5

Fault-Tolerant Consensus

Coordination problems require processors to agree on a common course of action. Such problems are typically very easy to solve in reliable systems of the kind we have considered so far. In real systems, however, the various components do not operate correctly all the time. In this chapter, we start our investigation of the problems arising when a distributed system is unreliable. Specifically, we consider systems in which processors' functionality is incorrect.

In Section 5.1, we consider benign types of failures in synchronous message passing systems. In this case, a faulty processor crashes, that is, stops operating, but does not perform wrong operations (e.g., deliver messages that were not sent). We study the *consensus* problem, a fundamental coordination problem that requires processors to agree on a common output, based on their (possibly conflicting) inputs. Matching upper and lower bounds on the number of rounds required for solving consensus are shown.

In Section 5.2, we consider more severe types of (mis)behavior of the faulty processors, still in synchronous message-passing systems. We assume failures are *Byzantine*, that is, a failed processor may behave arbitrarily. We show that if we want to solve consensus, less than a third of the processors can be faulty. Under this assumption, we present two algorithms for reaching consensus in the presence of Byzantine failures. One algorithm uses the optimal number of rounds but has exponential message complexity; the second algorithm has polynomial message complexity, but it doubles the number of rounds.

Finally, we turn to asynchronous systems. We show that consensus cannot be achieved by a deterministic algorithm in asynchronous systems, even if only one processor fails in a benign manner by simply crashing. This result holds whether communication is via messages or through shared read/write variables. Chapter 15 considers the ability of other types of shared variables to solve consensus. Chapter 16 studies weaker coordination problems that can be solved in such asynchronous systems.

In this chapter, we study both synchronous and asynchronous message-passing systems and asynchronous shared memory systems. In each section, we discuss how to modify the model of the respective reliable system to include the specific type of faulty behavior.

5.1 SYNCHRONOUS SYSTEMS WITH CRASH FAILURES

In this section, we discuss a simple scenario for fault-tolerant distributed computing: a synchronous system in which processors fail by simply ceasing to operate. For all message-passing systems in this section, we assume that the communication graph is complete, that is, processors are located at the nodes of a clique. We further assume that the communication links are completely reliable and all messages sent are delivered.

5.1.1 Formal Model

We need to modify the formal definitions from Chapter 2 for a synchronous message-passing system to handle processor crashes.

A vital parameter of the system definition is f, the maximum number of processors that can fail. We call the system *f-resilient.*

Recall that in the reliable case, an execution of the synchronous system consists of a series of rounds. Each round consists of the delivery of all messages pending in *outbuf* variables, followed by one computation event for every processor.

For an f-resilient system, the definition of an execution is modified as follows. There exists a subset F of at most f processors, the *faulty* processors; the set of faulty processors can be different in different executions, so that it is not known in advance which processors are faulty. Each round contains exactly one computation event for every processor not in F and *at most* one computation event for every processor in F. Furthermore, if a processor in F does not have a computation event in some round, then it has no computation event in any subsequent round. Finally, in the last round in which a faulty processor has a computation event, an arbitrary subset of its outgoing messages are delivered.

This last property is quite important and causes the difficulties associated with this failure model. If every crash is a *clean* crash, in which either all or none of the crashed processor's outgoing messages from its last step are delivered, consensus can be solved very efficiently (see Exercise 5.2). But the uncertainty in the effect of the crash means that processors must do more work (e.g., exchange more messages) in order to solve consensus.

Algorithm 15 Consensus algorithm in the presence of crash failures:
code for processor p_i, $0 \le i \le n-1$.

```
Initially $V = \{x\}$                                    // $V$ contains $p_i$'s input

round $k$, $1 \le k \le f+1$:
    send $\{v \in V : p_i$ has not already sent $v\}$ to all processors
    receive $S_j$ from $p_j$, $0 \le j \le n-1, j \ne i$
    $V := V \cup \bigcup_{j=0}^{n-1} S_j$
    if $k = f+1$ then $y := \min(V)$                          // decide
```

5.1.2 The Consensus Problem

Consider a system in which 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 well-ordered 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:

Termination: In every admissible execution, y_i is eventually assigned a value, for every nonfaulty processor p_i.

Agreement: In every execution, if y_i and y_j are assigned, then $y_i = y_j$, for all nonfaulty processors p_i and p_j. That is, nonfaulty processors do not decide on conflicting values.

Validity: In every execution, if, for some value v, $x_i = v$ for all processors p_i, and if y_i is assigned for some nonfaulty processor p_i, then $y_i = v$. That is, if all the processors have the same input, then any value decided upon must be that common input.

For two-element input sets, this validity condition is equivalent to requiring that every nonfaulty decision value be the input of some processor, as Exercise 5.1 asks you to show. Once a processor crashes, it is of no interest to the algorithm, and no requirements are placed on its decision.

Below we show matching upper and lower bounds of $f+1$ on the number of rounds required for reaching consensus in an f-resilient system.

5.1.3 A Simple Algorithm

The pseudocode appears in Algorithm 15. In the algorithm, each processor maintains a set of the values it knows to exist in the system; initially, this set contains only its own input. In later rounds, a processor updates its set by joining it with the sets received from other processors and broadcasts any new additions to the set to all processors. This continues for $f+1$ rounds. At this point, the processor decides on the smallest value in its set.

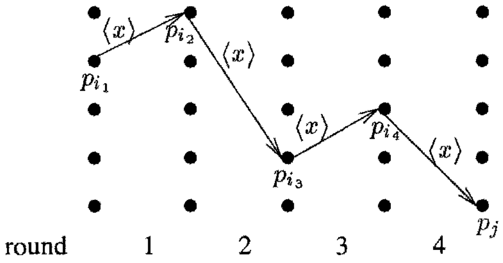

Fig. 5.1 Illustration for the proof of Lemma 5.1, $f = 3$.

Clearly, the algorithm requires exactly $f + 1$ rounds, which implies termination. Furthermore, it is obvious that the validity condition is maintained, because the decision value is an input of some processor. The next lemma is the key to proving that the agreement condition is satisfied.

Lemma 5.1 *In every execution, at the end of round $f + 1$, $V_i = V_j$, for every two nonfaulty processors p_i and p_j.*

Proof. It suffices to show that if $x \in V_i$ at the end of round $f + 1$, then $x \in V_j$ at the end of round $f + 1$, for all nonfaulty processors p_i and p_j.

Let r be the first round in which x is added to V_i (in Line 4), for any nonfaulty processor p_i. If x is initially in V_i, let r be 0. If $r \leq f$ then, in round $r + 1 \leq f + 1$, p_i sends x to each p_j, which causes p_j to add x to V_j, if it is not already present.

Otherwise, suppose $r = f + 1$ and let p_j be a nonfaulty processor that receives x for the first time in round $f + 1$. Then there must be a chain of $f + 1$ processors $p_{i_1}, \ldots, p_{i_{f+1}}$ that transfers the value x to p_j. That is, p_{i_1} sends x to p_{i_2} in round 1, p_{i_2} sends x to p_{i_3} in round 2, etc., and finally p_{i_f} sends x to $p_{i_{f+1}}$ in round f, and $p_{i_{f+1}}$ sends x to p_j in round $f + 1$. (Fig. 5.1 illustrates this situation for $f = 3$.) Since each processor sends a particular value only once, the processors $p_{i_1}, \ldots, p_{i_{f+1}}$ form a set of $f + 1$ distinct processors. Thus there must be at least one nonfaulty processor among $p_{i_1}, \ldots, p_{i_{f+1}}$. However, this processor adds x to its set at a round $\leq f < r$, contradicting the assumption that r is minimal. □

Therefore, nonfaulty processors have the same set in Line 5 and decide on the same value. This implies that the agreement condition is satisfied. Thus we have:

Theorem 5.2 *Algorithm 15 solves the consensus problem in the presence of f crash failures within $f + 1$ rounds.*

5.1.4 Lower Bound on the Number of Rounds

We now present a lower bound of $f + 1$ on the number of rounds required for reaching consensus in the presence of crash failures. This implies that the algorithm presented in Section 5.1.3 is optimal. We assume that $f \leq n - 2$.[1]

The intuition behind the lower bound is that if processors decide too early, they cannot distinguish between admissible executions in which they should make different decisions. The notion of indistinguishability is crucial to this proof and is central in our understanding of distributed systems. To capture this notion formally, we introduce the following definition of a processor's view of the execution.

Definition 5.1 *Let α be an execution and let p_i be a processor. The* view of p_i in α, *denoted by $\alpha|p_i$, is the subsequence of computation and message delivery events that occur in α at p_i together with the state of p_i in the initial configuration of α.*

Chapter 4 included a definition of two (shared memory) configurations being similar for a processor (Definition 4.1). Using the notion of view that was just defined, we can extend the definition of similarity to entire (message passing) executions. However, here we are only concerned if a *nonfaulty* processor cannot distinguish between the executions.

Definition 5.2 *Let α_1 and α_2 be two executions and let p_i be a processor that is nonfaulty in α_1 and α_2. Execution α_1 is* similar *to execution α_2* with respect to p_i, *denoted $\alpha_1 \overset{p_i}{\sim} \alpha_2$, if $\alpha_1|p_i = \alpha_2|p_i$.*

Some technical details in the lower bound proof are made easier if we restrict attention to consensus algorithms in which every processor is supposed to send a message to every other processor at each round. This does not impair the generality of the result, because any consensus algorithm can be modified trivially to conform to this rule by adding dummy messages where necessary. We also assume that every processor keeps a history of all the messages it has received in all the previous rounds, so that a configuration contains the information concerning which processors have failed and how many rounds (or parts of rounds) have elapsed.

An execution is said to be *failure sparse* if there is at most one crash per round. Such executions are very useful in proving the lower bound — even a single failure can cause some information to be lost, and stretching out the failures over more rounds increases the amount of time in which there is uncertainty about the decision. In the remainder of this section, we only consider executions that are prefixes of admissible failure-sparse executions and configurations appearing in such executions. In particular, all definitions in this section are with respect to failure-sparse executions, and we will only explicitly mention "failure sparse" when the argument crucially depends on this property.

[1] If $f = n - 1$ then consensus can be achieved within f rounds, by a small modification to Algorithm 15; see Exercise 5.3.

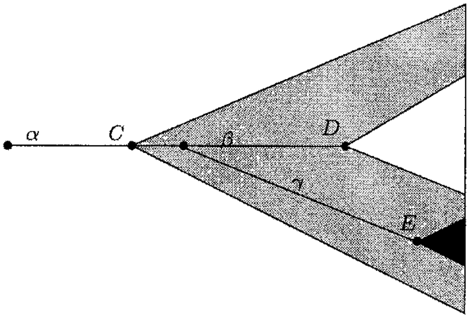

Fig. 5.2 Schematic of the set S of all admissible failure-sparse executions that include configuration C. Solid lines (α, β, γ) represent single executions. Shaded triangle with endpoint C, D or E represents all executions in S that include that configuration. If all decisions in the white triangle are 1, then D is 1-valent; if all decisions in the black triangle are 0, then E is 0-valent; in this case C is bivalent.

A key notion in this proof (and in others) is the set of decisions that can be reached from a particular configuration. The next few definitions formalize this notion.

The *valence* of configuration C is the set of all values that are decided upon by a nonfaulty processor in some configuration that is reachable from C in an (admissible failure sparse) execution that includes C. By the termination condition, the set cannot be empty. C is *univalent* if this set contains one value; it is 0-*valent* if this value is 0, and 1-*valent* if this value is 1. If the set contains two values then C is *bivalent*. Figure 5.2 shows an example of 0-valent, 1-valent, and bivalent configurations.

If some processor has decided in a configuration, the agreement condition implies that the configuration is univalent.

Theorem 5.3 *Any consensus algorithm for n processors that is resilient to f crash failures requires at least $f+1$ rounds in some admissible execution, for all $n \geq f+2$.*

Proof. Consider any consensus algorithm A for n processors and f crash failures, with $n \geq f + 2$.

The proof strategy is, first, to show that there exists an $(f - 1)$-round execution of A in which the configuration at the end is undecided. The next step is to show that with just one more round it is not possible for the processors to decide explicitly. Thus at least $f + 1$ rounds are required for decision.

The $(f-1)$-round execution in the first stage is constructed by induction. Lemma 5.4 shows that there is an "undecided" initial configuration. Lemma 5.5 shows how to construct an undecided k-round execution out of an undecided $(k - 1)$-round execution, up to the limit of $f - 1$. The executions manipulated by the proof are failure-sparse ones, and thus "undecided" here means bivalent with respect to failure-sparse executions.

Lemma 5.4 *Algorithm A has a bivalent initial configuration.*

Proof. Suppose in contradiction that all initial configurations are univalent. Since the initial configuration in which all inputs are 0 is 0-valent and the initial configuration in which all inputs are 1 is 1-valent, there must exist two configurations that differ in the input of only one processor yet have different valences.

In particular, let I_0 be a 0-valent initial configuration and I_1 be a 1-valent initial configuration such that I_0 and I_1 differ only in the input of processor p_i.

Consider the schedule σ in which p_i fails initially and no other processors fail. Assume that σ is long enough to ensure that all the nonfaulty processors decide when starting from I_0. Note that the resulting execution is failure sparse. Since I_0 is 0-valent, applying σ to I_0 results in a decision of 0.

What happens if σ is applied to I_1? The processors other than p_i are nonfaulty and cannot tell the difference between these two executions (formally, the executions are similar with respect to every processor other than p_i). Thus the nonfaulty processors decide 0 in I_1, contradicting the 1-valence of I_1.

Thus there is at least one bivalent initial configuration. □

Lemma 5.5 *For each k, $0 \leq k \leq f-1$, there is a k-round execution of A that ends in a bivalent configuration.*

Proof. The proof is by induction on k. The base case, $k = 0$, follows from Lemma 5.4.

Assume that the lemma is true for $k - 1 \geq 0$ and show it is true for $k \leq f - 1$. Let α_{k-1} be the $(k-1)$-round execution ending in a bivalent configuration whose existence is guaranteed by the inductive hypothesis.

Assume in contradiction that all one-round extensions of α_{k-1} with at most one additional crash end in a univalent configuration.

Without loss of generality, assume that the one-round failure-free extension of α_{k-1}, denoted β_k, leads to a 1-valent configuration. Since α_{k-1} ends in a bivalent configuration, there is another one-round extension of α_{k-1} that ends in a 0-valent configuration. Call this execution γ_k. Since we are working exclusively with failure-sparse execution, exactly one failure occurs in round k of γ_k. In the execution γ_k, let p_i be the processor that crashes and $q_1, \ldots, q_m$ be the processors to whom p_i fails to send (see Fig. 5.3); m is some value between 1 and n inclusive.

For each j, $0 \leq j \leq m$, define execution α_k^j to be the one-round extension of α_{k-1} in which p_i fails to send to $q_1, \ldots, q_j$. Note that $\alpha_k^0 = \beta_k$ and is 1-valent, whereas $\alpha_k^m = \gamma_k$ and is 0-valent.

What are the valences of the intermediate α_k^j executions? Somewhere in the sequence $\alpha_k^0, \alpha_k^1, \ldots, \alpha_k^{m-1}, \alpha_k^m$ there is a switch from 1-valent to 0-valent. Let j be such that α_k^j is 1-valent and α_k^{j+1} is 0-valent. Note that the only difference between α_k^j and α_k^{j+1} is that p_i sends to q_{j+1} in α_k^j but not in α_k^{j+1}.

The number of faulty processors in α_k^j (and also in α_k^{j+1}) is less than f, since at most $k - 1 < f - 1$ processors crash in α_{k-1} and p_i crashes in round k. Thus there is still one more processor that can crash without violating the bound f on the number of failures. Consider the admissible extensions δ_k^j and δ_k^{j+1} of α_k^j and α_k^{j+1}, respectively, in which q_{j+1} crashes at the beginning of round $k + 1$, without ever

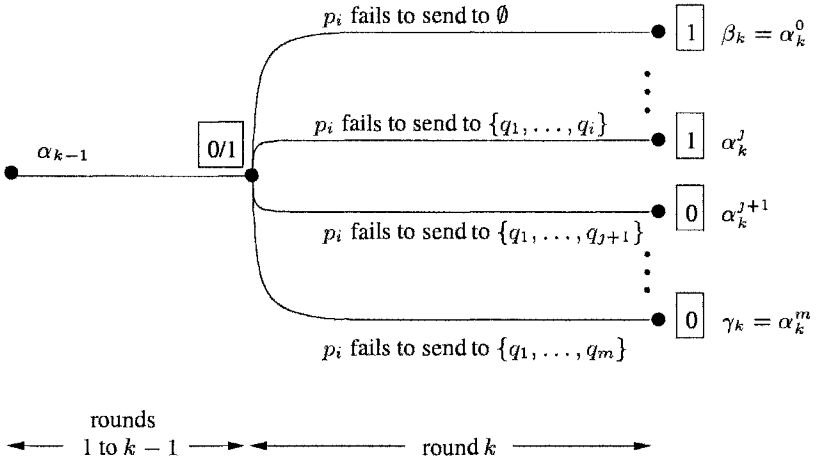
p_i fails to send to $\emptyset$
p_i fails to send to $\{q_1, \ldots, q_j\}$
p_i fails to send to $\{q_1, \ldots, q_{j+1}\}$
p_i fails to send to $\{q_1, \ldots, q_m\}$
α_{k-1}
0/1
1
1
0
0
$\beta_k = \alpha_k^0$
α_k^j
α_k^{j+1}
$\gamma_k = \alpha_k^m$
rounds 1 to $k-1$
round k

Fig. 5.3 Illustration for the proof that there is a k-round execution ending in a bivalent configuration (Lemma 5.5). Valences of configurations are indicated by values in boxes. Edge labels indicate to whom p_i fails to send.

getting a chance to reveal whether or not it received a message from p_i in round $k+1$, and no further processors crash. The two executions δ_k^j and δ_k^{j+1} are similar with respect to every nonfaulty processor, since the only difference between them is that p_i sends to q_{j+1} in δ_k^j but not in δ_k^{j+1}, yet q_{j+1} crashes before revealing this information. Thus α_k^j and α_k^{j+1} must have the same valence, which is a contradiction.

Thus there must exist a one-round extension of α_{k-1} with at most one additional crash that ends in a bivalent configuration. □

From the previous lemma, we have an $(f-1)$-round execution that ends in a bivalent configuration. The next lemma concerns round f — this round may not preserve bivalence, but we show that nonfaulty processors cannot determine yet what decision to make, and thus an additional round is necessary.

Lemma 5.6 *If α_{f-1} is an $(f-1)$-round execution of A that ends in a bivalent configuration, then there exists a one-round extension of α_{f-1} in which some nonfaulty processor has not decided.*

Proof. Let β_f be the one-round extension of α_{f-1} in which no failure occurs in round f. If β_f ends in a bivalent configuration, we are done. Suppose β_f ends in a univalent configuration, say 1-valent. Since the configuration at the end of α_{f-1} is bivalent, some other one-round extension of α results in a configuration that is either bivalent (in which case we are done) or 0-valent; call this execution γ_f. It must be that exactly one processor fails in round f of γ_f, that is, some processor p_i is faulty and fails to send a message to some nonfaulty processor p_j. The reason why p_j exists is that p_j cannot fail in round f, since p_i is the processor that fails in this round, and

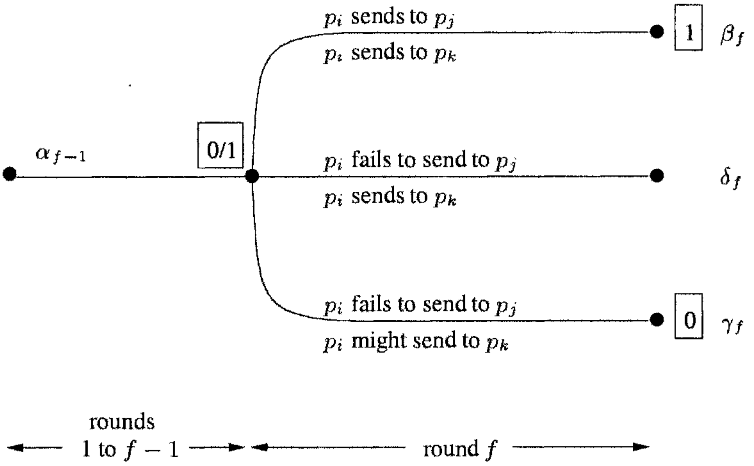

Fig. 5.4 Illustration for the proof that some processor is still undecided in round f (Lemma 5.6). Valences of configurations are indicated by values in boxes. Edge labels indicate to whom p_i fails or does not fail to send.

p_j cannot already have failed since otherwise there would be no observable difference between the executions. Consider a third one-round extension δ_f of α_{f-1} that is the same as γ_f except that p_i succeeds in sending to some nonfaulty processor p_k other than p_j; p_k must exist since $n \geq f+2$. (It is possible for δ_f to be the same as γ_f.) See Fig. 5.4.

The executions β_f and δ_f are similar with respect to p_k. Thus at the end of round f in δ_f, p_k is either undecided or has decided 1, since β_f is 1-valent. Similarly, the executions γ_f and δ_f are similar with respect to p_j. Thus at the end of round f in δ_f, p_j is either undecided or has decided 0, since γ_f is 0-valent. Since the algorithm satisfies the agreement property, it cannot be the case in δ_f that both p_j and p_k have decided. □

We now conclude the proof of Theorem 5.3. Lemmas 5.5 and 5.6 together imply the existence of an f-round execution in which some nonfaulty processor has not decided. In every admissible extension of this execution (for instance, the extension in which there are no further crashes), at least $f+1$ rounds are required for termination. □

5.2 SYNCHRONOUS SYSTEMS WITH BYZANTINE FAILURES

We now turn to study more severe, malicious failures, still in the context of synchronous systems. This model is often called the *Byzantine model*, because of the following metaphoric description of the consensus problem:

Several divisions of the Byzantine army are camped outside an enemy city. Each division is commanded by a general. The generals can communicate with each other only by reliable messengers. The generals should decide on a common plan of action, that is, they should decide whether to attack the city or not (cf. agreement), and if the generals are unanimous in their initial opinion, then that opinion should be the decision (cf. validity). The new wrinkle is that some of the generals may be traitors (that is why they are in the Byzantine army) and may try to prevent the loyal generals from agreeing. To do so, the traitors send conflicting messages to different generals, falsely report on what they heard from other generals, and even conspire and form a coalition.

5.2.1 Formal Model

We need to modify the definition of execution from Chapter 2 (for reliable synchronous message passing) to handle Byzantine processor failures. In an execution of an f-resilient Byzantine system, there exists a subset of at most f processors, the *faulty* processors.[2] In a computation step of a faulty processor, the new state of the processor and the contents of the messages sent are completely unconstrained. As in the reliable case, every processor takes a computation step in every round and every message sent is delivered in that round.

Thus a faulty processor can behave arbitrarily and even maliciously, for example, it can send different messages to different processors (or not send messages at all) when it is supposed to send the same message. The faulty processors can appear to coordinate with each other. In some situations, the recipient of a message from a faulty processor can detect that the sender is faulty, for instance, if the message is improperly formatted. Difficulties arise when the message received is plausible to the recipient, yet not correct. A faulty processor can also mimic the behavior of a crashed processor by failing to send any messages from some point onward.

5.2.2 The Consensus Problem Revisited

The definition of the consensus problem in the presence of Byzantine failures is the same as for crash failures and is repeated here. Each processor p_i has input and output state components, x_i and y_i; initially, x_i holds a value from some well-ordered set and y_i is undefined. Any assignment to y_i is irreversible. A solution to the consensus problem must guarantee the following:

Termination: In every admissible execution, y_i is eventually assigned a value, for every nonfaulty processor p_i.

Agreement: In every execution, if y_i and y_j are assigned, then $y_i = y_j$, for all nonfaulty processors p_i and p_j. That is, nonfaulty processors do not decide on conflicting values.

[2] In some of the literature, the upper bound on the number of Byzantine processors is denoted t, for *traitors*.

Validity: In every execution, if, for some value v, $x_i = v$ for all processors p_i, and if y_i is assigned for some nonfaulty processor p_i, then $y_i = v$. That is, if all the processors have the same input, then any value decided upon by a nonfaulty processor must be that common input.

For input sets whose size is larger than two, this validity condition is not equivalent to requiring that every nonfaulty decision value be the input of some processor, as Exercise 5.7 asks you to show. As in the crash case, no requirements are placed on the decisions of faulty processors.

We first show a lower bound on the ratio between faulty and nonfaulty processors. We then present two algorithms for reaching consensus in the presence of Byzantine failures. The first is relatively simple but uses exponential-size messages. The round complexity of this algorithm is $f + 1$ and matches the lower bound proved in Section 5.1.4. Recall that the $f + 1$ round lower bound was shown assuming crash failures. The same lower bound also holds in any system that is worse-behaved, including one with Byzantine failures; a Byzantine-faulty processor can act like a crash-faulty processor. The second algorithm is more complicated and doubles the number of rounds; however, it uses constant-size messages.

5.2.3 Lower Bound on the Ratio of Faulty Processors

In this section, we prove that if a third or more of the processors can be Byzantine, then consensus cannot be reached. We first show this result for the special case of a system with three processors, one of which might be Byzantine; the general result is derived by reduction to this special case.

Theorem 5.7 *In a system with three processors and one Byzantine processor, there is no algorithm that solves the consensus problem.*

Proof. Assume, by way of contradiction, that there is an algorithm for reaching consensus in a system with three processors, p_0, p_1, and p_2, connected by a complete communication graph. Let A be the local algorithm (state machine) for p_0, B the local algorithm for p_1, and C the local algorithm for p_2.

Consider a synchronous ring system with six processors in which p_0 and p_3 have A for their local algorithm, p_1 and p_4 have B for their local algorithm, and p_2 and p_5 have C for their local algorithm, as depicted in Figure 5.5(a). We cannot assume that such a system solves consensus, since the combination of A, B, and C only has to work correctly in a triangle. However, this system does have some particular, well-defined behavior, when each processor begins with an input value and there are no faulty processors.

The particular execution of the ring of interest is when the input values are 1 for p_0, p_1, and p_2 and 0 for p_3, p_4, and p_5 (see Fig. 5.5(a)). Call this execution β. This execution will be used to specify the behavior of faulty processors in some triangles.

Consider an execution α_1 of the algorithm in a triangle in which all processors start with input 1 and processor p_2 is faulty (see Fig. 5.5(b)). Furthermore, assume that processor p_2 is sending to p_0 the messages sent in β by p_5 (bottom left) to p_0

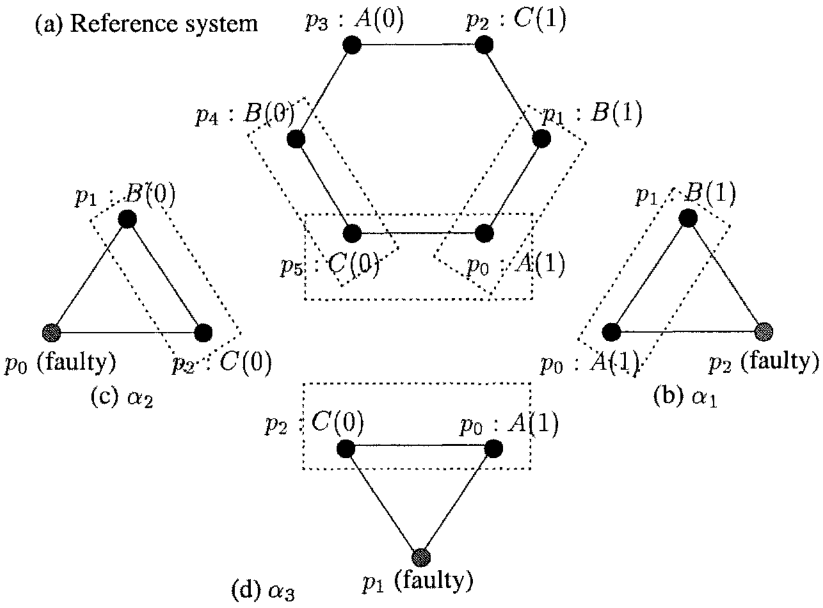

Fig. 5.5 Proof of Theorem 5.7.

and to p_1 the messages sent in β by p_2 (upper right) to p_1. By the validity condition, both p_0 and p_1 must decide 1 in α_1.

Now consider an execution α_2 of the algorithm in a triangle in which all processors start with input 0, and processor p_0 is faulty (Fig. 5.5(c)). Furthermore, assume that processor p_0 is sending to p_1 the messages sent in β by p_3 (top left) to p_4 and to p_2 the messages sent in β by p_0 (bottom right) to p_5. By the validity condition, both p_1 and p_2 must decide 0 in α_2.

Finally, consider an execution α_3 of the algorithm in a triangle where processor p_0 starts with input 1, processor p_2 starts with input 0, and processor p_1 is faulty (Fig. 5.5(d)). Furthermore, assume that processor p_1 is sending to p_2 the messages sent in β by p_4 (middle left) to p_5 and to p_0 the messages sent in β by p_1 (middle right) to p_0.

We now argue that $\alpha_1 \overset{p_0}{\sim} \alpha_3$. Since the messages sent by faulty processor p_2 are defined with reference to β, a simple induction on the round number verifies that p_0 has the same view in α_1 as it does in β and that p_1 has the same view in α_1 as it does in β. Similarly, induction on the round number verifies that p_0 has the same view in β as it does in α_3 and p_5 has the same view in β as p_2 does in α_3. Thus $\alpha_1 \overset{p_0}{\sim} \alpha_3$, and consequently, p_0 decides 1 in α_3.

But since $\alpha_2 \overset{p_2}{\sim} \alpha_3$ (cf. Exercise 5.10), p_2 decides 0 in α_3, violating the agreement condition, a contradiction. □

We prove the general case by reduction to the previous theorem.

Theorem 5.8 *In a system with n processors and f Byzantine processors, there is no algorithm that solves the consensus problem if $n \leq 3f$.*

Proof. Assume, by way of contradiction, that there exists an algorithm that reaches consensus in a system with n processors, f of which might be Byzantine. Partition the processors into three sets, P_0, P_1, and P_2, each containing at most $n/3$ processors. Consider now a system with three processors, p_0, p_1, and p_2. We now describe a consensus algorithm for this system, which can tolerate one Byzantine failure.

In the algorithm, p_0 simulates all the processors in P_0, p_1 simulates all the processors in P_1, and p_2 simulates all the processors in P_2. We leave the details of the simulation to the reader. If one processor is faulty in the three-processor system, then since $n/3 \leq f$, at most f processors are faulty in the simulated system with n processors. Therefore, the simulated algorithm must preserve the validity and agreement conditions in the simulated system, and hence also in the three-processor system.

Thus we have a consensus algorithm for a system with three processors that tolerates the failure of one processor. This contradicts Theorem 5.7. □

5.2.4 An Exponential Algorithm

In this section, we describe an algorithm for reaching consensus in the presence of Byzantine failures. The algorithm takes exactly $f + 1$ rounds, where f is the upper bound on the number of failures, and requires that $n \geq 3f + 1$. Thus the algorithm meets two of the lower bounds for consensus in the presence of Byzantine failure, on the number of rounds and resilience. However, it uses messages of exponential size.

The algorithm contains two stages. In the first stage, information is gathered by communication among the processors. In the second stage, each processor locally computes its decision value using the information collected in the previous stage.

It is convenient to describe the information maintained by each processor during the algorithm as a tree in which each path from the root to a leaf contains $f + 2$ nodes; thus the height of the tree is $f + 1$. We label nodes with sequences of processors' names in the following manner. The root is labeled with the empty sequence. Let v be an internal node in the tree labeled with the sequence $i_1, i_2, \ldots, i_r$; for every i between 0 and $n-1$ that is not in this sequence, v has one child labeled $i_1, i_2, \ldots, i_r, i$. (Fig. 5.6 contains an example for a system with $n = 4$ and $f = 1$; the shadings in the nodes at level 2 will be used in Fig. 5.7.) Note that no processor appears twice in the label of a node. A node labeled with the sequence π *corresponds* to processor p_i if π ends with i.

In the first stage of the algorithm, information is gathered and stored in the nodes of the tree. In the first round of the information gathering stage, each processor sends its initial value to all processors, including itself.[3] When a nonfaulty processor p_i

[3] Although processors do not actually have channels to themselves, they can "pretend" that they do.

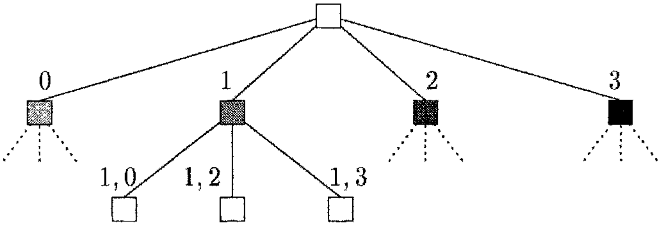

Fig. 5.6 The exponential information gathering tree; $n = 4, f = 1$.

receives a value x from processor p_j, it stores the received value at the node labeled j in its tree; a default value, $v_\perp$, is stored if x is not a legitimate value or if no value was received. In general, each processor broadcasts the rth level of its tree at the beginning of round r. When a processor receives a message from p_j with the value of the node labeled $i_1, \ldots, i_r$, it stores the value in the node labeled $i_1, \ldots, i_r, j$ in its tree. Figure 5.7 shows how the information received from p_2 in round 2, corresponding to the shaded nodes at level 2 in Figure 5.6, is stored at the nodes in level 3.

Intuitively, p_i stores in node $i_1, \ldots, i_r, j$ the value that "p_j says that p_{i_r} says that … that p_{i_1} said." Given a specific execution, we refer to this value as $tree_i(i_1, \ldots, i_r, j)$, omitting the subscript i when no confusion will arise.

Information gathering as described above continues for $f + 1$ rounds, until the entire tree has been filled in. At this point, the second stage of computing the decision value locally starts. Processor p_i computes the decision value by applying to each subtree a recursive data reduction function resolve. The value of the reduction function on p_i's subtree rooted at a node labeled with π is denoted resolve$_i(\pi)$, omitting the subscript i when no confusion will arise. The decision value is resolve$_i()$, that is, the result of applying the function to the root of the tree.

The function resolve is essentially a recursive majority vote and is defined for a node π as follows. If π is a leaf, then resolve(π) = *tree*(π); otherwise, resolve(π) is the majority value of resolve(π'), where π' ranges over all children of π ($v_\perp$ if no majority exists).

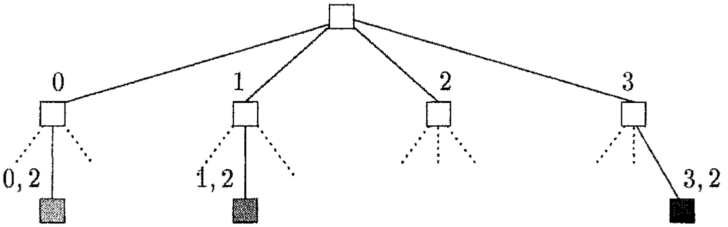

Fig. 5.7 How level 2 of p_2 is stored at level 3 of another processor.

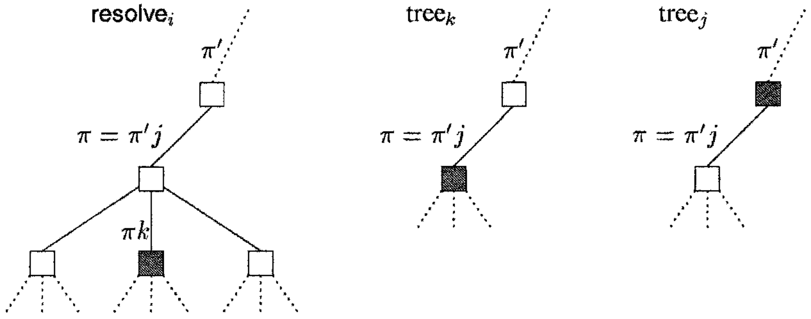

Fig. 5.8 Illustration for the proof of Lemma 5.9.

In summary, processor p_i gathers information for $f+1$ rounds, computes the reduced value using resolve$_i$ and decides on resolve$_i()$.

We now prove that the algorithm is correct. Fix an admissible execution of the algorithm. We first prove Lemma 5.9, which is useful in establishing the validity condition. It states that nonfaulty processor p_i's resolved value for what another nonfaulty process p_j reports for node π' equals the value that p_j has stored in its tree in node π'.

A key aspect of the proof is that, if a node π in p_i's tree corresponds to p_j, then the value stored in $tree_i(\pi)$ was received by p_i in a message from p_j.

Lemma 5.9 *For every tree node label π of the form $\pi' j$, where p_j is nonfaulty,* resolve$_i(\pi) = tree_j(\pi')$*, for every nonfaulty processor p_i.*

Proof. The proof is by induction on the height of the node π in the tree, starting from the leaves.

The basis of the induction is when π is a leaf. By definition, resolve$_i(\pi)$ equals $tree_i(\pi)$. Note that $tree_i(\pi)$ stores the value for π' that p_j sent to p_i in the last round. Since p_j is nonfaulty, this value is $tree_j(\pi')$.

For the inductive step, let π be an internal node. Note that π has depth at most f. Since the tree has $f+2$ levels, the root has degree n, and in every level of the tree the degree of nodes decreases by one, it follows that the degree of π is at least $n-f$. Since $n \geq 3f+1$, the degree of π is at least $2f+1$. Thus the majority of the children of π correspond to nonfaulty processors.

Let πk be some child of π that corresponds to a nonfaulty processor p_k (see Fig. 5.8). By the inductive hypothesis, resolve$_i(\pi k)$ equals $tree_k(\pi)$. Since p_j is nonfaulty, $tree_k(\pi)$ equals $tree_j(\pi')$, that is, p_j correctly reports to p_k the value that p_j has stored for π'.

Thus p_i resolves each child of π corresponding to a nonfaulty processor to $tree_j(\pi')$, and thus resolve$_i(\pi)$ equals the majority value, $tree_j(\pi')$. □

We can now show the validity condition. Suppose all nonfaulty processors start with the same input value, v. The decision of each nonfaulty processor p_i is resolve$_i()$, which is the majority of the resolved values for all children of the root. For each child j of the root where p_j is nonfaulty, Lemma 5.9 implies that $\mathsf{resolve}_i(j)$ equals $tree_j()$, which is p_j's input v. Since a majority of the children of the root correspond to nonfaulty processors, p_i decides v.

The next lemma is used to show the agreement condition. A node π is *common* in an execution if all nonfaulty processors compute the same reduced value for π, that is, $\mathsf{resolve}_i(\pi) = \mathsf{resolve}_j(\pi)$, for every pair of nonfaulty processors p_i and p_j. A subtree has a *common frontier* if there is a common node on every path from the root of the subtree to its leaves.

Lemma 5.10 *Let π be a node. If there is a common frontier in the subtree rooted at π, then π is common.*

Proof. The lemma is proved by induction on the height of π. The base case is when π is a leaf and it follows immediately.

For the inductive step, assume that π is the root of a subtree with height $k + 1$ and that the lemma holds for every node with height k. Assume in contradiction that π is not common. Since by hypothesis the subtree rooted at π has a common frontier, every subtree rooted at a child of π must have a common frontier. Since the children of π have height k, the inductive hypothesis implies that they are all common. Therefore, all processors resolve the same value for all the children of π and the lemma follows since the resolved value for π is the majority of the resolved values of its children. □

Note that the nodes on each path from a child of the root of the tree to a leaf correspond to different processors. Because the nodes on each such path correspond to $f+1$ different processors, at least one of them corresponds to a nonfaulty processor and hence is common, by Lemma 5.9. Therefore, the whole tree has a common frontier, which implies, by Lemma 5.10, that the root is common. The agreement condition now follows. Thus we have:

Theorem 5.11 *There exists an algorithm for n processors that solves the consensus problem in the presence of f Byzantine failures within $f + 1$ rounds using exponential size messages, if $n > 3f$.*

In each round, every processor sends a message to every processor. Therefore, the total message complexity of the algorithm is $n^2(f+1)$. Unfortunately, in each round, every processor broadcasts a whole level of its tree (the one that was filled in most recently) and thus the longest message contains $n(n-1)(n-2)\cdots(n-(f+1)) = \Theta(n^{f+2})$ values.

5.2.5 A Polynomial Algorithm

The following simple algorithm uses messages of constant size, takes $2(f+1)$ rounds, and assumes that $n > 4f$. It shows that it is possible to solve the consensus problem

Algorithm 16 A polynomial consensus algorithm in the presence of Byzantine failures: code for p_i, $0 \leq i \leq n-1$.

```
Initially pref[$i$] $= x$                       // initial preference for self is for own input
and pref[$j$] $= v_\perp$, for any $j \neq i$                       // default for others

round $2k-1$, $1 \leq k \leq f+1$:                       // first round of phase $k$
    send ⟨pref[$i$]⟩ to all processors
    receive ⟨$v_j$⟩ from $p_j$ and assign to pref[$j$], for all $0 \leq j \leq n-1$, $j \neq i$
    let maj be the majority value of pref[0],...,pref[$n-1$] ($v_\perp$ if none)
    let mult be the multiplicity of maj

round $2k$, $1 \leq k \leq f+1$:                       // second round of phase $k$
    if $i = k$ then send ⟨maj⟩ to all processors                       // king of this phase
    receive ⟨king-maj⟩ from $p_k$ ($v_\perp$ if none)
    if mult $> \frac{n}{2} + f$
      then pref[$i$] := maj
      else pref[$i$] := king-maj
    if $k = f+1$ then $y$ := pref[$i$]                       // decide
```

with constant-size messages, although with an increase in the number of rounds and a decrease in the resilience.

The algorithm contains $f+1$ phases, each taking two rounds. Each processor has a preferred decision (in short, *preference*) for each phase, initially its input value. At the first round of each phase, all processors send their preferences to each other. Let v_i^k be the majority value in the set of values received by processor p_i at the end of the first round of phase k. If there is no majority, then a default value, $v_\perp$, is used. In the second round of the phase, processor p_k, called the *king* of the phase, sends its majority value v_k^k to all processors. If p_i receives more than $n/2+f$ copies of v_i^k (in the first round of the phase) then it sets its preference for the next phase to be v_i^k; otherwise, it sets its preference to be the phase king's preference, v_k^k, received in the second round of the phase. After $f+1$ phases, the processor decides on its preference.

Each processor maintains a local array *pref* with n entries. The pseudocode appears in Algorithm 16.

The following lemmas are with respect to an arbitrary admissible execution of the algorithm. The first property to note is *persistence of agreement*:

Lemma 5.12 *If all nonfaulty processors prefer v at the beginning of phase k, then they all prefer v at the end of phase k, for all k, $1 \leq k \leq f+1$.*

Proof. Since all nonfaulty processors prefer v at the beginning of phase k, each processor receives at least $n-f$ copies of v (including its own) in the first round of phase k. Since $n > 4f$, $n-f > n/2+f$, which implies that all nonfaulty processors will prefer v at the end of phase k. □

This immediately implies the validity property: If all nonfaulty processors start with the same input v, they continue to prefer v throughout the phases (because the preference at the end of one phase is the preference at the beginning of the next); finally, they decide on v at the end of phase $f + 1$.

Agreement is achieved by the king breaking ties. Because each phase has a different king and there are $f + 1$ phases, then at least one phase has a nonfaulty king.

Lemma 5.13 *Let g be a phase whose king p_g is nonfaulty. Then all nonfaulty processors finish phase g with the same preference.*

Proof. Suppose that all nonfaulty processors use the majority value received from the king for their preference (Line 11). Since the king is nonfaulty, it sends the same message and thus all the nonfaulty preferences are the same.

Suppose that some nonfaulty processor, say p_i, uses its own majority value, say v, for its preference (Line 10). Thus p_i receives more than $n/2 + f$ messages for v in the first round of phase g. Consequently every processor, including the king p_g, receives more than $n/2$ messages for v in the first round of phase g and sets its majority value to be v. Thus, no matter whether it executes Line 10 or Line 11 to set its preference, every nonfaulty processor has v for its preference. □

Therefore, at phase $g + 1$ all processors have the same preference, and the persistence of agreement (Lemma 5.12) implies that they will decide on the same value at the end of the algorithm. This implies that the algorithm has the agreement property and solves the consensus problem.

Clearly, the algorithm requires $2(f + 1)$ rounds and messages contain one bit. Thus we have:

Theorem 5.14 *There exists an algorithm for n processors that solves the consensus problem in the presence of f Byzantine failures within $2(f+1)$ rounds using constant size messages, if $n > 4f$.*

5.3 IMPOSSIBILITY IN ASYNCHRONOUS SYSTEMS

We have seen that the consensus problem can be solved in synchronous systems in the presence of failures, both benign (crash) and severe (Byzantine). We now turn to asynchronous systems. We assume that the communication system is completely reliable and the only possible failures are caused by unreliable processors. We show that if the system is completely asynchronous, then there is no consensus algorithm even in the presence of a single processor failure. The result holds even if processors fail only by crashing. The asynchronous nature of the system is crucial for this impossibility proof.

This impossibility result holds both for shared memory systems, if only read/write registers are used, and for message-passing systems. We first present the proof for shared memory systems in the simpler case of an $(n - 1)$-resilient algorithm (also called a *wait-free* algorithm), where all but one of the n processors might fail. Then we use a simulation to deduce the same impossibility result for the harder case of shared memory systems with n processors only one of which might crash. Another simulation, of shared memory in message-passing systems, allows us to obtain the impossibility result for message-passing systems as well.

The only change to the formal model needed is to allow the possibility of processors crashing, in both shared memory and message-passing asynchronous systems. This is done for shared memory simply by changing the definition of admissible executions to require that all but f of the processors must take an infinite number of steps, where f is the resiliency of the system (number of failures to be tolerated). In addition, for message passing, the definition of admissible execution requires that all messages sent must be eventually delivered, except for messages sent by a faulty processor in its last step, which may or may not be delivered.

The precise problem statement is the same as for the synchronous model in Section 5.1.2; we emphasize that x_i and y_i are private state components of processor p_i, not shared variables. We concentrate on the case of trying to decide when the input set is simply $\{0, 1\}$.

5.3.1 Shared Memory—The Wait-Free Case

We first consider the relatively simple situation of $n > 1$ processors all but one of which might crash. That is, we show that there is no wait-free algorithm for consensus in the asynchronous case. We assume that the shared registers are single-writer but multi-reader. (Chapter 10 shows that multi-writer registers can be simulated with single-writer registers, and thus this impossibility result also holds for multi-writer registers.)

The proof proceeds by contradiction. We assume there is a wait-free algorithm and then create an admissible execution in which no processor decides. This proof relies on the notion of bivalence, which was first introduced in Section 5.1.4 for a specific class of executions (failure-sparse ones) in the synchronous model. We adapt the definition to the asynchronous model and generalize it for all admissible executions, as follows.

Throughout this section, we consider only configurations that are reachable from an initial configuration by a prefix of an admissible execution. The *valence* of configuration C is the set of all values that are decided upon, by any processor, in some configuration reachable from C. Here, the term "reachable" is with respect to any execution, and not just failure-sparse executions as it was in Section 5.1.4. In an asynchronous system, a faulty processor cannot be distinguished from a nonfaulty processor in a finite execution, and therefore, the definition of valence refers to the decision of any processor, not just the nonfaulty processors. Bivalence, univalence, 1-valence, and 0-valence are defined analogously to the definitions in Section 5.1.4, but again with regard to any execution, not just failure-sparse ones, and any processor, not just nonfaulty ones.

The proof constructs an infinite execution in which every configuration is bivalent and thus no processor can decide.

In this section, we are concerned with the similarity of *configurations* in a shared memory model, as opposed to the message-passing synchronous lower bound in Section 5.1.4, where we worked with similar *executions*. Here we review Definition 4.1, originally used to study mutual exclusion algorithms: Two configurations C_1 and C_2 are *similar* to processor p_i, denoted $C_1 \overset{p_i}{\sim} C_2$, if the values of all the shared variables and the state of p_i are the same in C_1 and C_2. Lemma 5.15 shows that if two univalent configurations are similar for a single processor, then they cannot have different valences.

Lemma 5.15 *Let C_1 and C_2 be two univalent configurations. If $C_1 \overset{p_i}{\sim} C_2$, for some processor p_i, then C_1 is v-valent if and only if C_2 is v-valent, for $v = 0, 1$.*

Proof. Suppose C_1 is v-valent. Consider an infinite execution from C_1 in which only p_i takes steps. Since the algorithm is supposed to be wait-free, this execution is admissible and eventually p_i must decide. Since C_1 is v-valent, p_i must eventually decide v. Apply the same schedule to C_2. Since $C_1 \overset{p_i}{\sim} C_2$ and only p_i takes steps, it follows that p_i decides v also in the execution from C_2. Thus C_2 is also v-valent. □

Lemma 5.16 states that some initial configuration is bivalent. Exercise 5.15 asks you to prove this fact, which is just a simpler version of Lemma 5.4.

Lemma 5.16 *There exists a bivalent initial configuration.*

Note that given a configuration C, there are n possible configurations that can immediately follow C: one for every possible processor to take the next step. If C is bivalent and the configuration resulting by letting p_i take a step from C is univalent, then p_i is said to be *critical* in C. We next prove that not all the processors can be critical in a bivalent configuration:

Lemma 5.17 *If C is a bivalent configuration, then at least one processor is not critical in C.*

Proof. Assume, by way of contradiction, that all processors are critical in C. Since C is bivalent, it follows that there exist two processors, p_i and p_j, such that $i(C)$ is 0-valent and $j(C)$ is 1-valent. The rest of the proof depends on the type of accesses performed by the processors in these steps.

If p_i and p_j access different registers or if both read the same register, then $i(j(C))$ is the same as $j(i(C))$, which implies that $i(C)$ and $j(C)$ cannot have different valences.

Since registers are single-writer, the only remaining case is when one processor writes to a shared register and the other processor reads from the same register. Without loss of generality, assume that p_i writes to R and p_j reads from R. Consider the configurations $i(C)$ and $i(j(C))$, that is, the configurations resulting when p_i takes a step from C and when p_j and then p_i take a step from C (see Fig. 5.9). Note that $i(j(C))$ is 1-valent and $i(C)$ is 0-valent. However, $i(j(C)) \overset{p_i}{\sim} i(C)$, which contradicts Lemma 5.15. □

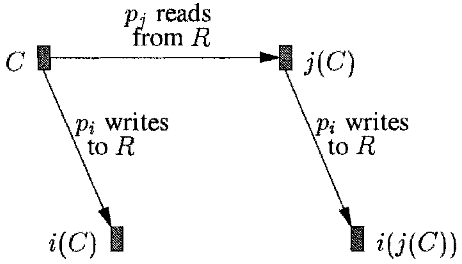

Fig. 5.9 Illustration for the proof of Lemma 5.17.

We now inductively create an admissible execution $C_0 i_1 C_1 i_2 \ldots$ in which the configurations remain bivalent forever. Let C_0 be the initial bivalent configuration provided by Lemma 5.16. Suppose the execution has been created up to bivalent configuration C_k. By Lemma 5.17, some processor is not critical in C_k; denote this processor by p_{i_k}. Then p_{i_k} can take a step without resulting in a univalent configuration. We apply the event i_k to C_k to obtain C_{k+1}, which is bivalent, and repeat the process again.

The execution constructed in this manner is admissible for the wait-free case, because at least one processor takes an infinite number of steps. It is possible that after some point, all the p_{i_k}'s are the same, meaning that $n - 1$ processors have failed. The strong assumption of wait freedom made our life easier in constructing this counterexample execution (cf. the complications in modifying this proof for the 1-resilient case in Exercise 5.18).

To summarize, we have constructed an admissible execution in which all the configurations are bivalent. Therefore, no processor ever decides, contradicting the termination property of the algorithm and implying:

Theorem 5.18 *There is no wait-free algorithm for solving the consensus problem in an asynchronous shared memory system with n processors.*

5.3.2 Shared Memory—The General Case

The impossibility proof of Section 5.3.1 assumed that the consensus algorithm is wait-free, that is, each processor must be able to decide even when all other processors fail. However, a stronger claim holds: There is no consensus algorithm even if only one processor may fail. This section proves this impossibility result. There exists a direct proof of this claim (see the chapter notes and Exercise 5.18); here we prove the more general result by reduction to the impossibility result of Section 5.3.1. Specifically, we assume, by way of contradiction, that there is a consensus algorithm for a system of $n > 2$ processors that tolerates the failure of one processor and show that there is a wait-free consensus algorithm for a system of two processors, that is, an algorithm that tolerates the failure of one processor. (Note that in the case of two processors, an algorithm that tolerates the failure of one processor is wait-free, and vice versa.) As we just proved that there can be no wait-free algorithm for any value of n, including $n = 2$, the assumed 1-resilient algorithm cannot exist.

5.3.2.1 Overview of Simulation The reduction works by having two processors *simulate* the code of the local algorithms (state machines) of the n processors. To explain the idea in more detail, let us denote the n *simulated* processors by $q_0, \ldots, q_{n-1}$ and, as usual, denote the two *simulating* processors by p_0 and p_1.

A simple approach to the simulation would be to let p_0 simulate half of the processors and let p_1 simulate the other half of the processors. Under this approach, however, a failure of a *single* simulating processor may lead to the failure of a *majority* of the simulated processors. Thus we would only be able to derive the impossibility of solving consensus when a majority of the processors may fail, and not when just a single processor may fail.

Instead, each of p_0 and p_1 goes through the codes of $q_0, \ldots, q_{n-1}$, in round-robin order, and tries to simulate their computation, one step at a time. Each simulating processor uses its input as the input for each simulated code. Once a decision is made by some simulated code, this decision is taken as the output by the simulating processor, which then stops the simulation.

It is possible that both p_0 and p_1 may try to simulate the same step, say, the kth, of the same simulated processor, q_j. To guarantee the consistency of the simulation, we require that p_0 and p_1 agree on each step of each simulated processor. Roughly and glossing over some details, this is done as follows: When a simulating processor simulates the kth step of q_j, it writes its suggestion for this step, then it checks to see whether the other processor has written a suggestion for the kth step of q_j. If the other processor has not yet written a suggestion, the first processor declares itself as the winner (by setting a flag to be 1), and its suggestion is used henceforth as the kth step for the simulated processor q_j. Otherwise, if the other processor has already written a suggestion, the first processor sets its flag to be 0. If both processors set their flags to 0, processors subsequently break the tie by using the suggestion of p_0 as the kth simulated step. (This is very similar to the asymmetric code for two-processor mutual exclusion, Algorithm 11.)

There are situations when it is not clear which processor wins, for example, if the flag of p_0 is 0 and the flag of p_1 is not yet set for the last simulated step of some processor q_j. In this case, we cannot know the result of this step until p_1 sets its flag. Thus the simulation of the code of q_j might be blocked if p_1 fails before writing its flag. Superficially, this seems to imply that the simulation algorithm is not wait-free. Yet, note that the reason we are blocked is that some simulating processor (in the above example, p_1) is in the middle of simulating a step of q_j. Clearly, this means it is not in the middle of simulating any step of any other simulated processor $q_{j'}$. As we shall show below, this means that the other processor (in the above example, p_0) can continue to simulate the other processors' codes on its own until it is able to decide. Thus the simulation of at most one processor can be stuck.

We now discuss the main details that were glossed over in the above description, namely, what a step is and what the suggestions made by simulating processors are. We make the following assumptions about the algorithm being simulated:

1. Each processor q_j can write to a single shared register, whose initial value is arbitrary

2. The code of each processor q_j consists of strictly alternating read steps and write steps, beginning with a read step

3. Each write step of each processor q_j writes q_j's current state into q_j's shared register

There is no loss of generality in making these assumptions, because any algorithm in which processors communicate by reading and writing single-writer multi-reader registers can be expressed in this form. (See Exercise 5.23.)

Thus q_j's computation is a sequence of *pairs* of read steps and write steps. Each pair can be viewed as a kind of "super-step" in which q_j reads the state of some other processor $q_{j'}$, changes its local state, based on its previous state and the value read, and writes its new state. Of course, the read and write are not done atomically—other steps of other processors can be interposed between them. The suggestions made by the simulating processors can be different and must be reconciled in the simulation. The state of the simulated processor can be read only after the suggestions are reconciled. Thus the suggestions are for the states of simulated processors (which equal the values of shared registers) at the end of the pairs, that is, after the write steps.

5.3.2.2 The Simulation The algorithm employs the following shared data structures. For each simulating processor p_i, $i = 0, 1$, for each simulated processor q_j, $j = 0, \ldots, n-1$, and for each integer $k \geq 0$, there are two registers, both written by p_i and read by p_{1-i}:

Suggest$[j, k, i]$: The state of simulated processor q_j at the end of the kth pair, as suggested by simulating processor p_i. The initial value is $\perp$.

Flag$[j, k, i]$: The competition flag of simulating processor p_i, for the kth pair of simulated processor q_j. The initial value is $\perp$.

In addition, each simulating processor maintains some local variables for bookkeeping during the simulation. The main local variable is an array *lastpair*, where *lastpair*$[j]$ is the number of pairs of steps by q_j that it has simulated. The meanings of the other local variables should be obvious from their usage in the code.

The pseudocode for the simulation appears in Algorithm 17. The function transition, given a simulated processor, the current state, and the value read, produces the next state of the simulated processor, according to its code.

5.3.2.3 Correctness Proof All the lemmas are with respect to an arbitrary admissible execution of the simulation algorithm. We start with some basic properties of the synchronization structure of the algorithm. The following simple lemma holds because, for each pair, a simulating processor first sets its own suggestion and then

Algorithm 17 Simulating n processors and one failure: code for p_i, $i = 0, 1$.

```
Initially $lastpair[j] = 0, 0 \le j \le n-1$

$j := 0$                                                    // start with $q_0$
while true do
    if computed($j$,$lastpair[j]$) then          // previous pair for $q_j$ is computed
        $k$, $lastpair[j] := lastpair[j] + 1$
        if $Flag[j, k, 1-i] \neq 1$ then            // other processor has not won
            $s :=$ get-state($j, k-1$)
            if $s$ is a decision state of $q_j$ then
                decide same and terminate
            $r :=$ simulated processor whose variable is
                to be read next according to $s$
            $v :=$ get-read($r$)
            $Suggest[j, k, i] :=$ transition($j, s, v$)          // new state for $q_j$
            if $Suggest[j, k, 1-i] = \perp$ then                    // $p_i$ has won
                $Flag[j, k, i] := 1$
            else $Flag[j, k, i] := 0$
    $j := (j + 1) \bmod n$                      // go to next simulated processor

function computed($j, k$)                   // has $k$th pair of $q_j$ been computed?
    if $k = 0$ then return true
    if $Flag[j, k, 0] = 1$ or $Flag[j, k, 1] = 1$ then return true
    if $Flag[j, k, 0] = 0$ and $Flag[j, k, 1] = 0$ then return true // need not reread
    return false

function get-state($j, \ell$)                     // return state of $q_j$ after $\ell$th pair
    if $\ell = 0$ then return initial state of $q_j$ with input equal to $p_i$'s input
    $w :=$ winner($j, \ell$)             // who won competition on $\ell$th pair of $q_j$?
    return $Suggest[j, \ell, w]$

function get-read($r$)                     // return current value of $q_r$'s variable
    $m := 1$
    while computed($r, m$) do $m := m + 1$
                        // $m - 1$ is largest numbered pair that is computed for $q_r$
    if $m - 1 = 0$ then return initial value of $q_r$'s variable
    return get-state($r, m - 1$)

function winner($j, k$)                // who won competition on $k$th pair of $q_j$?
    if $Flag[j, k, 1] = 1$ then return 1 else return 0
```

checks the other processor's suggestion. Therefore, at least one of them sees the other processor's suggestion and sets its flag to be 0.

Lemma 5.19 *For every simulated processor q_j and every $k \geq 1$, at most one of Flag$[j, k, 0]$ and Flag$[j, k, 1]$ equals 1 in every configuration.*

This implies that if one processor's flag is set to 1, then the other processor's flag will be set to 0, if it is set at all. Note, however, that it is possible that both processors will set their flags to 0, if they write their suggestions "together" and then read each other's. We say that k is a *computed pair* of q_j either if *Flag*$[j, k, 0]$ and *Flag*$[j, k, 1]$ are both set or one of them is not set and the other one equals 1. We define the *winner* of the kth computed pair of q_j to be the simulating processor p_i, $i = 0, 1$, that sets *Flag*$[j,k,i]$ to 1, if there is such a processor, and to be p_0 otherwise. By Lemma 5.19, the winner is well-defined. Note that the function winner is only called for computed pairs and it returns the id of the winner, according to this definition. Furthermore, the procedures get-state and get-read return the winner's suggestion.

There is a slight asymmetry between get-state and get-read for pair 0: get-state returns the initial state of the processor, which includes the input value, whereas get-read returns the initial value of the register, which does not include the input value.

Each processor p_i executes Lines 4 through 14 of the main code with particular values of j and k at most once; we will refer to the execution of these lines as p_i's simulation of q_j's kth pair.

Lemma 5.20 states that if one processor simulates a pair on its own (in the sense made precise by the lemma), then its flag will be set to 1. As a result, its suggestion for this pair will be taken subsequently.

Lemma 5.20 *For every simulated processor q_j, and every $k \geq 1$, if simulating processor p_i executes Line 12 of its simulation of q_j's kth pair before p_{1-i} executes Line 11 of its simulation of q_j's kth pair, then p_i sets Flag$[j, k, i]$ to 1.*

Thus, if one processor simulates on its own, it is able to decide on the simulated pair without waiting for the other simulating processor. We can already argue progress:

Lemma 5.21 *Suppose simulating processor p_i never fails or decides. Then the values of its lastpair$[j]$ variable grow without bound, for all j, $0 \leq j \leq n - 1$, except possibly one.*

Proof. Suppose there exists a simulated processor q_{j_0} such that simulating processor p_i's *lastpair*$[j_0]$ variable does not grow without bound. Since the variable never decreases, it reaches some value k_0 and never changes. Since p_i is stuck on pair k_0 for q_{j_0}, p_i writes 0 to *Flag*$[j_0, k_0, i]$ and never finds *Flag*$[j_0, k_0, 1 - i]$ set. This behavior is caused by the other simulating processor, p_{1-i}, crashing after writing *Suggest*$[j_0, k_0, 1 - i]$ in Line 11 and before writing *Flag*$[j_0, k_0, 1 - i]$ in Line 13 or 14.

Suppose, in contradiction, that p_i also fails to make progress on q_{j_1} for some $j_1 \neq j_0$. Let k_1 be the highest value reached by p_i's *lastpair*$[j_1]$ variable. It is not hard to see that k_1 is at least 1.

By the assumption that p_i is stuck on the k_1-th pair for q_{j_1}, computed(j_1, k_1) is never true. Thus neither *Flag*$[j_1, k_1, i]$ nor *Flag*$[j_1, k_1, 1-i]$ is ever set to 1. As a result, p_i executes Lines 6 through 14 of its simulation of q_{j_1}'s k_1-th pair. By Lemma 5.20, it must be that p_i executes Line 12 of its simulation of q_{j_1}'s k_1-th pair after p_{1-i} executes Line 11 of its simulation of q_{j_1}'s k_1-th pair, or else p_i would set its flag to 1. Thus p_i finds the other processor's suggestion already set and sets its flag to 0. Since computed(j_1, k_1) is never true, it must be that p_{1-i} never sets its flag, that is, it fails after Line 11 but before Line 13 or 14 of its simulation of q_{j_1}'s k_1-th pair. But this contradicts the fact that p_{1-i} fails during the simulation of q_{j_0}, not q_{j_1}. □

The above lemma guarantees that if one simulating processor does not halt, then it makes progress through the simulated codes of at least $n-1$ processors. Yet this does not necessarily mean that the processor will eventually decide correctly, or even decide at all. This will follow only if we show that the codes are simulated correctly. To prove this, we explicitly construct, for each admissible execution α of the simulation, a corresponding admissible execution β of $q_0, \ldots, q_{n-1}$, in which the same state transitions are made and at most one (simulated) processor fails. Because the algorithm of $q_0, \ldots, q_{n-1}$ is assumed to solve the consensus problem in the presence of one fault, it follows that the nonfaulty simulated processors eventually decide correctly in β and, therefore, the nonfaulty simulating processors eventually decide correctly in α.

For every simulated processor q_j, and every $k \leq 1$, we first identify two points in α: one for the read done in the kth pair from the register of some other processor and another for the write to q_j's register in the kth pair. Note that because the simulated algorithm is for the read/write model, these points can be separate. The *read point* is when the winner of the kth pair of q_j returns from the last call to computed in Line 27 of get-read. This is the call that returns false, based on the values of the two flags that are read. The *write point* is when the winner of the kth pair of q_j sets its flag to 1, or, if neither flag is ever set to 1, when the second simulating processor writes 0 to its flag.

Strictly speaking, the read point specified above is not well-defined, because the execution of computed does two reads and thus does not correspond to a single event in α. Note, however, that one of these flags is the winner's own flag and, therefore, this read need not be from the shared memory, but can instead be done from a copy in the local memory. Therefore, the execution of computed translates into a single shared memory operation, and the read point is well-defined.[4]

The next lemma shows that the values returned by get-read in α are consistent with the read and write points defined.

Lemma 5.22 *Let v be the value suggested by the winner of q_j's kth pair, that is, v is the value written to Suggest$[j, k, i]$, where $i =$* winner(j, k). *Then, in α, any read*

[4] Another solution is to use atomic snapshots, which will be defined in Chapter 10.

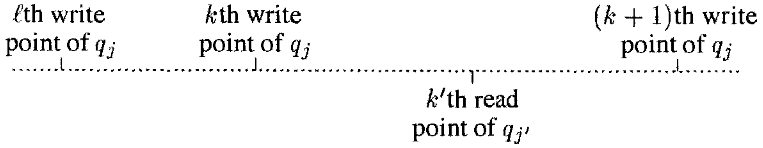
ℓth write point of q_j
kth write point of q_j
$(k+1)$th write point of q_j
k'th read point of $q_{j'}$

Fig. 5.10 Illustration for the proof of Lemma 5.22, $\ell \leq k$.

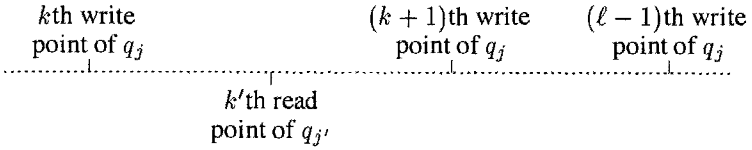
kth write point of q_j
$(k+1)$th write point of q_j
$(\ell-1)$th write point of q_j
k'th read point of $q_{j'}$

Fig. 5.11 Illustration for the proof of Lemma 5.22, $\ell > k+1$.

from the register of q_j whose read point is between the kth write point of q_j and the $(k+1)$st write point of q_j (if it exists) returns v.

Proof. Consider some pair, say the k'-th, of $q_{j'}$ that reads from the register of q_j, such that its read point is between the kth write point of q_j and the next write point of q_j. Without loss of generality, assume p_0 is the winner for this pair of $q_{j'}$ and let ℓ be the value of m when p_0 finishes Line 27 of procedure get-read. We argue that $\ell = k+1$, which proves the lemma.

If $\ell \leq k$ then, since the write point of the kth pair of q_j is before the read point of the k'th pair of $q_{j'}$, the write point of the ℓth pair of q_j is also before the read point of the k'th pair of $q_{j'}$ (see Fig. 5.10). Therefore, either the winner of the ℓth pair of q_j has set its flag to 1 or both simulating processors wrote 0 to their flags, before the read point of the k'th pair of $q_{j'}$. Thus when p_0 checks computed(j, ℓ) in the while loop of get-read, it continues the loop beyond ℓ, a contradiction.

On the other hand, if $\ell > k+1$, then the write point for the $(\ell-1)$th pair of q_j is after the read point of the k'th pair of $q_{j'}$ (see Fig. 5.11). Therefore, at the read point of the k'th pair of $q_{j'}$, the winner of the ℓth pair of q_j has not written 1 to its flag, and one of the processors has not written at all to its flag. Therefore, in Line 27 of get-read, p_0 finds computed($j, k+1$) false and exits the while loop at $k+1$, which is less than ℓ, a contradiction. □

We now construct an execution β of $q_0, \ldots, q_{n-1}$ based on execution α; we will show a precise correspondence between the two executions that will allow us to deduce the desired impossibility.

Consider the sequence of read and write points, for all simulated processors, in α. (The occurrence of the points forms a sequence because each point is an atomic occurrence in α.) Let σ be the sequence of simulated processor indices corresponding to the read and write points. Define an initial configuration C_0, in which the input value of each q_i is the input value of the simulating processor that is the winner for the first pair of q_i. If there is no winner for the first pair of q_i, then use p_0's input as the input value of q_i in C_0. Let β be the execution of $q_0, \ldots, q_{n-1}$ obtained by starting with C_0 and applying computation events in the order specified by σ. In other words, we let the simulated processors take steps in the order of their read and write points in α.

Lemma 5.23 shows that the values suggested by the winners for the pairs in α are consistent with the states and register values in β.

Lemma 5.23 *Let q_j be any processor and k be such that q_j executes at least $k > 0$ pairs in β. Then in α,*
(a) eventually computed(j, k) *is true, and, after that point,*
(b) the value of Suggest$[j, k, w]$, where w is winner(j, k), *is equal to the value of q_j's state (and shared register) after its kth pair in β.*

Proof. (a) By the construction of β from α, if q_j executes at least k pairs in β, it must be that computed(j, k) is set to true.

(b) We will prove this by induction on the prefixes of α.

For the basis, we consider the initial configuration of α. Since every processor has completed 0 pairs at this point, the lemma is vacuously true.

Suppose the lemma is true for prefix α' of α. Let π be the next event in α following α'. If π does not cause any additional pair to be computed, then the lemma remains true.

Suppose π causes computed(j, k) to become true. Let i be winner(j, k).

First, we show that p_i's execution of get-state$(j, k-1)$ returns the correct value. If $k = 1$ (this is the first pair by q_j), then get-state$(j, k-1)$ returns the initial state of q_j, with input equal to p_i's input. If $k > 1$, then get-state$(j, k-1)$ returns *Suggest*$[j, k-1, w]$, where w is winner$(j, k-1)$. By the inductive hypothesis, *Suggest*$[j, k-1, w]$ equals the value of q_j's state (and shared register) after its $(k-1)$-st pair in β. Let s be the value returned by get-state$(j, k-1)$.

Suppose the read step of q_j's kth pair involves reading the register of q_r and at the time of this read, q_r has performed h pairs (and thus h writes).

We now show that p_i's execution of get-read(r) returns the correct value. If $h = 0$, then get-read(r) returns the initial value of q_r's variable. If $h > 0$, then get-read(r) returns *Suggest*$[r, h, w']$, where w' is winner(r, h). By the inductive hypothesis, *Suggest*$[r, h, w']$, equals the value of q_r's state (and shared register) after its hth pair in β. By construction of β, the read point of this execution of get-read is between the hth and $(h+1)$st write points of q_r. By Lemma 5.22, the value read is correct. Let v be the value returned by get-read(r).

Thus the winning suggestion for q_j's kth pair is transition(j, s, v), which is the value of q_j's state after its kth pair in β. □

Exercise 5.24 asks you to put together the pieces shown by Lemma 5.21 and Lemma 5.23 in order to prove that Algorithm 17 correctly simulates an n-processor consensus algorithm with two processors. Consequently, if there is a 1-resilient consensus algorithm for n processors, then there is a 1-resilient consensus algorithm for two processors. But a 1-resilient consensus algorithm for two processors is wait-free, and Theorem 5.18 states that no such algorithm can exist. Thus we have proved:

Theorem 5.24 *There is no consensus algorithm for a read/write asynchronous shared memory system that can tolerate even a single crash failure.*

5.3.3 Message Passing

Finally, we extend the result of Section 5.3.2 to message-passing systems. Again, this is done by simulation; that is, we show how to simulate a message-passing algorithm by a shared memory algorithm. Therefore, if there is a message-passing algorithm for consensus there would be a shared memory algorithm for consensus, which is impossible (Theorem 5.24).

The simulation is simple: For each ordered pair of processors we have a separate single-writer single-reader register. The "sender" writes every new message it wishes to send in this register by appending the new message to the prior contents, and the "receiver" polls this register at every step to see whether the sender has sent anything new. This can be very easily done, if we assume that registers can hold an infinite number of values.

Because the receiver needs to check whether a message was sent by a number of senders, it has to poll a number of registers (one for each sender). However, in each computation step, the receiver can read only one register. Therefore, the reader should read the registers in a round-robin manner, checking each register only once every number of steps. This scheme introduces some delay, because a message is not necessarily read by the reader immediately after it is written by the sender. However, this delay causes no problems, because the message-passing algorithm is asynchronous and can withstand arbitrary message delays. As a result:

Theorem 5.25 *There is no algorithm for solving the consensus problem in an asynchronous message-passing system with n processors, one of which may fail by crashing.*

Exercises

5.1 Show that for two-element input sets, the validity condition given in Section 5.1.2 is equivalent to requiring that every nonfaulty decision be the input of some processor.

5.2 Consider a synchronous system in which processors fail by *clean* crashes, that is, in a round, a processor either sends all its messages or none. Design an algorithm that solves the consensus problem in one round.

5.3 (a) Modify Algorithm 15 to achieve consensus within f rounds, in the case $f = n - 1$.

Algorithm 18 k-set consensus algorithm in the presence of crash failures: code for processor p_i, $0 \leq i \leq n-1$.

```
Initially $V = \{x\}$

round $r$, $1 \leq r \leq \frac{f}{k} + 1$:                    // assume that $k$ divides $f$
    send $V$ to all processors
    receive $S_j$ from $p_j$, $0 \leq j \leq n-1, j \neq i$
    $V := V \cup \bigcup_{j=0}^{n-1} S_j$
    if $r = f/k + 1$ then $y := \min(V)$                    // decide
```

(b) Show that f is a lower bound on the number of rounds required in this case.

5.4 Design a consensus algorithm for crash failures with the following *early stopping* property: If f' processors fail in an execution, then the algorithm terminates within $O(f')$ rounds.

Hint: Processors need not decide in the same round.

5.5 Define the *k-set consensus* problem as follows. Each processor starts with some arbitrary integer value x_i and should output an integer value y_i such that:

Validity: $y_i \in \{x_0, \ldots, x_{n-1}\}$, and

k-Agreement: the number of different output values is at most k.

Show that Algorithm 18 solves the k-set consensus problem in the presence of f crash failures, for any $f < n$. The algorithm is similar to Algorithm 15 (for consensus in the presence of crash failures) and is based on collecting information.

What is the message complexity of the algorithm?

5.6 Present a synchronous algorithm for solving the k-set consensus problem in the presence of $f = n - 1$ crash failures using an algorithm for consensus as a black box. Using Algorithm 15 as the black box, the round complexity of the algorithm should be $(\frac{n}{k} + 1)$, and its message complexity should be $O(\frac{n^2}{k}|V|)$, where $|V|$ is the number of possible input values. For simplicity assume that k divides n.

5.7 Show that, if the input set has more than two elements, the validity condition given in Section 5.2.2 is not equivalent to requiring that every nonfaulty decision be the input of some processor. In particular, design an algorithm that satisfies the validity condition of Section 5.2.2 but does not guarantee that every nonfaulty decision is the input of some processor.

Hint: Consider the exponential message and phase king algorithms when the size of the input set is larger than 2.

5.8 Consider the exponential message consensus algorithm described in Section 5.2.4. By the result of Section 5.2.3, the algorithm does not work correctly if $n = 6$ and $f = 2$. Construct an execution for this system in which the algorithm violates the conditions of the consensus problem.

5.9 Repeat Exercise 5.8 for the polynomial message algorithm of Section 5.2.5.

5.10 Prove that $\alpha_2 \overset{p_2}{\sim} \alpha_3$ in the proof of Theorem 5.7.

5.11 Modify the exponential information gathering algorithm in Section 5.2.4 to reduce the number of messages to be $O(f^3 + fn)$.

5.12 Show that to satisfy the stronger validity condition (every nonfaulty decision is some nonfaulty input) for Byzantine failures, n must be greater than $\max(3, m) \cdot f$, where m is the size of the input set.

5.13 Assuming n is sufficiently large, modify the exponential message algorithm of Section 5.2.4 to satisfy the stronger validity condition of Exercise 5.12.

5.14 Assuming n is sufficiently large, modify the polynomial message algorithm of Section 5.2.5 to satisfy the stronger validity condition of Exercise 5.12.

5.15 Prove Lemma 5.16. That is, assume there is a wait-free consensus algorithm for the asynchronous shared memory system and prove that it has a bivalent initial configuration.

5.16 Consider a variation of the consensus problem in which the validity condition is the following: There must be at least one admissible execution with decision value 0, and there must be at least one admissible execution with decision value 1. Prove that there is no wait-free algorithm for this problem in an asynchronous system.

Hint: Modify the the proof of the existence of a bivalent initial configuration (Lemma 5.16).

5.17 In the *transaction commit* problem for distributed databases, each of n processors forms an independent opinion whether to commit or abort a distributed transaction. The processors must come to a consistent decision such that if even one processor's opinion is to abort, then the transaction is aborted, and if all processors' opinions are to commit, then the transaction is committed. Is this problem solvable in an asynchronous system subject to crash failures? Why or why not?

5.18 This exercise guides you through a direct proof of the impossibility of 1-resilient consensus for shared memory. Assume A is a 1-resilient consensus algorithm for n processors in shared memory.

(a) Prove that there is a bivalent initial configuration of A.

(b) Let D be a bivalent configuration of A and p_i be any processor. Using the outline given below, prove that there exists a schedule σ ending with the event i such that $\sigma(D)$ is bivalent.

Outline: Suppose, in contradiction, that there is no such schedule. Then every schedule that ends in i, when started at D, leads to a univalent configuration. Without loss of generality, assume that $i(D)$ is 0-valent.

(b.1) Show that there exists a finite schedule α, in which p_i does not take a step, and a processor p_j other than p_i, such that $i(\alpha(D))$ is 0-valent and $i(j(\alpha(D)))$ is 1-valent.

(b.2) Let $D_0 = \alpha(D)$ and $D_1 = j(\alpha(D))$. Consider the possible actions being performed by p_i and p_j in taking a step from D_0 (e.g., reading or writing) and show that a contradiction is obtained in each case.

(c) Combine (a) and (b) above to show there exists an admissible execution of A that does not satisfy the termination condition.

5.19 Consider an asynchronous shared memory system in which there are only test&set registers (as defined in Chapter 4) and two processors. Show that it is possible to solve consensus in this system even if one processor can crash.

5.20 Show that the consensus problem cannot be solved in an asynchronous system with only test&set registers and three processors, if two processors may fail by crashing. The proof may follow Section 5.3.1:

1. Define $C \overset{\neg\{p_i,p_j\}}{\sim} C'$ to mean that C is similar to C' for all processors but p_i and p_j.
2. Argue why Lemma 5.16 holds in this case as well.
3. Prove Lemma 5.17 for this model. (This is where most of the work is.)

5.21 Show that consensus cannot be solved in an asynchronous shared memory system with only test&set registers, with $n > 2$ processors, two of which may fail by crashing.

5.22 Can consensus be solved in an asynchronous shared memory system with $n > 2$ processors, two of which may fail by crashing, if we allow read/write operations, in addition to test&set operations?

5.23 Argue why the restricted form of the algorithms assumed in Section 5.3.2 does not lose any generality.

5.24 Prove that Algorithm 17 correctly simulates an n-processor consensus algorithm with two processors.

5.25 Modify Algorithm 17 (and its correctness proof) so that get-read skips steps that are known to be computed (based on *lastpair*).

5.26 An alternative version of the consensus problem requires that the input value of one distinguished processor (called the *general*) be distributed to all the other processors (called the *lieutenants*); this problem is known as *single-source consensus*. In more detail, the conditions to be satisfied are:

Termination: Every nonfaulty lieutenant must eventually decide

Agreement: All the nonfaulty lieutenants must have the same decision

Validity: If the general is nonfaulty, then the common decision value is the general's input.

The difference is in the validity condition: note that if the general is faulty, then the nonfaulty processors need not decide on the general's input but they must still agree with each other. Consider the synchronous message-passing model subject to Byzantine faults. Show how to transform a solution to the consensus problem into a solution to the general's problem and vice versa. What are the message and round overheads of your transformations?

Chapter Notes

The consensus problem was introduced by Lamport, Pease, and Shostak in two papers [163, 207]. The problem was originally defined for Byzantine failures. The simple algorithm for solving consensus in the presence of crash failures presented in Section 5.1.3 is based on an algorithm of Dolev and Strong that uses authentication to handle more severe failures [99]. The lower bound on the number of rounds needed for solving consensus was originally proved by Fischer and Lynch [107] for Byzantine failures and was later extended to crash failures by Dolev and Strong [99]. Subsequent work simplified and strengthened the lower bound [101, 185, 193]. Our presentation is based on that by Aguilera and Toueg [7].

After considering crash failures, we turned to the Byzantine failure model; this is a good model for human malfeasance. It is a worst-case assumption and is also applicable to machine errors—it covers the situation when a seemingly less malicious fault happens at just the wrong time for the software, causing the most damage. If a system designer is not sure exactly how errors will be manifested, a conservative assumption is that they will be Byzantine.

The $3f + 1$ lower bound on the number of faulty processors as well as the simple exponential algorithm were first proved in [207]. Our presentation follows later formulations of these results by Fischer, Lynch, and Merritt [109] for the $3f + 1$ lower bound, and by Bar-Noy, Dolev, Dwork, and Strong [44] for the exponential algorithm of Section 5.2.4. We also presented a consensus algorithm with constant message size (Algorithm 16); this algorithm is due to Berman and Garay [50]. Garay and Moses present a consensus algorithm tolerating Byzantine failures; the algorithm sends messages with polynomial size, works in $f + 1$ rounds, and requires only that $n > 3f$ [119].

There is an efficient reduction from the consensus problem with a general to ordinary consensus by Turpin and Coan [255].

The consensus algorithms we have presented assume that all processors can directly exchange messages with each other; this means that the topology of the communication graph is a clique. For other topologies, Dolev [91] has shown that a necessary and sufficient condition for the existence of a consensus algorithm tolerating Byzantine failures is that the connectivity of the graph is at least $2f + 1$. Fischer, Lynch, and Merritt present an alternative proof for the necessity of this condition, using an argument similar to the $3f + 1$ lower bound on the number of faulty processors [109].

Exercises 5.12 through 5.14 were suggested by Neiger [196].

The impossibility of achieving consensus in an asynchronous system was first proved in a breakthrough paper by Fischer, Lynch, and Paterson [110]. Their proof dealt only with message-passing systems. Exercise 5.18 walks through their proof, adapted for shared memory. Later, the impossibility result was extended to the shared memory model by Loui and Abu-Amara [173] and (implicitly) by Dolev, Dwork, and Stockmeyer [92]. Loui and Abu-Amara [173] provide a complete direct proof of the general impossibility result for shared memory systems. Special cases of this result were also proved by Chor, Israeli, and Li [81]. Loui and Abu-Amara also studied the possibility of solving consensus by using test&set operations (see Exercises 5.19 through 5.22).

We have chosen to prove the impossibility result by first concentrating on the shared memory wait-free case and then extending it to the other cases, a single failure and message-passing systems, by reduction. Not only is the shared memory wait-free case simpler, but this also fits better with our aim of unifying models of distributed computation by showing simulations (explored in more depth in Part II of the book).

The impossibility proof for the shared memory wait-free case follows Herlihy [134], who uses the consensus problem to study the "power" of various object types, as shall be seen in Chapter 15. The simulation result for 1-resiliency is based on the algorithm of Borowsky and Gafni [58].

The simulation depends on the ability to map the inputs of the simulating processors into inputs of the simulated algorithm and then to map back the outputs of the simulated algorithm to outputs of the simulating processors. This mapping is trivial for the consensus problem but is not necessarily so for other problems; for more discussion of this issue and another description of the simulation, see the work of Borowsky, Gafni, Lynch, and Rajsbaum [59].