: The design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE*, 66(10):1240–1255, October 1978.

262. K.-L. Wu and W. K. Fuchs. Recoverable distributed shared virtual memory. *IEEE Transactions on Computers*, 39(4):460–469, April 1990.

263. Zhonghua Yang and T. Anthony Marsland, editors. *Global States and Time in Distributed Systems*. IEEE Computer Society Press, 1994.

Distributed Computing: Fundamentals, Simulations and Advanced Topics, Second Edition
Hagit Attiya and Jennifer Welch
 ISBN: 0-471-45324-2

Index

WILEY SERIES ON PARALLEL AND DISTRIBUTED COMPUTING

Series Editor: Albert Y. Zomaya

Parallel and Distributed Simulation Systems / Richard Fujimoto

Surviving the Design of Microprocessor and Multimicroprocessor Systems: Lessons Learned / Veljko Milutinović

Mobile Processing in Distributed and Open Environments / Peter Sapaty

Introduction to Parallel Algorithms / C. Xavier and S. S. Iyengar

Solutions to Parallel and Distributed Computing Problems: Lessons from Biological Sciences / Albert Y. Zomaya, Fikret Ercal, and Stephan Olariu (*Editors*)

New Parallel Algorithms for Direct Solution of Linear Equations / C. Siva Ram Murthy, K. N. Balasubramanya Murthy, and Srinivas Aluru

Practical PRAM Programming / Joerg Keller, Christoph Kessler, and Jesper Larsson Traeff

Computational Collective Intelligence / Tadeusz M. Szuba

Parallel and Distributed Computing: A Survey of Models, Paradigms, and Approaches / Claudia Leopold

Fundamentals of Distributed Object Systems: A CORBA Perspective / Zahir Tari and Omran Bukhres

Pipelined Processor Farms: Structured Design for Embedded Parallel Systems / Martin Fleury and Andrew Downton

Handbook of Wireless Networks and Mobile Computing / Ivan Stojmenoviić *(Editor)*

Internet-Based Workflow Management: Toward a Semantic Web / Dan C. Marinescu

Parallel Computing on Heterogeneous Networks / Alexey L. Lastovetsky

Tools and Environments for Parallel and Distributed Computing / S. Hariri and M. Parashar (*Editors*)

Distributed Computing: Fundamentals, Simulations and Advanced Topics, *Second Edition* / Hagit Attiya and Jennifer Welch

Distributed Computing: Fundamentals, Simulations and Advanced Topics, Second Edition
Hagit Attiya and Jennifer Welch
 ISBN: 0-471-45324-2