This Page Intentionally Left Blank

Chapter 12

Consensus

In this chapter, we introduce another complication into our study of the asynchronous shared memory model: the possibility of failures. We only consider faulty processes, not faulty memory. In fact, we only consider the simplest type of process failure: *stopping failure*, whereby a process just stops without warning.

The problem we study in this chapter is one of *consensus*. We have already considered consensus problems extensively in the setting of synchronous message-passing systems, in Chapters 5, 6, and 7. For the case of process failures, we have shown that basic consensus problems are solvable, not only for stopping failures, but also for less well-behaved Byzantine failures. However, we gave several results showing that the costs of solutions, measured in terms of the number of processes and the amount of time required, are necessarily large.

Perhaps surprisingly, the situation turns out to be very different in the asynchronous setting, at least for read/write shared memory. Namely, we present a fundamental impossibility result, saying that a basic consensus problem *cannot be solved at all* in the asynchronous read/write shared memory setting, even if it is known that at most one process will fail. The same result holds, as you will see in Chapters 17 and 21, for the asynchronous network setting, and the reasons are essentially the same.

The impossibility of consensus is considered to be one of the most fundamental results of the theory of distributed computing. It has practical implications for any distributed application in which some type of agreement is required. For example, processes in a database system may need to agree on whether a transaction commits or aborts. Processes in a communication system may need to agree on whether or not a message has been received. Processes in a control system may need to agree on whether or not a particular other process is faulty. Then the impossibility result implies that there is no purely asynchronous algorithm that reaches the needed agreement and tolerates any failures at all.

This means that, in practice, designers must go outside the asynchronous model in order to solve such problems, for example, relying on timing information or being willing to settle for only probabilistic correctness.

12.1 The Problem

We define a particular consensus problem in the shared memory setting. Our presentation is informal, but it can be formalized in terms of the model defined in Chapter 9. Chapter 10 contains a similar informal presentation for the mutual exclusion problem, plus some guidelines showing how it can be formalized. You may find it useful to skim Sections 10.1 and 10.2 now.

The architecture we use here is essentially the same one we used in Chapters 9–11, with processes interacting with the environment via ports and communicating with each other via shared variables. See Figure 10.1, for example. We assume that $n \geq 2$, where n is the number of ports. The entire assembly of processes and variables is modelled as a single I/O automaton. We model the users as automata U_i also, as we did in Chapters 10 and 11. See Example 9.2.1 for one collection of users for the agreement problem. We assume a fixed value set V, $|V| \geq 2$, for the inputs and decisions.

This time, we assume that the external interface of each user U_i consists of output actions $init(v)_i$, where $v \in V$ is an input value for the shared memory system, and input actions $decide(v)_i$ inputs, where $v \in V$ is a decision value. The external interface of the shared memory system includes all the input actions $init(v)_i$, where $v \in V$ is an input value and i is a port name (i.e., process index), and all the output actions $decide(v)_i$, where $v \in V$ is a decision value and i is a port name. Thus, we are assuming that the inputs for the problem arrive from the users in input actions. (Note that most of the research papers in this area assume that the initial values appear in designated variables in the initial process states, while decisions are written into designated state variables. The formulation we use is more consistent with the style we are using elsewhere in the book.) Each user automaton must satisfy one restriction: it can only perform at most one $init_i$ event in an execution; that is, we assume that each process receives at most one input.

It is easy to formalize all of this in terms of I/O automata, as in Chapters 10 and 11. We assume in this chapter that there is exactly one task per process; in light of Exercise 8.8, this is not a significant restriction.

We assume that the processes are subject to *stopping failures*, by which we mean that they might simply stop without warning. Formally, we model this by including special $stop_i$ input actions, one for each process, in the external interface (external signature) of the shared memory system. A $stop_i$ event has

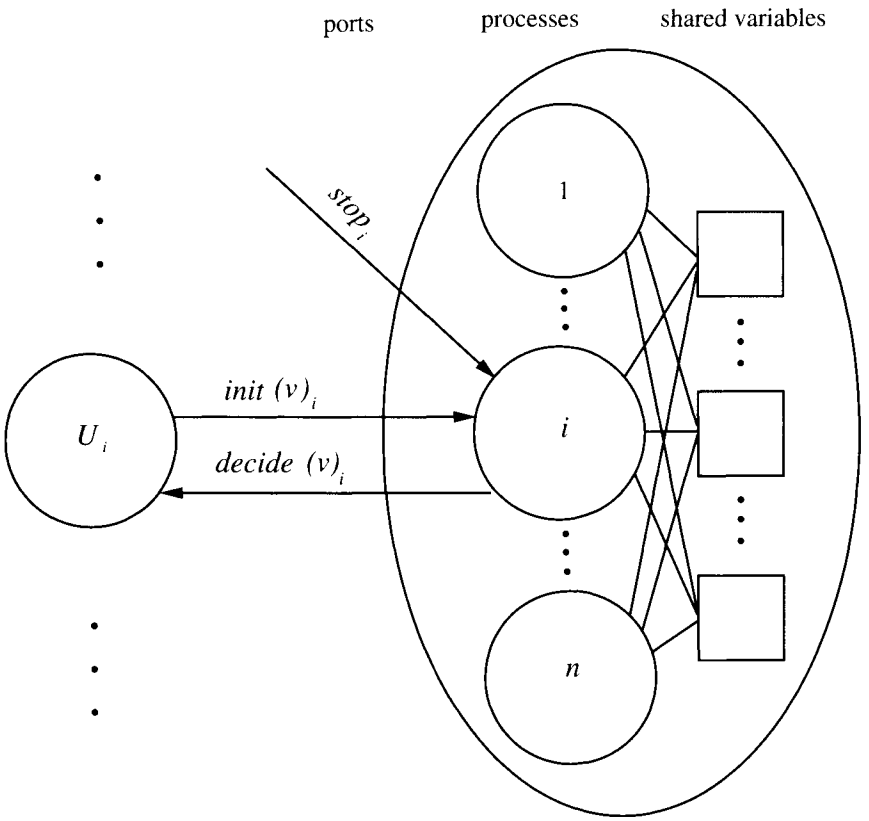

Figure 12.1: Shared memory system for the agreement problem.

the effect of disabling any future locally controlled actions of process i. The $stop_i$ actions are not considered to be part of the external interfaces of the user automata; they just arrive from some unspecified external environment (see Section 9.6). The complete architecture is depicted in Figure 12.1. With this method of modelling failures, and according to the formal definitions in Chapter 8, the *fair executions* of the system are those in which each process that does not fail, as well as each user task, gets infinitely many opportunities to perform locally controlled steps. We say that an execution of the system is *failure-free* if it contains no *stop* events.

We say that a sequence of $init_i$ and $decide_i$ actions is *well-formed* for user i, provided that it is some prefix of a sequence of the form $init(v)_i$, $decide(w)_i$ (i.e., the empty sequence, just an $init(v)_i$, or a two-action sequence $init(v)_i$, $decide(w)_i$). In particular, it does not contain repeated inputs at port i, nor repeated decisions at port i, nor does it contain any decision without a preceding input. Our as-

sumptions about the user automata imply that each U_i preserves well-formedness (according to the definition of "preserves" in Section 8.5.4).

We require the following properties of any execution, fair or not, of the combined system.

Well-formedness: For any i, the interactions between U_i and the system are well-formed for i.

Agreement: All decision values are identical.

Validity: If all *init* actions that occur contain the same value v, then v is the only possible decision value.

Notice that the agreement and validity conditions are analogous to the corresponding conditions in Section 6.1, for the stopping agreement problem in the synchronous model; the main difference is in the input/output conventions.

We also need some kind of termination condition. The most basic requirement is the following, for failure-free executions.

Failure-free termination: In any fair failure-free execution in which *init* events occur on all ports, a *decide* event occurs on each port.

We say that a shared memory system A *solves the agreement problem* for a particular collection of users U_i if it guarantees the well-formedness, agreement, validity, and failure-free termination conditions for the users U_i. We say that it *solves the agreement problem* if it solves the agreement problem for all collections of users.

We also consider some stronger termination conditions involving fault-tolerance. The strongest condition we consider is the following, for executions in which any number of processes might fail:

Wait-free termination: In any fair execution in which *init* events occur on all ports, a *decide* event occurs on every non-failing port (i.e., every port i on which no $stop_i$ event occurs).

That is, any process that does not fail eventually decides, regardless of the failures of any of the other processes. This condition is analogous to the termination condition given in Section 6.1, for the stopping agreement problem in the synchronous setting. This condition is called *wait-freedom* because it implies that no process can ever be blocked, waiting indefinitely for help from any other process.

Note that we have stated the wait-freedom condition to assume that inputs arrive on all ports. We could have stated it equivalently to assume only that an input arrives at port i. We leave it as an exercise for you to show that this reformulation is in fact equivalent to our original statement.

Because the main impossibility result of this chapter involves only a single process failure rather than arbitrary process failures, we need yet another termination condition.

f-failure termination, $0 \leq f \leq n$: In any fair execution in which *init* events occur on all ports, if there are *stop* events on at most f ports, then a *decide* event occurs on every non-failing port.

It should be easy to see that the failure-free termination and wait-free termination conditions are the special cases of the f-failure termination condition where f is equal to 0 and n, respectively. The *single-failure termination* condition is the special case where $f = 1$.

Lemma 12.1 *Let A be an algorithm in the given architecture and U_i, $1 \leq i \leq n$, be a collection of users.*

1. *If A guarantees wait-free termination for the users U_i, then A guarantees f-failure termination for the U_i for any f, $0 \leq f \leq n$.*
2. *If A guarantees f-failure termination condition for the U_i for any f, $0 \leq f \leq n$, then A guarantees failure-free termination.*

We say that a shared memory system *guarantees wait-free termination*, *guarantees f-failure termination*, and so on, provided that it guarantees the corresponding condition for all collections of users.

Trace properties. As we did in Chapters 10 and 11, we can express the correctness conditions of this chapter equivalently in terms of trace properties. Each of these trace properties P has a signature consisting of $init(v)_i$ and $decide(v)_i$ outputs and $stop_i$ inputs. The external actions of the combined system are also exactly these actions, and the requirement in each case is that the fair traces of the combined system are all in $traces(P)$.

Synchronous termination conditions. The wait-free termination condition is similar to the termination condition used for the stopping agreement problem in the *synchronous* model, in Section 6.1, as well as to the strong termination condition used for the commit problem in Section 7.3. The failure-free termination condition is similar to the weak termination condition of Section 7.3.

In most of this chapter, we will consider the case of read/write shared memory, since that is the case in which the impossibility results hold. We allow the variables to be multi-writer/multi-reader registers. Near the end of the chapter, in Sections 12.3 and 12.4, we briefly consider other variable types.

12.2 Agreement Using Read/Write Shared Memory

Throughout this section, we suppose that A is an algorithm in the read/write shared memory model that solves the agreement problem and guarantees 1-failure termination. Our objective is to reach a contradiction, showing that such an A cannot exist.

We first make some simplifying restrictions on A, all without loss of generality, and then present some needed terminology. Next, we prove a result about the input values. Then, because the proof is easier, we show that the agreement problem is unsolvable in the read/write shared memory model, if the very strong *wait-free termination* condition is required. Finally, we show the main result--that not even a single fault can be tolerated.

12.2.1 Restrictions

For simplicity, and without loss of generality, we make the following four assumptions: First, we assume that the value set V is just $\{0, 1\}$. Second, we consider A in combination with a particular collection of users, trivial automata, each of which generates a single (arbitrary) *init* event and does nothing else.

Third, we assume that A is "deterministic," in the sense that the automaton has a unique initial state; that from any automaton state, any process has at most one locally controlled step; and that for any automaton state and any *init* input, there is a unique resulting automaton state. This does not restrict generality, because if we are given a nondeterministic solution, we could simply prune out all but one of the alternatives in each case. (This notion of determinism is similar to the one described in Exercise 8.9.)

Finally, we assume that every non-failed process always has a locally controlled step enabled, even after it decides. This does not restrict generality because we can always include dummy internal steps.

12.2.2 Terminology

We define an *initialization* to be an execution of the combination of A and the users consisting exactly of n *init* steps, one for each port, in order of index. Thus, the trace of an initialization has the form

$$init(v_1)_1, init(v_2)_2, \ldots, init(v_n)_n$$

where $v_1, \ldots, v_n \in V$. We define an execution α to be *input-first* provided that it begins with an initialization. Our proofs involve only input-first executions.

We define a finite execution α to be 0-*valent* if 0 is the only value that appears in a *decide* event in α or in any execution that extends α; moreover, we insist

that the value 0 actually does occur in some such decide event. After a 0-valent execution, the algorithm is already committed to 0 as the only decision value, even though no actual *decide*(0) event might yet have occurred. Similarly, α is 1-*valent* if the only such value is 1. We say that α is *univalent* if it is either 0-valent or 1-valent, and *bivalent* if each value appears in some extension. Figure 12.2 depicts a bivalent execution.

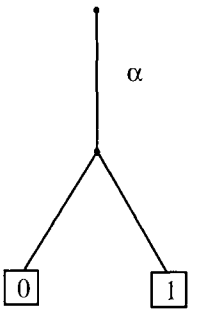

Figure 12.2: Bivalent execution α.

The following lemma says that this classification is exhaustive, in the absence of failures. That is, there are no finite failure-free executions after which *no* decision is possible.

Lemma 12.2 *Each finite failure-free execution α of A is either univalent or bivalent.*

Proof. Any such α can be extended to a fair failure-free execution α'. Then the failure-free termination condition guaranteed by A implies that in α', all processes eventually decide. □

If α is a finite failure-free execution and i is any process, then define $extension(\alpha, i)$ to be the execution obtained by extending execution α by a single step of i. The fact that this is well-defined depends on two of the restrictions we made above: that every non-failed process always has a locally controlled step enabled and that the system is deterministic. This notation is extended to sequences of process indices in the obvious way, for example, $extension(\alpha, ij) = extension(extension(\alpha, i), j)$.

12.2.3 Bivalent Initializations

We begin by showing that A must have a *bivalent initialization*. This means that the final decision value cannot be determined just from the inputs. In contrast,

if the algorithm is not required to tolerate any faults, then there are simple agreement algorithms in which the final value is completely determined by the inputs. We leave the discovery of such algorithms for an exercise.

Lemma 12.3 *A has a bivalent initialization.*

Proof. Suppose not; then all the initializations are univalent. Note that the initialization α_0 consisting of all 0s must be 0-valent, by the validity condition. Similarly, the initialization α_1 consisting of all 1s must be 1-valent.

Now we construct a *chain* of initializations, spanning from α_0 to α_1.[1] At each step of the chain, we simply change the initial value of a single process from 0 to 1; thus, any two consecutive initializations in the chain differ only in the input to one process. By assumption, every initialization in the chain is univalent, so there must be two consecutive initializations in the chain, say α and α', such that α is 0-valent and α' is 1-valent. Suppose that they differ in the initial value of process i.

Now consider any fair execution that extends α and in which i fails immediately after the initialization (i.e., the next action is $stop_i$), but in which none of the other processes ever fails. Then all processes other than i must eventually decide, by the 1-failure termination condition. Since α is 0-valent, this decision must be 0.

Now we claim that it is possible to extend α' in the same way and still obtain a decision of 0. This is because α and α' are identical except for the initial value of i, and i fails immediately after the initialization in both extensions; thus, the rest of the processes can behave in exactly the same way after α' as after α. See Figure 12.3.

But this contradicts the assumption that α' is 1-valent. □

12.2.4 Impossibility for Wait-Free Termination

Now we can prove the first (simpler) impossibility result—the one for wait-free termination. Namely, we suppose in this subsection that algorithm A has the wait-free termination property, which is stronger than the 1-failure termination property we have already assumed. We use the wait-free termination property to obtain a contradiction.

The contradiction is based on pinpointing a way in which a decision might be made. In particular, we define a *decider* execution α to be a finite failure-free input-first execution satisfying the following conditions:

[1] This chain construction is similar to the constructions used in the proof of Theorem 6.33, the lower bound on the number of rounds needed for agreement in the synchronous setting.

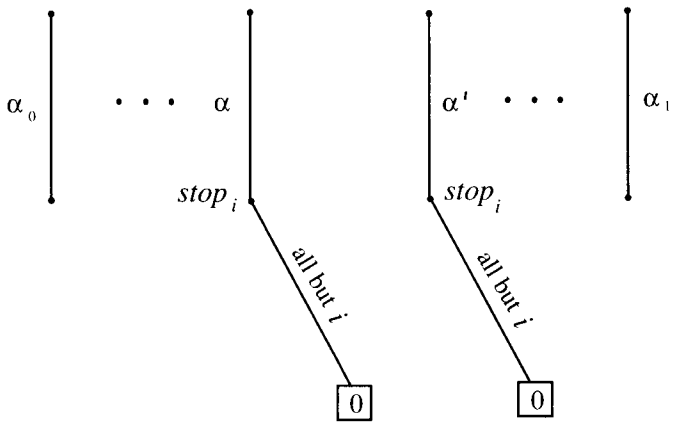

Figure 12.3: Construction for Lemma 12.3.

1. α is bivalent.
2. For every i, $extension(\alpha, i)$ is univalent.

Thus, after a decider execution, no decision has yet been determined, but any additional (non-*stop*) process step will determine the decision. We prove that A (with the wait-free termination property) must have a decider execution.

Lemma 12.4 *A has a decider execution.*

Proof. Suppose the contrary: that any bivalent failure-free input-first execution has a one-step bivalent failure-free extension.

Then starting with a bivalent initialization (whose existence is guaranteed by Lemma 12.3), we can produce an infinite failure-free input-first execution α, all of whose finite prefixes are bivalent. Thus, in α, no process ever decides. The construction is simple: at each stage, we start with a bivalent failure-free input-first execution, and we extend it by one step to another bivalent failure-free execution. Our assumption at the beginning of this proof says that we can do this.

Since α is infinite, it must contain infinitely many steps for some process, say i. We claim that i must decide in α, which yields a contradiction.

To see this, modify α by inserting a $stop_j$ event for each process j that only takes finitely many steps, right after its last step in α. Call the modified execution α'. Then α' is a fair execution in which process i does not fail. The wait-free termination condition then implies that i must decide in α'. But α and α' look identical to process i, so i decides in α also. This is the contradiction needed to prove this lemma. □

Now we can obtain the contradiction that we need to prove the impossibility result for wait-free termination.

Lemma 12.5 *A does not exist.*

Proof. By Lemma 12.4, we may fix some decider execution α. Since α is bivalent, there exist two processes, say i and j, such that after α, a step of i leads to a 0-valent execution and a step of j leads to a 1-valent execution. That is, $extension(\alpha, i)$ is 0-valent and $extension(\alpha, j)$ is 1-valent. Obviously, $i \neq j$. See Figure 12.4.

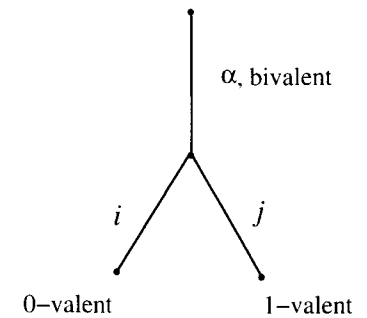

Figure 12.4: Execution α is a decider; $extension(\alpha, i)$ is 0-valent, and $extension(\alpha, j)$ is 1-valent.

We complete the proof with a case analysis, getting a contradiction for each possibility.

1. Process i's step is a read step.

   Consider extending $extension(\alpha, j)$ in such a way that no process fails, process i takes no further steps, and each process except for i takes infinitely many steps. This looks to every process except i like a fair execution in which process i fails immediately and no other process fails. Thus, by the wait-free termination condition (in fact, 1-failure termination is enough here), all processes except i must eventually decide, and since $extension(\alpha, j)$ is 1-valent, they must decide 1.

   Now, note that the states after α and $extension(\alpha, i)$ are indistinguishable to every process except i, in the sense defined in Section 9.3. This is because i's step is just a read, so the only thing it changes is the state of process i. So we can take the same suffix that we previously ran after α, beginning with the step of j, and run it after $extension(\alpha, i)$. In this case also, all processes except i decide 1, which contradicts the assumption that $extension(\alpha, i)$ is 0-valent. See Figure 12.5.

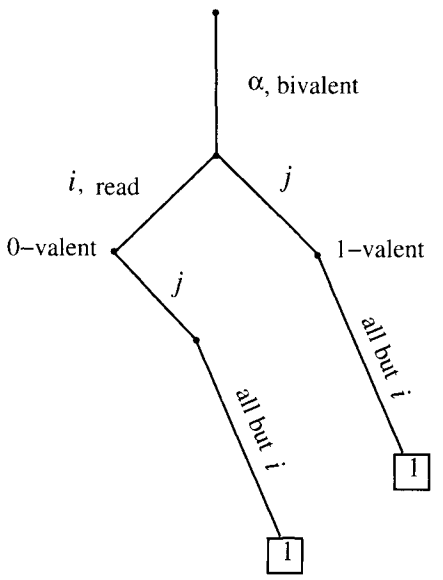

Figure 12.5: Construction for Case 1.

2. Process j's step is a read step.

   This case is symmetric to Case 1, and the same argument applies.

3. Process i's step and process j's step are both writes.

   We distinguish two subcases.

   (a) Processes i and j write to different variables.

   Consider two executions that extend α, one by allowing first i to take its step and then j, and the other allowing first j, then i. Since the two steps involve different processes and different variables, the system state is the same after either execution. See Figure 12.6.

   But then we have a common system state that can be reached after either a 0-valent or a 1-valent execution. If we run all the processes from this state with no failures, they are required to decide. However, either decision yields a contradiction. For instance, if a decision of 0 is reached, then we have a decision of 0 in an execution extending a 1-valent prefix.

   (b) Processes i and j write to the same variable.

   As in Case 1, we can run all processes but i after $extension(\alpha, j)$ until they decide 1. This time, note that the states after $extension(\alpha, j)$

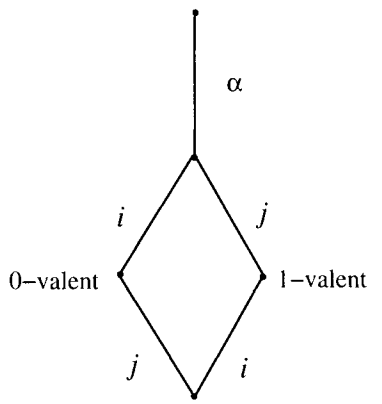

Figure 12.6: Construction for Case 3(a).

and *extension*(α, ij) are indistinguishable to every process except for i. This is because the step of j overwrites the value written by i, so the only memory of i's step is in the state of process i. So we can take the same suffix that we previously ran after *extension*(α, j) and run it after *extension*(α, ij). In this case also, all processes except i decide 1, which contradicts the 0-valence of *extension*(α, i). See Figure 12.7.

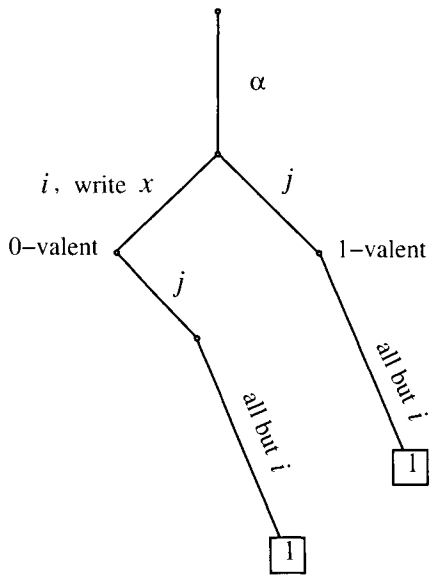

Figure 12.7: Construction for Case 3(b).

So, we have contradictions in all possible cases, and thus we conclude that no such algorithm A can exist. □

We have proved the first impossibility theorem:

Theorem 12.6 *For $n \geq 2$, there is no algorithm in the read/write shared memory model that solves the agreement problem and guarantees wait-free termination.*

12.2.5 Impossibility for Single-Failure Termination

Notice that the proof of Theorem 12.6 in the previous section does not work for the case where we only assume 1-failure termination. The problem is in the proof of Lemma 12.4, where we use the wait-free termination condition to assert that process i must decide in a fair execution in which it does not fail. In this section, we strengthen Theorem 12.6 to obtain the corresponding result for systems with 1-failure termination.

This time, the proof is based on the following lemma, which says that a bivalent execution can be extended to allow a given process to take a step, while still maintaining bivalence.

Lemma 12.7 *If α is a bivalent failure-free input-first execution of A, and i is any process, then there is a failure-free extension α' of α such that extension(α', i) is bivalent.*

See Figure 12.8.

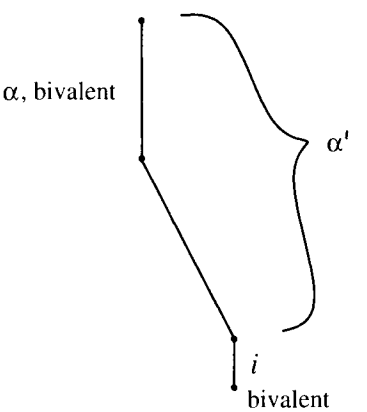

Figure 12.8: Maintaining bivalence while allowing process i to take a step.

At this point, you might prefer to skip ahead to the proof of Theorem 12.8, to see how Lemma 12.7 implies the impossibility result, before delving into the more technical proof of the lemma.

Proof. We prove this lemma by contradiction. Suppose that the lemma is false. Then there must be some bivalent failure-free input-first execution α of A and some process i such that for every failure-free extension α' of α, $extension(\alpha', i)$ is univalent. This implies, in particular, that $extension(\alpha, i)$ is univalent; suppose without loss of generality that it is 0-valent.

Since α is bivalent, there is some extension α'' of α containing a decision of 1. We may assume without loss of generality that α'' is failure-free, since otherwise we could simply eliminate any *stop* actions without affecting the decision. Then it must be that $extension(\alpha'', i)$ is 1-valent. We consider what happens if i takes a step at each point along the "path" from α to α''. See Figure 12.9.

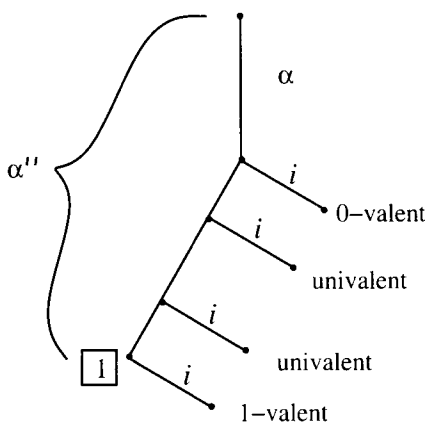

Figure 12.9: Process i's step leads to univalence.

At the beginning of the path, i's step yields 0-valence, while at the end, it yields 1-valence. At each intermediate point, it yields univalence. Therefore, it must be that there are two consecutive points in the path such that at the first of these points, i's step yields 0-valence, while at the second, i's step yields 1-valence. Let α' be the execution up to the first point. See Figure 12.10.

Suppose that j is the process that takes the intervening step. We claim that $j \neq i$. This is true because if $j = i$, then we have a situation where one step of i leads to 0-valence while two steps of i lead to 1-valence; since the processes are deterministic, this gives a 0-valent execution with a 1-valent extension, which is nonsense.

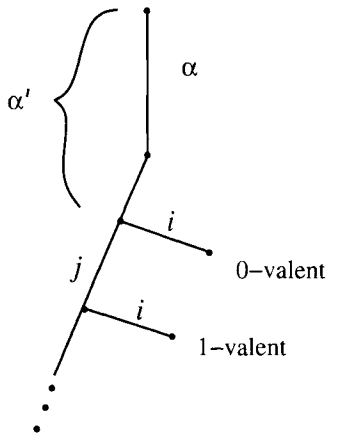

Figure 12.10: Two consecutive points where i yields different valences.

We finish with a case analysis similar to the one in the proof of Lemma 12.5, obtaining a contradiction for each case.

1. Process i's step is a read step.

   Then we claim that the states after $extension(\alpha', ji)$ and $extension(\alpha', ij)$ are indistinguishable to every process except for i. This is because the steps of i involved in these two extensions are both read steps, which do not affect anything except the state of process i.

   Consider extending $extension(\alpha', ij)$ in such a way that i takes no further steps and every other process takes infinitely many steps. By the 1-failure termination condition, all processes except i must eventually decide, and since $extension(\alpha', i)$ is 0-valent, they must decide 0. By the indistinguishability claim just above, we can take the same suffix that we previously ran after $extension(\alpha', ij)$ and run it after $extension(\alpha', ji)$. In this case also, the processes decide 0, which contradicts the 1-valence of $extension(\alpha', ji)$. See Figure 12.11.

2. Process j's step is a read step.

   The argument is similar to that for Case 1. This time, the states after $extension(\alpha', i)$ and $extension(\alpha', ji)$ are indistinguishable to all processes except j. We can let all processes except j run after $extension(\alpha', i)$, forcing them eventually to decide 0. Then we can run them in the same way after $extension(\alpha', ji)$. They again decide 0, contradicting the 1-valence of $extension(\alpha', ji)$.

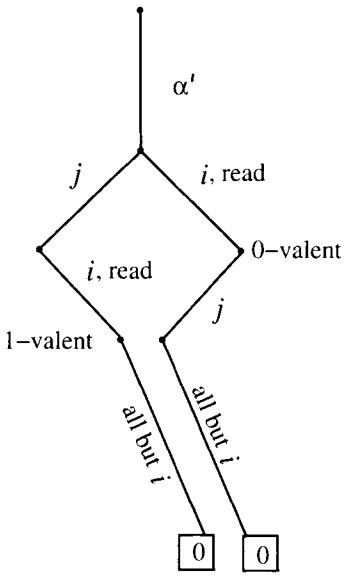

Figure 12.11: Construction for Case 1.

3. Process i's step and process j's step are both writes.

   (a) Processes i and j write to different variables.

   In this case we get the same sort of commutative scenario as in Case 3(a) of the proof of Lemma 12.5; see Figure 12.6. This implies the same contradiction as in that proof.

   (b) Processes i and j write to the same variable.

   In this case, the states after $extension(\alpha', i)$ and $extension(\alpha', ji)$ are indistinguishable to all processes except for j, because i's step overwrites j's step. Running all processes except for j after $extension(\alpha', i)$ and $extension(\alpha', ji)$ yields the same contradiction as before. See Figure 12.12.

□

We can now prove the main theorem.

Theorem 12.8 *For $n \geq 2$, there is no algorithm in the read/write shared memory model that solves the agreement problem and guarantees 1-failure termination.*

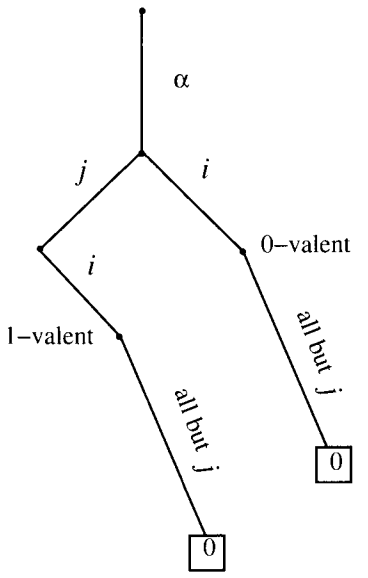

Figure 12.12: Construction for Case 3(b).

Proof. We use Lemma 12.7 to construct a fair failure-free input-first execution in which no process ever decides. This contradicts the failure-free termination requirement.

The construction begins with a bivalent initialization, whose existence is guaranteed by Lemma 12.3. Then we repeatedly extend the current execution, including at least one step of process 1 in the first extension, then at least one step of 2 in the second extension, and so on, in round-robin order, all while maintaining bivalence and avoiding failures. Lemma 12.7 implies that each extension step is possible.

The resulting execution is fair, because each process takes infinitely many steps. However, no process ever reaches a decision, which gives the needed contradiction. □

12.3 Agreement Using Read-Modify-Write Shared Memory

In contrast to the situation for read/write shared memory, it is very easy to solve the agreement problem, guaranteeing wait-free termination, using read-modify-write shared memory. In fact, a single read-modify-write shared variable is enough.

***RMWAgreement* algorithm:**

The shared variable begins with the value *unknown*. Each process accesses the variable. If it sees the value *unknown*, then it changes the value to its own initial value and decides on this value. On the other hand, if it sees a value $v \in V$, then it does not change the value written in the variable but instead accepts the previously written value as its decision value.

The precondition-effect code for process i of the *RMWAgreement* algorithm is the same as the code in Example 9.1.1, with the addition of some code for handling failures. Namely, the state contains an additional component *stopped*, a set of processes, initially empty. There is a new $stop_i$ action, which puts i into *stopped*. The $access_i$ and $decide_i$ actions have the additional precondition that $i \notin stopped$, and the $init_i$ action only makes its changes if $i \notin stopped$.

Theorem 12.9 *The RMWAgreement algorithm solves the agreement problem and guarantees wait-free termination.*

Proof. Straightforward. Wait-free termination follows, because each process i, after receiving an $init_i$ input, is immediately enabled to perform an $access_i$ and then a $decide_i$. Agreement and validity follow, because the first process to perform an *access* establishes the common decision value. □

12.4 Other Types of Shared Memory

The agreement problem can also be considered using shared memory of other variable types besides read/write and read-modify-write. For example, we can consider variables with operations such as *swap*, *test-and-set*, *fetch-and-add*, and *compare-and-swap*. These operations are defined in Example 9.4.3. Among the known results are the following theorems:

Theorem 12.10 *The agreement problem for any n can be solved with wait-free termination, using a single shared variable that allows compare-and-swap operations only.*

Theorem 12.11 *If $n \geq 3$, then the agreement problem cannot be solved with wait-free termination using shared variables that allow any combination of swap, test-and-set, fetch-and-add, read, and write operations.*

Proofs. The proofs are left for exercises. □

12.5 Computability in Asynchronous Shared Memory Systems*

The agreement problem is only one example of a "decision problem" that can be considered in the asynchronous shared memory model with stopping failures. In this section, we define the general notion of a decision problem, give some examples, and state (without proof) some typical computability results.

Our definition of a decision problem is based on the preliminary definition of a decision mapping. A *decision mapping* D specifies, for each length n vector w of inputs over some fixed value set V, a nonempty set $D(w)$ of allowable length n vectors of decisions. The vector w represents the inputs of processes $1, \ldots, n$, in order of process index, and, similarly, each vector in $D(w)$ represents the decisions of processes $1, \ldots, n$, also in order of process index.

We use a decision mapping D in the formulation of a problem to be solved by an asynchronous shared memory system. The external interface of the shared memory system consists of $init(v)_i$, $decide(v)_i$, and $stop_i$ actions, just as for the agreement problem. The well-formedness condition and the various termination conditions are defined in exactly the same way as for the agreement problem. However, in place of the agreement and validity conditions used in the agreement problem, we require only the single validity condition:

Validity: In any execution in which *init* events occur on all ports, it is possible to complete the vector of decisions that are reached by the processes to a vector in $D(w)$, where w is the given input vector.

Example 12.5.1 The agreement problem as a decision problem

The agreement problem is an example of a decision problem, based on the decision mapping D, defined as follows. For any vector $w = v_1, \ldots, v_n$ of inputs in V, the set $D(w)$ of allowable vectors of decisions is defined by

1. If $v_1 = v_2 = \ldots = v_n = v$, then $D(w)$ contains the single vector $x_1, \ldots, x_n$ such that $x_1 = x_2 = \ldots = x_n = v$.
2. If $v_i \neq v_j$ for some i and j, then $D(w)$ consists of exactly those vectors $x_1, \ldots, x_n$ such that $x_1 = x_2 = \ldots = x_n$.

It is easy to see that the decision problem based on D is the same as the agreement problem. (One apparent difference is that the definition of a general decision problem only mentions executions in which *init* inputs arrive on all ports, whereas the definition of the agreement problem involves other executions as well. But this is not an

important difference, because any finite execution can be extended to one in which inputs arrive everywhere.)

Two other important examples of decision problems are the k-agreement problem and the approximate agreement problem, both of which we studied in the synchronous network model in Chapter 7. In the *k-agreement* problem, where k is any positive integer, the agreement and validity conditions of the agreement problem are replaced with the following:

Agreement: In any execution, there is a subset W of V, $|W| = k$, such that all decision values are in W.

Validity: In any execution, any decision value for any process is the initial value of some process.

The agreement condition is weaker than that for ordinary agreement in that it permits k decision values rather than only one. The validity condition is a slight strengthening of the validity condition for ordinary agreement. It is easy to formalize the k-agreement problem as a decision problem. The following can be shown, though we omit the proofs here and leave them for exercises:

Theorem 12.12 *The k-agreement problem is solvable with $k-1$-failure termination in the asynchronous read/write shared memory model using single-writer/multi-reader registers.*

Theorem 12.13 *The k-agreement problem is not solvable with k-failure termination in the asynchronous shared memory model with multi-writer/multi-reader registers.*

In the *approximate agreement* problem, the set V of values is the set of real numbers, and processes are permitted to send real-valued data in messages. Now instead of having to agree exactly, as in the agreement problem, the requirement is that the processes agree approximately, to within a small positive tolerance ϵ. That is, the agreement and validity conditions of the agreement problem are replaced with the following:

Agreement: In any execution, any two decision values are within ϵ of each other.

Validity: In any execution, any decision value is within the range of the initial values.

Again, it is easy to formalize this problem as a decision problem.