Chapter 6

Distributed Consensus with Process Failures

In this chapter we continue the study of consensus problems in the synchronous model, which we began in Chapter 5. This time, we consider the case where processes, but not links, may fail. Of course, it is more sensible to talk about failure of physical "processors" than of logical "processes," but to stay consistent with the terminology elsewhere in the book, we use the term *process*. We investigate two failure models: the *stopping failure* model, where processes may simply stop without warning, and the *Byzantine failure* model, where faulty processes may exhibit completely unconstrained behavior. Stopping failures are intended to model unpredictable processor crashes. Byzantine failures are intended to model any arbitrary type of processor malfunction, including, for example, failures of individual components within the processors.

The term *Byzantine* was first used for this type of failure in a landmark paper by Lamport, Pease, and Shostak, in which a consensus problem is formulated in terms of *Byzantine generals*. As in the coordinated attack problem of Chapter 5, the Byzantine generals attempt to agree on whether or not to carry out an attack. This time, however, the generals must worry not about lost messengers, but about the possible traitorous behavior of some generals. The term *Byzantine* is intended as a pun—the battle scenario takes place in ancient Byzantium, and the behavior of some of the traitorous generals can only be described as "Byzantine."

In the particular consensus problem we consider in this chapter, which we call simply the *agreement problem*, the processes start with individual inputs from a particular value set V. All the nonfaulty processes are required to produce outputs from the same value set V, subject to simple agreement and validity

conditions. (For validity, we assume that if all processes begin with the same value v, the only allowed decision value is v.)

The agreement problem is a simplified version of a problem that originally arose in the development of on-board aircraft control systems. In this problem, a collection of processors, each with access to a separate altimeter, and some of which may be faulty, attempt to agree on the airplane's altitude. Byzantine agreement algorithms have also been incorporated into the hardware of fault-tolerant multiprocessor systems; there, they are used to help a small collection of processors to carry out identical computations, agreeing on the results of every step. This redundancy allows the processors to tolerate the (Byzantine) failure of one processor. Byzantine agreement algorithms are also useful in processor fault diagnosis, where they can permit a collection of processors to agree on which of their number have failed (and should therefore be replaced or ignored).

In both of our failure models, we will need to assume limitations on the frequency of occurrence of process(or) failures. How should such limitations be expressed? In other work on analysis of systems with processor failures, these limitations often take the form of probability distributions governing the occurrences of failures. Here, instead of using probabilities, we simply assume that the number of failures is bounded in advance, by a fixed number f. This is a simple assumption to work with, since it avoids the complexities of reasoning about probabilistic failure occurrences. In practice, this assumption may be realistic in the sense that it may be unlikely that more than f failures will occur. However, we should keep in mind that the assumption is somewhat problematic: in most practical situations, if the number of failures is already large, then it is likely that more failures will occur. Assuming a bound on the number of failures implies that failures are *negatively correlated*, whereas in practice, failures are usually independent or positively correlated.

After defining the agreement problem, for both stopping and Byzantine failures, we present a series of algorithms. We then prove lower bounds on the number of processes needed to solve the problem for Byzantine failures, and on the number of rounds needed to solve the problem for either type of failure.

6.1 The Problem

We assume that the network is an n-node connected undirected graph with processes $1, \ldots, n$, where each process knows the entire graph. Each process starts with an input from a fixed value set V in a designated state component; we assume that, for each process, there is exactly one start state containing each input value. The goal is for the processes to eventually output decisions from the set V, by setting special *decision* state components to values in V. We use the same

synchronous model that we have been using in Chapters 3–5, only this time we allow the possibility that a limited number (at most f) of processes might fail. In this chapter, we assume that the links are perfectly reliable—all the messages that are sent are delivered. We consider two kinds of process failures: stopping failures and Byzantine failures.

In the stopping failure model, at any point during the execution of the algorithm, a process might simply stop taking steps altogether. In particular, a process might stop *in the middle of a message-sending step*; that is, at the round in which the process stops, only a subset of the messages the process is supposed to send might actually be sent. In this case, we assume that *any subset* of the messages might be sent. A process might also stop after sending its messages for some round but before performing its transition for that round.

For the stopping failure model, the correctness conditions for the agreement problem are

Agreement: No two processes decide on different values.

Validity: If all processes start with the same initial value $v \in V$, then v is the only possible decision value.

Termination: All nonfaulty processes eventually decide.

In the Byzantine failure model, a process might fail not just by stopping, but by exhibiting arbitrary behavior. This means that it might start in an *arbitrary state*, not necessarily one of its start states; might send *arbitrary messages*, not necessarily those specified by its *msgs* function; and might perform *arbitrary state transitions*, not necessarily those specified by its *trans* function. (As a technical but convenient special case, we even allow for the possibility that a Byzantine process behaves completely correctly.) The only limitation on the behavior of a failed process is that it can only affect the system components over which it is supposed to have control, namely, its own outgoing messages and its own state. It cannot, for example, corrupt the state of another process, or modify or replace another process's messages.

For the Byzantine failure model, the agreement and validity conditions are slightly different from those for the stopping failure model:

Agreement: No two nonfaulty processes decide on different values.

Validity: If all nonfaulty processes start with the same initial value $v \in V$, then v is the only possible decision value for a nonfaulty process.

Termination: The termination condition is the same.

The modified conditions reflect the fact that in the Byzantine model, it is impossible to impose any limitations on what the faulty processes might start

with or what they might decide. We refer to the agreement problem for the Byzantine failure model as the *Byzantine agreement problem.*

Relationship between the stopping and Byzantine agreement problems. It is not quite the case that an algorithm that solves the Byzantine agreement automatically solves the agreement problem for stopping failures; the difference is that in the stopping case, we require that all the processes that decide, *even those that subsequently fail*, must agree. If the agreement condition for the stopping failure case is replaced by the one for the Byzantine failure case, then the implication does hold. Alternatively, if all the nonfaulty processes in the Byzantine algorithm always decide at the same round, then the algorithm also works for stopping failures. The proofs are left as exercises.

Stronger validity condition for stopping failures. An alternative validity condition that is sometimes used for the stopping failure model is as follows.

Validity: Any decision value for any process is the initial value of some process.

It is easy to see that this condition implies the validity condition we have already stated. We will use this stronger condition in our definition of the k-agreement problem, a generalization of the agreement problem, in Chapter 7. In this chapter, we use the weaker condition we gave earlier; this slightly weakens our claims about algorithms and slightly strengthens our impossibility results. For the algorithms in this chapter, we will indicate explicitly whether or not this stronger validity condition is satisfied.

Complexity measures. For the time complexity, we count the number of rounds until all the nonfaulty processes decide. For the communication complexity, we count both the number of messages and number of bits of communication; in the stopping case, we base these counts on the messages sent by all processes, but in the Byzantine case, we only base it on the messages sent by nonfaulty processes. This is because there is no way to provide nontrivial bounds on the communication sent by faulty processes in the Byzantine model.

6.2 Algorithms for Stopping Failures

In this section, we present algorithms for agreement in the stopping failure model, for the special case of a complete n-node graph. We begin with a basic algorithm in which each process just repeatedly broadcasts the set of all values it has ever seen. We continue with some reduced-complexity versions of the basic algorithm,

and finally, we present algorithms that use a strategy known as *exponential information gathering (EIG)*. Exponential information gathering algorithms, though costly and somewhat complicated, extend to less well-behaved fault models.

Conventions. In this and the following section, we use v_0 to denote a prespecified default value in the input set V. We also use b to denote an upper bound on the number of bits needed to represent any single value in V.

6.2.1 A Basic Algorithm

The agreement problem for stopping failures has a very simple algorithm, called *FloodSet*. Processes just propagate all the values in V that they have ever seen and use a simple decision rule at the end.

> ***FloodSet* algorithm (informal):**
>
> Each process maintains a variable W containing a subset of V. Initially, process i's variable W contains only i's initial value. For each of $f+1$ rounds, each process broadcasts W, then adds all the elements of the received sets to W.
>
> After $f+1$ rounds, process i applies the following decision rule. If W is a singleton set, then i decides on the unique element of W; otherwise, i decides on the default value v_0.

The code follows.

> ***FloodSet* algorithm (formal):**
>
> The message alphabet consists of subsets of V.
>
> ***states*$_i$:**
> *rounds* $\in \mathbb{N}$, initially 0
> *decision* $\in V \cup \{unknown\}$, initially *unknown*
> $W \subseteq V$, initially the singleton set consisting of i's initial value
>
> ***msgs*$_i$:**
> if *rounds* $\leq f$ then send W to all other processes
>
> ***trans*$_i$:**
> *rounds* := *rounds* + 1
> let X_j be the message from j, for each j from which a message arrives
> $W := W \cup \bigcup_j X_j$
> if *rounds* $= f+1$ then
> if $|W| = 1$ then *decision* := v, where $W = \{v\}$
> else *decision* := v_0

In arguing the correctness of *FloodSet*, we use the notation $W_i(r)$ to denote the value of variable W at process i after r rounds. As usual, we use the subscript i to denote the instance of a state component belonging to process i. We say that a process is *active* after r rounds if it does not fail by the end of r rounds.

The first easy lemma says that if there is ever a round at which no process fails, then all the active processes have the same W at the end of that round.

Lemma 6.1 *If no process fails during a particular round r, $1 \leq r \leq f+1$, then $W_i(r) = W_j(r)$ for all i and j that are active after r rounds.*

Proof. Suppose that no process fails at round r and let I be the set of processes that are active after r rounds (or equivalently, after $r-1$ rounds). Then, because every process in I sends its own W set to all other processes, at the end of round r, the W set of each process in I is exactly the set of values that are held by processes in I just before round r. □

We next claim that if all the active processes have the same W sets after some particular round r, then the same is true after subsequent rounds.

Lemma 6.2 *Suppose that $W_i(r) = W_j(r)$ for all i and j that are active after r rounds. Then for any round r', $r \leq r' \leq f+1$, the same holds, that is, $W_i(r') = W_j(r')$ for all i and j that are active after r' rounds.*

Proof. The proof is left as an exercise. □

The following lemma is crucial for the agreement property.

Lemma 6.3 *If processes i and j are both active after $f+1$ rounds, then $W_i = W_j$ at the end of round $f+1$.*

Proof. Since there are at most f faulty processes, there must be some round r, $1 \leq r \leq f+1$, at which no process fails. Lemma 6.1 implies that $W_i(r) = W_j(r)$ for all i and j that are active after r rounds. Then Lemma 6.2 implies that $W_i(f+1) = W_j(f+1)$ for all i and j that are active after $f+1$ rounds. □

Theorem 6.4 *FloodSet solves the agreement problem for stopping failures.*

Proof. Termination is obvious, by the decision rule. For validity, suppose that all the initial values are equal to v. Then v is the only value that ever gets sent anywhere. Each set $W_i(f+1)$ is nonempty, because it contains i's initial value. Therefore, each $W_i(f+1)$ must be exactly equal to $\{v\}$, so the decision rule says that v is the only possible decision.

For agreement, let i and j be any two processes that decide. Since decisions only occur at the end of round $f+1$, it means that i and j are active after $f+1$ rounds. Lemma 6.3 then implies that $W_i(f+1) = W_j(f+1)$. The decision rule then implies that i and j make the same decision. □

Complexity analysis. *FloodSet* requires exactly $f+1$ rounds until all nonfaulty processes decide. The total number of messages is $O((f+1)n^2)$. Each message contains a set of at most n elements (since each element must be the initial value of some process), so the number of bits per message is $O(nb)$. Thus, the total number of communication bits is $O((f+1)n^3b)$.

Alternative decision rule. The decision rule given for *FloodSet* is somewhat arbitrary. Since *FloodSet* guarantees that all nonfaulty processes obtain the same set W after $f+1$ rounds, various other decision rules would also work correctly, as long as all the processes apply the same rule. For instance, if the value set V has a total ordering, then all processes could simply choose the minimum value in W. This alternative rule has the advantage that it guarantees the stronger validity condition mentioned near the end of Section 6.1. The decision rule given for *FloodSet* does not guarantee this stronger condition, because the default value v_0 might not be the initial value of any process.

Process versus communication failures. The *FloodSet* algorithm shows that the agreement problem is solvable for process stopping failures. This positive result should be contrasted with the impossibility results for the coordinated attack problem in a setting with communication failures. (See Theorem 5.1 and Exercise 5.1.)

6.2.2 Reducing the Communication

It is possible to reduce the amount of communication somewhat from the $O((f+1)n^2)$ messages and $O((f+1)n^3b)$ bits used by *FloodSet*. For example, the number of messages can be reduced to $2n^2$ and the number of bits of communication to $O(n^2b)$ by using the following simple idea. Notice that at the end, each process i only needs to know the exact elements of its set W_i if $|W_i| = 1$; otherwise, i needs to know only the fact that $|W_i| \geq 2$. So it is plausible that each process might need to broadcast *only the first two values* it sees, rather than all values. This idea is the basis for the following algorithm.

***OptFloodSet* algorithm:**

The processes operate as in *FloodSet*, except that each process i broadcasts at most two values altogether. The first broadcast is at round 1, when i

broadcasts its initial value. The second broadcast is at the first round r, $2 \leq r \leq f+1$, such that at the beginning of round r, i knows about some value v different from its initial value (if any such round exists). Then i broadcasts this new value v. (If there are two or more new values at this round, then any one of these may be selected for broadcast.)

As in *FloodSet*, process i decides v if its final set W_i is the singleton set $\{v\}$ and otherwise decides v_0.

Complexity analysis. The number of rounds for *OptFloodSet* is the same as for *FloodSet*, $f+1$. The number of messages is at most $2n^2$, since each process sends at most two non-*null* messages to each other process. The number of bits of communication is $O(n^2 b)$.

We prove the correctness of *OptFloodSet* by relating it to *FloodSet* using a *simulation relation* (a similar strategy was used in Section 4.1.3 to prove correctness of *OptFloodMax* by relating it to *FloodMax*). This requires first filling in the details in the description of *OptFloodSet*, including explicit *rounds*, *decision*, and W variables as in *FloodSet*. We use the notation $W_i(r)$ and $OW_i(r)$, respectively, to denote the values of W_i after r rounds of *FloodSet* and *OptFloodSet*, respectively. The following lemma describes message propagation in *FloodSet*.

Lemma 6.5 *In FloodSet, suppose that i sends a round $r+1$ message to j, and j receives and processes it. Then $W_i(r) \subseteq W_j(r+1)$.*

Proof. The proof is left as an exercise. □

The key pruning property of *OptFloodSet* is captured by the following lemma.

Lemma 6.6 *In OptFloodSet, suppose that i sends a round $r+1$ message to j, and j receives and processes it. Then*

1. *If $|OW_i(r)| = 1$, then $OW_i(r) \subseteq OW_j(r+1)$.*
2. *If $|OW_i(r)| \geq 2$, then $|OW_j(r+1)| \geq 2$.*

Moreover, the same two conclusions hold in case i does not fail in the first r rounds, and does not send a round $r+1$ message to j, but just because OptFloodMax does not specify that any such message is supposed to be sent.

Proof. The proof is left as an exercise. □

Now we run *OptFloodSet* and *FloodSet* side by side, with the same inputs and same failure pattern. That is, the same processes fail at the same rounds in

both executions. Moreover, if process i sends only some of its round r messages in one algorithm, then it sends its round r messages to the same processes in the other algorithm; more precisely, there is no j to which i sends a message at round r in one algorithm but fails to send one that it is supposed to send in the other algorithm. We give invariant assertions relating the states of the two algorithms.

Lemma 6.7 *After any number of rounds r, $0 \leq r \leq f+1$:*

1. $OW_i(r) \subseteq W_i(r)$.
2. *If* $|W_i(r)| = 1$, *then* $OW_i(r) = W_i(r)$.

Proof. The proof is left as an exercise. □

Lemma 6.8 *After any number of rounds r, $0 \leq r \leq f+1$:*

If $|W_i(r)| \geq 2$, *then* $|OW_i(r)| \geq 2$.

Proof. By induction. The basis case, $r = 0$, is true vacuously. Assume now that the lemma holds for r. We show that it holds for $r+1$. Suppose that $|W_i(r+1)| \geq 2$. If $|W_i(r)| \geq 2$, then by inductive hypothesis we have that $|OW_i(r)| \geq 2$, which implies that $|OW_i(r+1)| \geq 2$, as needed.

So assume that $|W_i(r)| = 1$. Then Lemma 6.7 implies that $OW_i(r) = W_i(r)$. We consider two subcases.

1. $|W_j(r)| = 1$ for all j from which i receives a round $r+1$ message in *FloodSet*.
   Then for all such j, we have by Lemma 6.7 that $OW_j(r) = W_j(r)$, so that $|OW_j(r)| = 1$. Lemma 6.6 implies that for all such j, $OW_j(r) \subseteq OW_i(r+1)$. It follows that $OW_i(r+1) = W_i(r+1)$, which is sufficient to prove the inductive step.

2. $|W_j(r)| \geq 2$ for some j from which i receives a round $r+1$ message in *FloodSet*.
   Then by the inductive hypothesis, $|OW_j(r)| \geq 2$. Then Lemma 6.6 implies that $|OW_i(r+1)| \geq 2$, as needed. □

Lemma 6.9 *After any number of rounds r, $0 \leq r \leq f+1$, the rounds and decision variables have the same values in FloodSet and OptFloodSet.*

Proof Sketch. The interesting thing to show is that the same decision is made by any process i at round $f+1$ in the two algorithms. This follows from Lemmas 6.7 and 6.8 for $r = f+1$ and the decision rules of the two algorithms. □

Theorem 6.10 *OptFloodSet solves the agreement problem for stopping failures.*

Proof. By Lemma 6.9 and Theorem 6.4 (the correctness theorem for *FloodSet*). □

Other ways to reduce communication complexity. There are other ways to reduce the communication complexity of *FloodSet*. For example, recall that if V has a total ordering, the decision rule can be modified to simply choose the minimum value in W. Then it is possible to modify the *FloodSet* algorithm so that each node just remembers and relays the minimum value it has seen so far, rather than all values. This algorithm uses $O((f+1)n^2b)$ communication bits. It can be proved correct by a simulation relating it to *FloodSet* (with the modified decision rule). This algorithm satisfies the stronger validity condition of Section 6.1.

6.2.3 Exponential Information Gathering Algorithms

In this section, we present algorithms for agreement with stopping failures based on a strategy known as *exponential information gathering* (*EIG*). In exponential information gathering algorithms, processes send and relay initial values for several rounds, recording the values they receive along various communication paths in a data structure called an *EIG tree*. At the end, they use a commonly agreed-upon decision rule based on the values recorded in their trees.

EIG algorithms are generally costly for solving agreement with stopping failures, both in terms of the number of bits that are communicated and the amount of local storage used. The main reason we present this strategy here is that the same *EIG* tree data structure can be used for solving Byzantine agreement, as we show in Section 6.3.2. The stopping failure case provides a simple introduction to the use of this data structure. A second reason for presenting this strategy for stopping failures is that simple stopping failure *EIG* algorithms can easily be adapted to solve the agreement problem for a restricted form of the Byzantine failure model known as the *authenticated Byzantine failure* model.

The basic data structure used by *EIG* algorithms is a labelled *EIG tree* $T = T_{n,f}$, whose paths from the root represent chains of processes along which initial values are propagated; all chains represented consist of distinct processes. The tree T has $f+2$ levels, ranging from level 0 (the root) to level $f+1$ (the leaves). Each node at level k, $0 \leq k \leq f$, has exactly $n-k$ children. Each node in T is labelled by a string of process indices as follows. The root is labelled by the empty string λ, and each node with label $i_1 \ldots i_k$ has exactly $n-k$ children with

labels $i_1 \ldots i_k j$, where j ranges over all the elements of $\{1, \ldots, n\} - \{i_1, \ldots, i_k\}$. See Figure 6.1 for an illustration.

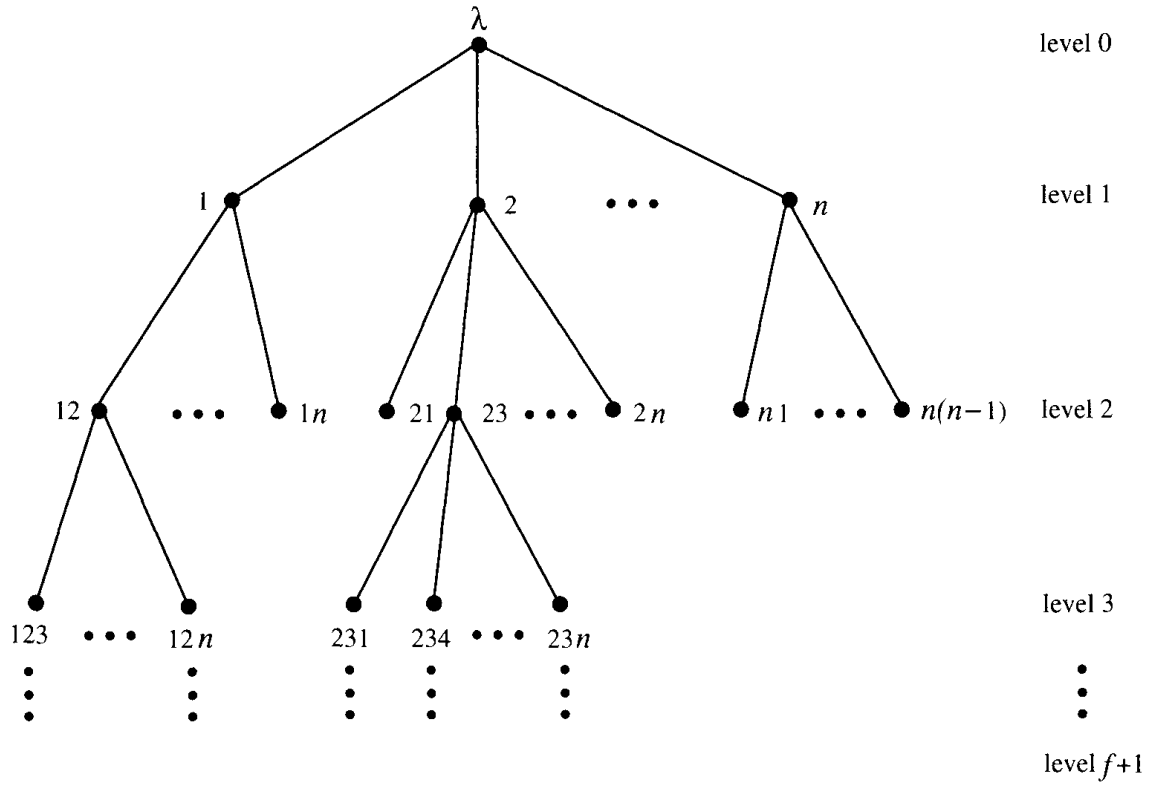

Figure 6.1: The *EIG* tree $T_{n,f}$.

In the *EIG* algorithm for stopping failures, which we call *EIGStop*, the processes simply relay values on all possible paths. Each process maintains a copy of the *EIG* tree $T = T_{n,f}$. The computation proceeds for exactly $f + 1$ rounds. In the course of the computation, the processes decorate the nodes of their trees with values in V or *null*, decorating all those at level k at the end of round k. The root of process i's tree gets decorated with i's input value. Also in process i's tree, if the node labelled by the string $i_1 \ldots i_k$, $1 \leq k \leq f + 1$, is decorated by a value $v \in V$, then it means that i_k has told i at round k that i_{k-1} has told i_k at round $k - 1$ that ... that i_1 has told i_2 at round 1 that i_1's initial value is v. On the other hand, if the node labelled by the string $i_1 \ldots i_k$ is decorated by *null*, then it means that the chain of communication $i_1, i_2, \ldots, i_k, i$ has been broken by a failure. After $f + 1$ rounds, the processes use their individual decorated trees to decide on a value in V, based on a commonly agreed-upon decision rule (described below). A more detailed description of the algorithm follows.

In this algorithm description and in some others later on, it is convenient to pretend that each process i is able to send messages to itself in addition to the

other processes; this can help to make the algorithm descriptions more uniform. These messages are technically not permitted in the model, but there is no harm in allowing them because the fictional transmissions could just be simulated by local computation.

***EIGStop* algorithm:**

For every string x that occurs as a label of a node of T, each process has a variable $val(x)$. Variable $val(x)$ is used to hold the value with which the process decorates the node labelled x. Initially, each process i decorates the root of its tree with its own initial value, that is, it sets its $val(\lambda)$ to its initial value.

Round 1*:* Process i broadcasts $val(\lambda)$ to all processes, including i itself. Then process i records the incoming information:

1. If a message with value $v \in V$ arrives at i from j, then i sets its $val(j)$ to v.
2. If no message with a value in V arrives at i from j, then i sets $val(j)$ to *null*.

Round k, $2 \leq k \leq f+1$*:* Process i broadcasts all pairs (x, v), where x is a level $k-1$ label in T that does not contain index i, $v \in V$, and $v = val(x)$.[1] Then process i records the incoming information:

1. If xj is a level k node label in T, where x is a string of process indices and j is a single index, and a message saying that $val(x) = v \in V$ arrives at i from j, then i sets $val(xj)$ to v.
2. If xj is a level k node label and no message with a value in V for $val(x)$ arrives at i from j, then i sets $val(xj)$ to *null*.

At the end of $f+1$ rounds, process i applies a decision rule. Namely, let W be the set of non-*null vals* that decorate nodes of i's tree. If W is a singleton set, then i decides on the unique element of W; otherwise, i decides on v_0.

It should not be hard to see that the trees get decorated with the values we indicated earlier. That is, process i's root gets decorated with i's input value. Also, if process i's node labelled by the string $i_1 \ldots i_k$, $1 \leq k \leq f+1$, is decorated by a value $v \in V$, then it must be that i_k has told i at round k that i_{k-1} has told

[1] In order to fit our formal model, in which only one message can be sent from i to each other process at each round, all the messages with the same destination are packaged together into one large message.

i_k at round $k-1$ that ... that i_1 has told i_2 at round 1 that i_1's initial value is v. Moreover, if process i's node labelled by the string $i_1 \ldots i_k$, $1 \leq k \leq f+1$, is decorated by *null*, then it must be that i_k does send a message to i at round k giving a value for $i_1, \ldots, i_{k-1}$.

Example 6.2.1 Execution of *EIGStop*

As an example of how the *EIGStop* algorithm executes, consider the case of three processes ($n = 3$), one of which may be faulty ($f = 1$). Then the protocol executes for 2 rounds, and the tree has 3 levels. The structure of the *EIG* tree $T_{3,1}$ is as in Figure 6.2.

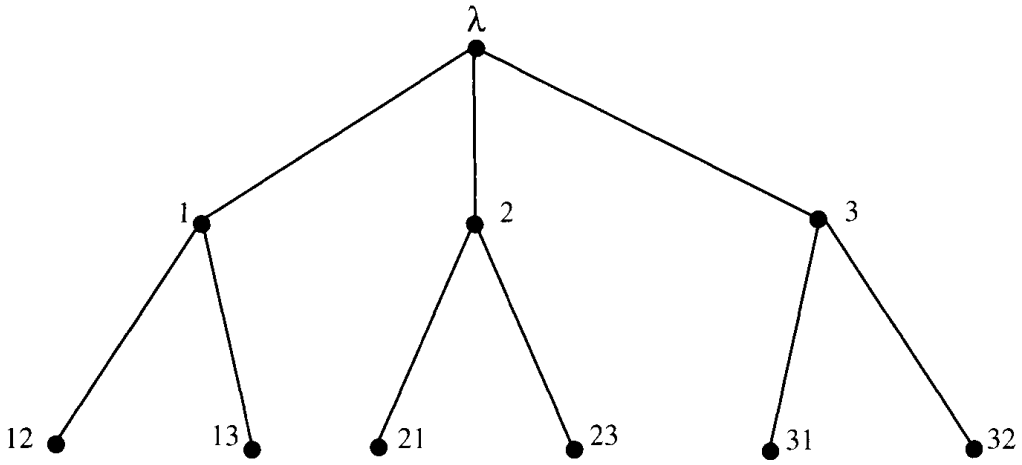

Figure 6.2: Structure of *EIG* tree $T_{3,1}$.

Suppose that processes 1, 2, and 3 have initial values 0, 0, and 1, respectively. Suppose that process 3 is faulty and that it fails after sending its round 1 message to 1 but not to 2. Then the three processes' trees get filled in as in Figure 6.3.

Note that process 2 does not discover that process 3's initial value is 1 until it hears this from process 1 at round 2.

To see that *EIGStop* works correctly, we first give two lemmas that relate the values in the various trees. The first lemma describes the initialization and the relationships between *vals* at different processes at adjacent levels in the trees.

Lemma 6.11 *After $f+1$ rounds of the EIGStop algorithm, the following hold:*

1. *$val(\lambda)_i$ is i's input value.*

2. *If xj is a node label and $val(xj)_i = v \in V$, then $val(x)_j = v$.*

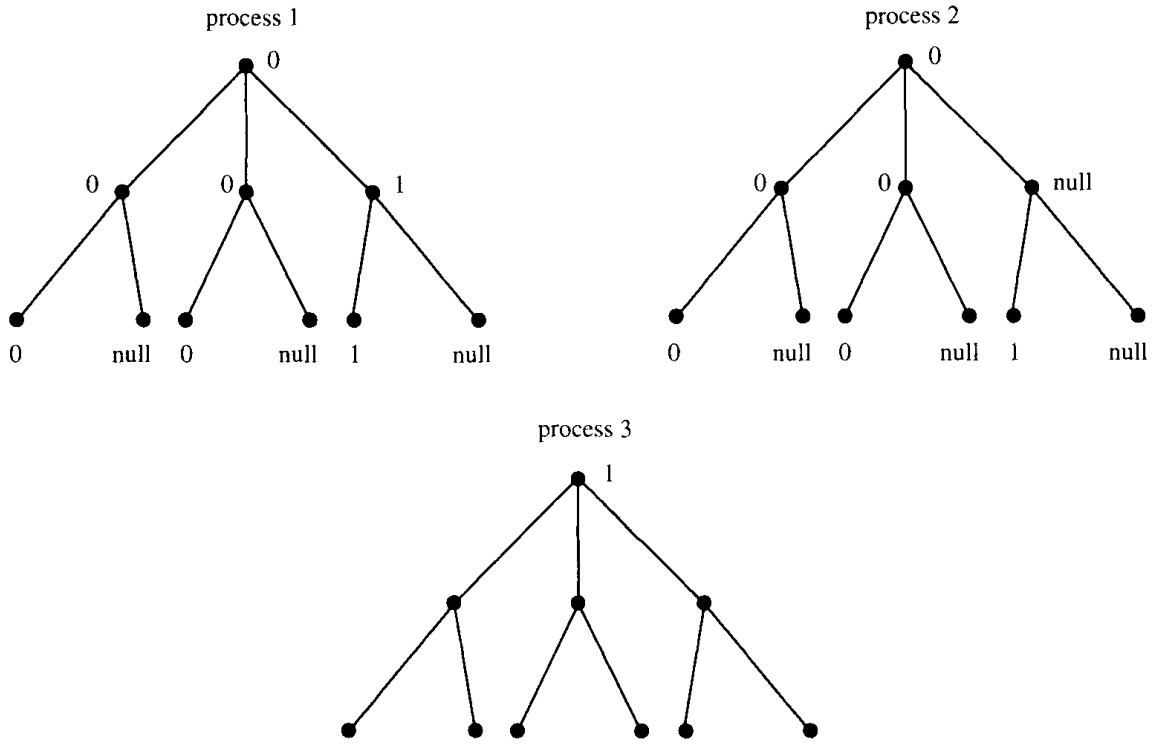

Figure 6.3: Execution of *EIGStop*; process 3 fails at round 1.

3. *If xj is a node label and $val(xj)_i = null$, then either $val(x)_j = null$ or else j fails to send a message to i at round $|x| + 1$.*

Proof. The proof is left as an exercise. □

The second lemma describes the relationship between *vals* at not-necessarily-adjacent levels in the trees. The first two conditions trace the origin of values appearing anywhere in the trees. The third condition is a technical one, asserting that any value v that appears in a tree must appear in that tree at some node whose label does not contain the index i. Loosely speaking, this means that the first time that process i learns a value, it is not as a result of propagating the value to itself.

Lemma 6.12 *After $f + 1$ rounds of the EIGStop algorithm, the following hold.*

1. *If y is a node label, $val(y)_i = v \in V$, and xj is a prefix of y, then $val(x)_j = v$.*

2. *If $v \in V$ appears in the set of vals at any process, then $v = val(\lambda)_i$ for some i.*

3. *If $v \in V$ appears in the set of vals at process i, then there is some label y that does not contain i such that $v = val(y)_i$.*

Proof. Part 1 follows from repeated use of part 2 of Lemma 6.11.

For part 2, suppose that $v = val(y)_i$. If $y = \lambda$, we are done. Otherwise, let j be the first index in y. Part 1 then implies that $v = val(\lambda)_j$.

For part 3, suppose to the contrary that v only appears as the *val* for labels containing i and let y be a shortest label such that $v = val(y)_i$. Then y has a prefix of the form xi. But then part 1 implies that $val(x)_i = v$, which contradicts the choice of y. □

The next lemma provides the key to the agreement property.

Lemma 6.13 *If processes i and j are both nonfaulty, then $W_i = W_j$.*

Proof. We may assume that $i \neq j$. We show inclusion both ways.

1. $W_i \subseteq W_j$.
   Suppose $v \in W_i$. Then Lemma 6.12 implies that $v = val(x)_i$ for some label x that does not contain i. We consider two cases:

   (a) $|x| \leq f$.
   Then $|xi| \leq f + 1$, so since string x does not contain index i, (nonfaulty) process i relays value v to process j at round $|xi|$. This implies that $val(xi)_j = v$, so $v \in W_j$.

   (b) $|x| = f + 1$.
   Then because there are at most f faulty processes and all indices in x are distinct, there must be some nonfaulty process l whose index appears in x. Therefore, x has a prefix of the form yl, where y is a string. Then Lemma 6.12 implies that $val(y)_l = v$. Since process l is nonfaulty, it relays v to process j at round $|yl|$. Therefore, $val(yl)_j = v$, so again $v \in W_j$.

2. $W_j \subseteq W_i$.
   Symmetric to the previous case.

The two cases together imply the needed equality. □

Example 6.2.2 Cases in the proof of Lemma 6.13

Example 6.2.1 illustrates the two cases, (a) and (b), considered in the proof of Lemma 6.13. Process 1 first decorates its tree with a value of 1 at round 1, which is not the last round, so as in case (a), process

2 decorates its tree with 1 by round 2. In particular, $val(3)_1 = 1$, so $val(31)_2 = 1$.

On the other hand, process 2 first decorates its tree with a value of 1 at the last round, round 2, setting $val(31)_2 = 1$. This implies that some nonfaulty process index, in this case 1, must appear in the node label. Then as in case (b), the value 1 appears at node 31 in process 1's tree. That is, $val(31)_2 = 1$, so $val(31)_1 = 1$.

Theorem 6.14 *EIGStop solves the agreement problem for stopping failures.*

Proof. Termination is obvious, by the decision rule.

For validity, suppose that all the initial values are equal to v. Then the only values that ever decorate any process's tree are v and *null*, by Lemma 6.12. Each set W_i is nonempty, since it contains i's initial value. Therefore, each W_i must be exactly equal to $\{v\}$, so the decision rule says that v is the only possible decision.

For agreement, let i and j be any two processes that decide. Since decisions only occur at the end, this means that i and j are nonfaulty. Then Lemma 6.13 implies that $W_i = W_j$. The decision rule then implies that i and j make the same decision. □

Complexity analysis. The number of rounds is $f + 1$, and the number of messages sent is $O((f+1)n^2)$. (This counts each combined message sent by any process to any other at any round as a single message.) The number of *bits* communicated is exponential in the number of failures: $O\left(n^{f+1}b\right)$.

Alternative decision rule. Since *EIGStop* guarantees that the same set W of values appears in the trees of nonfaulty processes, various other decision rules would also work correctly. For instance, if the value set V has a total ordering, then all processes could simply choose the minimum value in W. As before, this has the advantage that it guarantees the stronger validity condition mentioned in Section 6.1.

It is possible to reduce the amount of communication in the *EIGStop* algorithm in much the same way as we did for *FloodSet*. As before, each process i only needs to know the exact elements of its set W_i in case $|W_i| = 1$. So again, it is plausible that the processes might need to broadcast only the first two values they learn about.

***OptEIGStop* algorithm:**

The processes operate as in *EIGStop*, except that each process i broadcasts at most two values altogether. The first broadcast is at round 1, when i

broadcasts its initial value. The second broadcast is at the first round r, $2 \leq r \leq f+1$, such that at the beginning of round r, i knows about some value v different from its initial value (if any such round exists). Then i broadcasts the new value v, together with the label of any level $r-1$ node x that is decorated with v. (If there are two or more possible choices of (x, v), then any one of these may be selected for broadcast.)

As in *EIGStop*, let W be the set of non-*null* *val*s that decorate nodes of i's tree. If W is a singleton set, then i decides on the unique element of W; otherwise, i decides on v_0.

Complexity analysis. *OptEIGStop* uses $f+1$ rounds. The number of messages is at most $2n^2$, since each process sends at most two non-*null* messages to each other process. The number of bits of communication is $O\left(n^2(b+(f+1)\log n)\right)$: the value part of each messages uses $O(b)$ bits, while the label part uses $O((f+1)\log n)$ bits.

The correctness of *OptEIGStop* can be proved by relating it to *EIGStop* using a simulation relation. The proof is similar to the proof of correctness of *OptFloodSet*. Alternatively, a correctness proof that relates *OptEIGStop* to *OptFloodSet* can be given. Details are left for exercises.

6.2.4 Byzantine Agreement with Authentication

Although the *EIG* algorithms described in this section are designed to tolerate stopping failures only, it happens that they can also tolerate some worse types of failures. They cannot cope with the full difficulty of the Byzantine fault model, where processes can exhibit arbitrary behavior. However, they can cope with an interesting restriction on the Byzantine fault model in which processes have the extra power to *authenticate* their communications, based on the use of *digital signatures*. A digital signature for process i is a transformation that i can apply to any of its outgoing messages in order to prove that the message really did originate at i. No other process is able to generate i's signature without i's cooperation. Digital signatures are a reasonable capability to assume in modern communication networks.

We do not provide a formal definition of the Byzantine model with authentication—in fact, we do not know of a nice formal definition—but just describe it informally. In this model, it is assumed that processes can use digital signatures to authenticate any of their outgoing messages. In the literature, it is usually assumed that the initial values originate from some common source, which also signs them; here, we assume that each nonfaulty process starts in an initial state containing a single input value signed by the source, while each faulty process

starts in some state containing some set of input values signed by the source. Faulty processes are permitted to send arbitrary messages and perform arbitrary state transitions; the only limitation is that they are unable to generate signatures of nonfaulty processes or of the source.

The correctness conditions to be satisfied in this model are the usual termination and agreement conditions for Byzantine agreement, plus the following validity condition:

Validity: If all processes start with exactly one initial value $v \in V$, signed by the source, then v is the only possible decision value for a nonfaulty process.

It is not difficult to see that the *EIGStop* and *OptEIGStop* algorithms, modified so that all messages are signed and only correctly signed messages are accepted, solve the agreement problem for the authenticated Byzantine failure model. The proofs are similar to those given for the stopping failure model and are left as exercises.

6.3 Algorithms for Byzantine Failures

In this section, we present algorithms for Byzantine agreement, for the special case of an n-node complete graph. We begin with one that uses exponential information gathering. Then we show how an algorithm that solves Byzantine agreement for a binary value set, $V = \{0, 1\}$, can be used as a "subroutine" for solving Byzantine agreement for a general value set V. Finally, we describe a Byzantine agreement algorithm with reduced communication complexity.

A common property that all these algorithms have is that the number of processes they use is *more than three times* the number of failures, $n > 3f$. This situation is different from what we saw for the stopping failure case, where there were no special requirements on the relationship between n and f. This process bound reflects the added difficulty of the Byzantine fault model. In fact, we will see in Section 6.7 that this bound is inherent. This might seem surprising at first, because you might guess that $2f + 1$ processes could tolerate f Byzantine faults, using some sort of majority voting algorithm. (There is a standard fault-tolerance technique known as *triple-modular redundancy*, in which a task is triplicated and the majority result accepted; you might think that this method could be used to solve Byzantine agreement for one faulty process, but you will see that it cannot.)

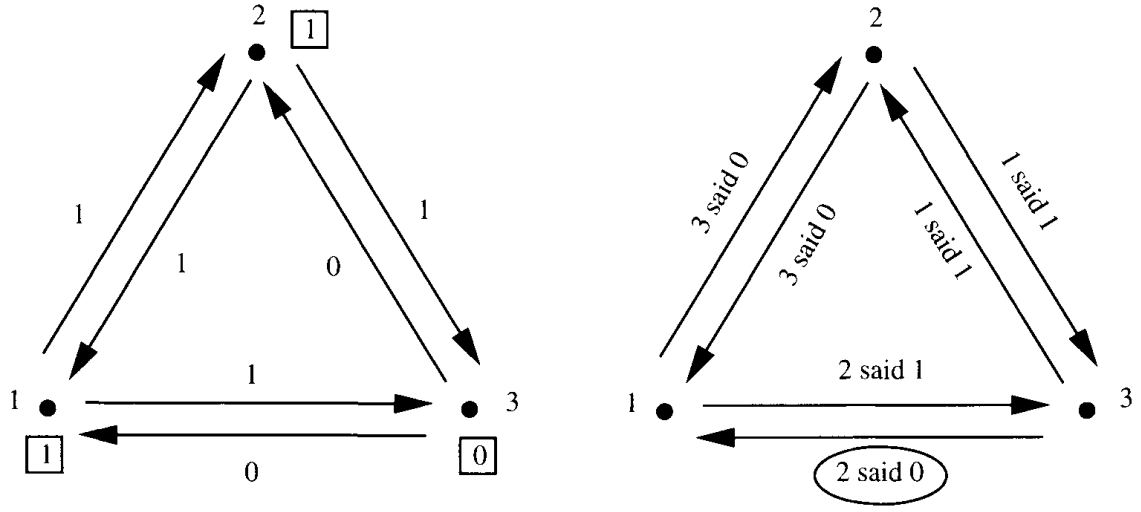

Figure 6.4: Execution α_1—false message is circled.

6.3.1 An Example

Before presenting the *EIG* Byzantine agreement algorithm, we give an idea of why the Byzantine agreement problem is more difficult than the agreement problem for stopping failures. Specifically, we give an example suggesting (though not proving) that three processes cannot solve Byzantine agreement, if there is the possibility that even one of them might be faulty.

Suppose that processes 1, 2, and 3 solve the Byzantine agreement problem, tolerating one fault. Suppose, for example, that they decide at the end of two rounds and that they operate in a particular, constrained manner: at the first round, each process simply broadcasts its initial value, while in the second round, each process reports to each other process what was told to it in the first round by the third process. Consider the following execution.

> *Execution* α_1*:*
> Processes 1 and 2 are nonfaulty and start with initial values of 1, while process 3 is faulty and starts with an initial value of 0. In the first round, all processes report their values truthfully. In the second round, processes 1 and 2 report truthfully what they heard in the first round, while process 3 tells 1 (falsely) that 2 sent 0 in round 1 and otherwise behaves truthfully. Figure 6.4 shows the interesting messages that are sent in α_1. In this execution, the validity condition requires that processes 1 and 2 both decide 1.

Now consider a second execution.

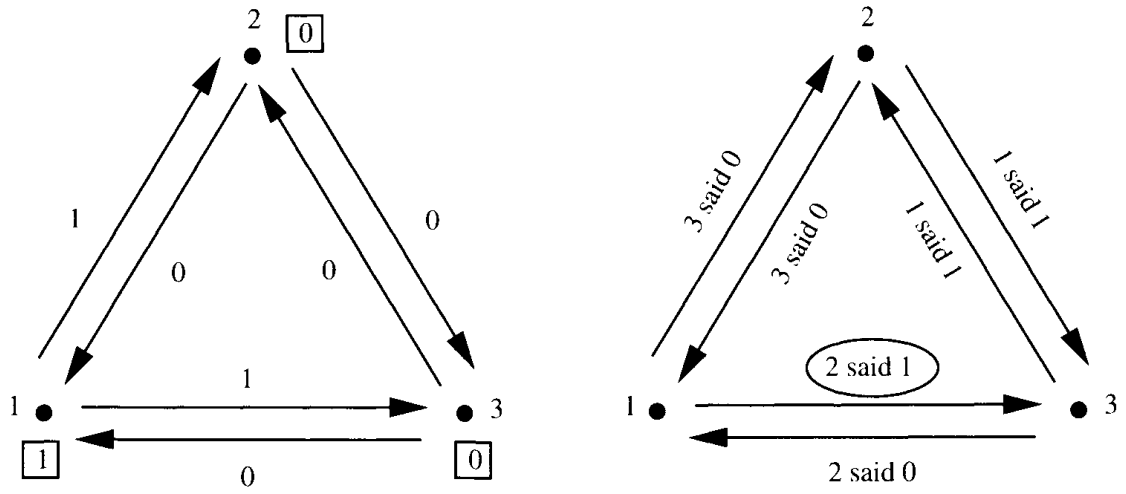

Figure 6.5: Execution α_2—false message is circled.

Execution α_2:
This is symmetric to α_1. This time, processes 2 and 3 are nonfaulty and start with initial values of 0, while process 1 is faulty and starts with an initial value of 1. In the first round, all processes report their values truthfully. In the second round, processes 2 and 3 report truthfully what they heard in the first round, while process 1 tells 3 (falsely) that 2 sent 1 in round 1 and otherwise behaves truthfully. Figure 6.5 shows the interesting messages that are sent in α_2. In this execution, the validity condition requires that processes 2 and 3 both decide 0.

To get a contradiction, consider a third execution.

Execution α_3:
Now suppose that processes 1 and 3 are nonfaulty and start with 1 and 0, respectively. Process 2 is faulty, telling 1 that its initial value is 1 and telling 3 that its initial value is 0. All processes behave truthfully in the second round. The situation is shown in Figure 6.6.

Notice that process 2 sends the same messages to 1 in α_3 as it does in α_1, and sends the same messages to 3 in α_3 as it does in α_2, in both rounds. In fact, it is easy to check that α_3 and α_1 are indistinguishable to process 1, $\alpha_3 \overset{1}{\sim} \alpha_1$, and similarly $\alpha_3 \overset{3}{\sim} \alpha_2$. Since process 1 decides 1 in α_1, it also does so in α_3, and since process 3 decides 0 in α_2, it also does so in α_3. But this violates the agreement condition for α_3, which contradicts the assumption that processes 1, 2, and 3 solve the Byzantine agreement problem. We have shown that no algorithm of this particularly simple form can solve Byzantine agreement.

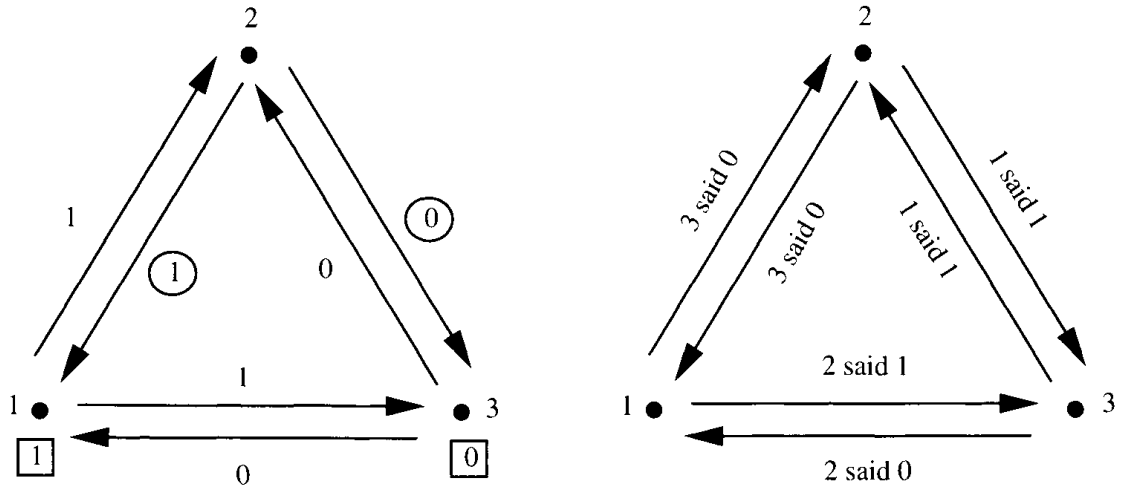

Figure 6.6: Execution α_3—conflicting messages are circled.

Note that process 1, for example, can tell that some process is faulty in α_3, since process 2 tells 1 that its value is 1, but process 3 tells 1 that 2 said its value is 0. The problem is that process 1 is unable to tell which of 2 and 3 is faulty.

This example does not constitute a proof that three processes cannot solve Byzantine agreement with the possibility of a single fault. This is because the argument presupposes that the algorithm uses only two rounds and sends particular types of messages. But it is possible to extend the example to more rounds and arbitrary types of messages. In fact, as we will see in Section 6.4, the ideas can be extended to show that $n > 3f$ processes are needed to solve Byzantine agreement in the presence of f faults.

6.3.2 EIG Algorithm for Byzantine Agreement

We now give an *EIG* algorithm for Byzantine agreement, which we call *EIGByz*. Unlike the *EIGStop* algorithm, *EIGByz* presupposes that the number of processes is large relative to the number of faults, in particular, that $n > 3f$. This is necessary because of the limitations described in Sections 6.3.1 and 6.4. Before you read about this algorithm, we suggest that you try to construct an algorithm of your own for a special case, say $n = 7$ and $f = 2$.

The *EIGByz* algorithm for n processes with f faults uses the same *EIG* tree data structure, $T_{n,f}$, that is used in *EIGStop*. Essentially the same propagation strategy is used as for *EIGStop*; the only difference is that a process that receives an "ill-formed" message corrects the information to make it look sensible. The decision rule is quite different, however—it is no longer the case that a process

can trust all values that appear anywhere in its tree. Now processes must take some action to mask values that arrive in false messages.

***EIGByz* algorithm:**

The processes propagate values for $f+1$ rounds exactly as in the *EIGStop* algorithm, with the following exceptions. If a process i ever receives a message from another process j that is not of the specified form (e.g., it contains complete garbage or contains duplicate values for the same node in j's tree), then i "throws away" the message, that is, acts just as if process j did not send it anything at that round.

At the end of $f+1$ rounds, process i adjusts its *val* assignment so that any *null* value is replaced by the default value v_0.

Then to determine its decision, process i works from the leaves up in its adjusted, decorated tree, decorating each node with an additional *newval*, as follows. For each leaf labelled x, $newval(x) := val(x)$. For each non-leaf node labelled x, $newval(x)$ is defined to be the *newval* held by a strict majority of the children of node x, that is, the element $v \in V$ such that $newval(xj) = v$ for a majority of the nodes of the form xj, provided that such a majority exists. If no majority exists, process i sets $newval(x) := v_0$. Process i's final decision is $newval(\lambda)$.

To show the correctness of *EIGByz*, we start with some preliminary assertions. The first says that all nonfaulty processes agree on the values relayed directly from nonfaulty processes.

Lemma 6.15 *After $f+1$ rounds of the EIGByz algorithm, the following holds. If i, j, and k are all nonfaulty processes, with $i \neq j$, then $val(x)_i = val(x)_j$ for every label x ending in k.*

Proof. If $k \notin \{i, j\}$, then the result follows from the fact that, since k is nonfaulty, it sends the same message to i and j at round $|x|$. If $k \in \{i, j\}$, then the result follows similarly from the convention by which each process relays values to itself. □

The next lemma asserts that all nonfaulty processes agree on the *newval*s computed for nodes whose labels end with nonfaulty process indices.

Lemma 6.16 *After $f+1$ rounds of the EIGByz algorithm, the following holds. Suppose that x is a label ending with the index of a nonfaulty process. Then there is a value $v \in V$ such that $val(x)_i = newval(x)_i = v$ for all nonfaulty processes i.*

Proof. By induction on the tree labels, working from the leaves up—that is, from those of length $f+1$ down to those of length 1.

Basis: Suppose x is a leaf, that is, that $|x| = f+1$. Then Lemma 6.15 implies that all nonfaulty processes i have the same $val(x)_i$; call this common value v. Then also $newval(x)_i = v$ for every nonfaulty process i, by the definition of *newval* for leaves. So v is the required value.

Inductive step: Suppose $|x| = r$, $1 \leq r \leq f$. Then Lemma 6.15 implies that all nonfaulty processes i have the same $val(x)_i$; call this value v. Therefore, every nonfaulty process l sends the same value v for x to all processes, at round $r+1$, so $val(xl)_i = v$ for all nonfaulty i and l. Then the inductive hypothesis implies that also $newval(xl)_i = v$ for all nonfaulty processes i and l.

We now claim that a majority of the labels of children of node x end in nonfaulty process indices. This is true because the number of children of x is exactly $n - r \geq n - f$. Since we have assumed that $n > 3f$, this number must be strictly greater than $2f$. Since at most f of the children have labels ending in indices of faulty processes, we have the needed majority.

It follows that for any nonfaulty i, $newval(xl)_i = v$ for a majority of children xl of node x. Then the majority rule used in the algorithm implies that $newval(x)_i = v$ for all nonfaulty i. So v is the required value. □

We now argue validity.

Lemma 6.17 *If all nonfaulty processes begin with the same initial value $v \in V$, then v is the only possible decision value for a nonfaulty process.*

Proof. If all nonfaulty processes begin with v, then all nonfaulty processes broadcast v at the first round, and therefore $val(j)_i = v$ for all nonfaulty processes i and j. Lemma 6.16 implies that $newval(j)_i = v$ for all nonfaulty i and j. Then the majority rule used in the algorithm implies that $newval(\lambda)_i = v$ for all nonfaulty i. Therefore, i's decision is v, as needed. □

To show the agreement property, we need two more definitions. First, we say that a subset C of the nodes of a rooted tree is a *path covering* provided that every path from the root to a leaf contains at least one node in C.

Second, consider any execution α of the *EIGByz* algorithm. A tree node x is said to be *common* in α provided that at the end of $f+1$ rounds in α, all the nonfaulty processes i have the same $newval(x)_i$. A set of tree nodes (e.g., a path covering) is said to be *common* in α if all the nodes in the set are common in α. Notice that Lemma 6.16 implies that if i is nonfaulty, then for every x, xi is a common node.

Lemma 6.18 *After $f+1$ rounds of any execution α of EIGByz, there exists a path covering that is common in α.*

Proof. Let C be the set of nodes of the form xi, where i is nonfaulty. As observed just above, all nodes in C are common. To see why C is a path covering, consider any path from the root to a leaf. It contains exactly $f+1$ non-root nodes, and each such node ends with a distinct process index, by construction of T. Since there are at most f faulty processes, there is some node on the path whose label ends in a nonfaulty process index. This node must be in C. □

The following lemma shows how common nodes propagate up the tree.

Lemma 6.19 *After $f+1$ rounds of EIGByz, the following holds. Let x be any node label in the EIG tree. If there is a common path covering of the subtree rooted at x, then x is common.*

Proof. By induction on tree labels, working from the leaves up.

Basis: Suppose that x is a leaf. Then the only path covering of x's subtree consists of the single node x itself. So x is common, as needed.

Inductive step: Suppose that $|x| = r$, $0 \leq r \leq f$. Suppose that there is a common path covering C of x's subtree. If x itself is in C, then x is common and we are done, so suppose $x \notin C$.

Consider *any* child xl of x. Since $x \notin C$, C induces a common path covering for the subtree rooted at xl. So by the inductive hypothesis, xl is common. Since xl was chosen to be an arbitrary child of x, all the children of x are common. Then the definition of $newval(x)$ implies that x is common. □

As a simple consequence, we obtain

Lemma 6.20 *After $f+1$ rounds of EIGByz, the root node λ is common.*

Proof. Immediate by Lemmas 6.18 and 6.19. □

We now tie the pieces together in the main correctness theorem.

Theorem 6.21 *EIGByz solves the Byzantine agreement problem for n processes with f failures, if $n > 3f$.*

Proof. Termination is obvious. Validity follows from Lemma 6.17. Agreement follows from Lemma 6.20 and the decision rule. □

Complexity analysis. The costs are the same as for the *EIGStop* algorithm: $f+1$ rounds, $O\left((f+1)n^2\right)$ messages, and $O\left(n^{f+1}b\right)$ bits of communication. In addition, there is the new requirement that the number of processes be large relative to the number of failures: $n > 3f$.

6.3.3 General Byzantine Agreement Using Binary Byzantine Agreement

In this subsection, we show how to use an algorithm that solves Byzantine agreement for inputs in $\{0,1\}$ as a subroutine for solving general Byzantine agreement. The overhead is just 2 extra rounds, $2n^2$ extra messages, and $O\left(n^2b\right)$ bits of communication. This can lead to a substantial savings in the total number of bits that need to be communicated, since it is not necessary to send values in V, but only binary values, while executing the subroutine. This improvement, however, is not sufficient to reduce the number of bits of communication from exponential to polynomial in f.

We call the algorithm *TurpinCoan*, after its designers. The algorithm assumes that $n > 3f$. As earlier, we pretend that each process can send messages to itself as well as to the other processes.

***TurpinCoan* algorithm:**

Each process has local variables x, y, z, and *vote*, where x is initialized to the process's input value and y, z, and *vote* are initialized arbitrarily.

Round 1*:* Process i sends its value of x to all processes, including itself. If, in the set of messages received at this round, there are $\geq n-f$ copies of a particular value $v \in V$, then i sets $y := v$; otherwise $y := null$.

Round 2*:* Process i sends its value of y to all processes, including itself. If, in the set of messages received at this round, there are $\geq n-f$ copies of a particular value in V, then i sets $vote := 1$; otherwise $vote := 0$. Also, i sets z equal to the non-*null* value that occurs most often among the messages received by i at this round, with ties broken arbitrarily; if all messages are *null*, then z remains undefined.

Round r, $r \geq 3$*:* The processes run the binary Byzantine agreement subroutine using the values of *vote* as the input values. If process i decides 1 in the subroutine and if z is defined, then the final decision of the algorithm is z; otherwise it is the default value v_0.

A key fact about the *TurpinCoan* algorithm is

Lemma 6.22 *There is at most one value $v \in V$ that is sent in round 2 messages by nonfaulty processes.*

Proof. Suppose for the sake of contradiction that nonfaulty processes i and j send round 2 messages containing values v and w respectively, where $v, w \in V$, $v \neq w$. Then i receives at least $n - f$ round 1 messages containing v. Since there are at most f faulty processes, and nonfaulty processes send the same round 1 messages to all processes, it must be that j receives at least $n - 2f$ messages containing v. Since $n > 3f$, this means j receives at least $f + 1$ messages containing v.

But also, since j sends w in round 2, j receives at least $n - f$ round 1 messages containing w, for a total of at least $(f + 1) + (n - f) > n$ messages. But the total number of round 1 messages received by j is only n, so this is a contradiction. □

Theorem 6.23 *The TurpinCoan algorithm solves general Byzantine agreement when given a binary Byzantine agreement algorithm as a subroutine, if* $n > 3f$.

Proof. Termination is easy to see.

To show validity, we must prove that if all nonfaulty processes start with the same initial value, v, then all nonfaulty processes decide v. So suppose that all nonfaulty processes start with v. Then all the $\geq n - f$ nonfaulty processes successfully broadcast round 1 messages containing v to all processes. So at round 1, all nonfaulty processes set their y variables to v. Then in round 2, each nonfaulty process receives at least $n - f$ messages containing v, which implies that it sets its z variable to v and its *vote* to 1. Since all the nonfaulty processes use input 1 for the binary Byzantine agreement subroutine, they all decide 1 in the subroutine, by the validity condition for the binary algorithm. This means that they all decide v in the main algorithm, which shows validity.

Finally, we show agreement. If the subroutine's decision value is 0, then v_0 is chosen as the final decision value by all nonfaulty processes and agreement holds by default.

So assume that the subroutine's decision value is 1. Then by the validity condition for the subroutine, some nonfaulty process i must begin the subroutine with $vote_i = 1$. This means that process i receives at least $n - f$ round 2 messages containing some particular value $v \in V$, so since there are at most f faulty processes, i receives at least $n - 2f$ round 2 messages containing v from nonfaulty processes. Then if j is any nonfaulty process, it must be that j also receives at least $n - 2f$ round 2 messages containing v from those same nonfaulty processes. By Lemma 6.22, no value in V other than v is sent by any nonfaulty process in round 2. So process j receives no more than f round 2 messages containing values in V other than v (and these must be from faulty processes). Since $n > 3f$, we have $n - 2f > f$, so v is the value that occurs most often in round 2 messages received by j. It follows that process j sets $z := v$ in round 2.

Since the subroutine's decision value is 1, this means that j decides v. Since this argument holds for any nonfaulty process j, agreement holds. □

In the proof of the *TurpinCoan* algorithm, the limitation of f on the number of faulty processes is used to obtain claims about the similarity between the views of different processes in an execution. This sort of argument also appears in proofs for other consensus algorithms, for instance the *approximate agreement* algorithm in Section 7.2.

Complexity analysis. The number of rounds is $r + 2$, where r is the number of rounds used by the binary Byzantine agreement subroutine. The extra communication used by *TurpinCoan*, in addition to that used by the subroutine, is $2n^2$ messages, each of at most b bits, for a total of $O(n^2 b)$ bits.

6.3.4 Reducing the Communication Cost

Although the *TurpinCoan* algorithm can be used to reduce the bit communication complexity of Byzantine agreement somewhat, its cost is still exponential in the number f of failures. Algorithms that are polynomial in the number of failures are much more difficult to obtain in the Byzantine failure model than in the stopping failure model. In this section, we present one example; this algorithm is not optimal in terms of time complexity, but it is fairly simple and uses some interesting techniques. This algorithm is for the special case of Byzantine agreement on a value in $\{0, 1\}$; the results of Section 6.3.3 show how this algorithm can be used to obtain a polynomial algorithm for a general value domain.

The algorithm uses a mechanism known as *consistent broadcast* for all its communication. This mechanism is a way of ensuring a certain amount of coherence among the messages received by different processes. Using consistent broadcast, a process i can *broadcast* a message of the form (m, i, r) at round r, and the message can be *accepted* by any of the processes (including i itself) at any subsequent round. The consistent broadcast mechanism is required to satisfy the following three conditions:

1. If nonfaulty process i broadcasts message (m, i, r) in round r, then the message is accepted by all nonfaulty processes by round $r + 1$ (i.e., it is either accepted at round r or round $r + 1$).

2. If nonfaulty process i does not broadcast message (m, i, r) in round r, then (m, i, r) is never accepted by any nonfaulty process.

3. If any message (m, i, r) is accepted by any nonfaulty process j, say at round r', then it is accepted by all nonfaulty processes by round $r' + 1$.

The first condition says that nonfaulty processes' broadcasts are accepted quickly, while the second says that no messages are ever falsely attributed to nonfaulty processes. The third condition says that any message that is accepted by a nonfaulty process (whether from a faulty or nonfaulty sender) must also be accepted by every other nonfaulty process soon thereafter.

The consistent broadcast mechanism can be implemented easily.

***ConsistentBroadcast* algorithm:**

In order to broadcast (m, i, r) at round r, process i sends a message $(\text{"}init\text{"}, m, i, r)$ to all processes at round r. If process j receives an $(\text{"}init\text{"}, m, i, r)$ message from process i at round r, it sends $(\text{"}echo\text{"}, m, i, r)$ to all processes at round $r + 1$.

If, before any round $r' \geq r + 2$, process j has received $(\text{"}echo\text{"}, m, i, r)$ messages from at least $f + 1$ processes, then j sends an $(\text{"}echo\text{"}, m, i, r)$ message at round r' (if it has not already done so).

If, by the end of any round $r' \geq r+1$, process j has received $(\text{"}echo\text{"}, m, i, r)$ messages from at least $n - f$ processes, then j accepts the communication at round r' (if it has not already done so).

Theorem 6.24 *The ConsistentBroadcast algorithm solves the consistent broadcast problem, if* $n > 3f$.

Proof. We verify the three properties.

1. Suppose that nonfaulty process i broadcasts message (m, i, r) at round r. Then i sends $(\text{"}init\text{"}, m, i, r)$ to all processes at round r, and each of the $\geq n-f$ nonfaulty processes sends $(\text{"}echo\text{"}, m, i, r)$ to all processes at round $r + 1$. Then, by the end of round $r + 1$, each nonfaulty process receives $(\text{"}echo\text{"}, m, i, r)$ messages from at least $n - f$ processes and so accepts the message.

2. If nonfaulty process i does not broadcast message (m, i, r) in round r, then it sends no $(\text{"}init\text{"}, m, i, r)$ messages, so no nonfaulty process ever sends an $(\text{"}echo\text{"}, m, i, r)$ message. Then no nonfaulty process ever accepts the message, because acceptance requires receipt of *echo* messages from at least $n - f > f$ processes.

3. Suppose message (m, i, r) is accepted by nonfaulty process j at round r'. Then j receives ("*echo*", m, i, r) messages from at least $n - f$ processes by round r'. Among these $n - f$ processes, there are at least $n - 2f \geq f + 1$ nonfaulty processes. Since nonfaulty processes send the same messages to all processes, every nonfaulty process receives at least $f+1$ ("*echo*", m, i, r) messages by round r'. This implies that every nonfaulty process sends an ("*echo*", m, i, r) message by round $r' + 1$, so that every process receives at least $n - f$ ("*echo*", m, i, r) messages by round $r' + 1$. Therefore, the message is accepted by all nonfaulty processes by round $r' + 1$.

□

Complexity analysis. The consistent broadcast of a single message uses $O(n^2)$ messages.

Now we describe a simple binary Byzantine agreement algorithm that uses consistent broadcast for all its communication. Called the *PolyByz algorithm*, it only sends around information about initial values of 1. It uses increasing thresholds for broadcasting messages.

***PolyByz* algorithm:**

The algorithm operates in $f + 1$ stages, where each stage consists of two rounds. The messages that are sent (using consistent broadcast) are all of the form $(1, i, r)$, where i is a process index and r is an odd round number. That is, messages are only sent at the first rounds of stages, and the only information ever sent is just the value 1.

The conditions under which process i broadcasts a message are as follows. At round 1, i broadcasts a message $(1, i, 1)$ exactly if i's initial value is 1. At round $2s - 1$, the first round of stage s, where $2 \leq s \leq f + 1$, i broadcasts a message $(1, i, 2s - 1)$ exactly if i has accepted messages from at least $f + s - 1$ different processes before round $2s - 1$ and i has not yet broadcast a message.

At the end of $2(f + 1)$ rounds, process i decides on 1 exactly if i has accepted messages from at least $2f + 1$ different processes by the end of round $2(f + 1)$. Otherwise, i decides 0.

Theorem 6.25 *PolyByz solves the binary Byzantine agreement problem, if* $n > 3f$.

Proof. Termination is obvious.

For validity, there are two cases. First, if all nonfaulty processes start with initial value 1, then at least $n - f \geq 2f + 1$ processes broadcast at round 1. By

property 1 of consistent broadcast, all nonfaulty processes accept these messages by round 2, so that each nonfaulty process accepts messages from at least $2f+1$ different processes by the end of round 2. This is sufficient to imply that each nonfaulty process decides 1.

On the other hand, if all nonfaulty processes start with initial value 0, then no nonfaulty process ever broadcasts. This is because the minimum number of acceptances needed to trigger a broadcast is $f+1$, which is impossible to achieve without a prior broadcast by a nonfaulty process. (We are using property 2 of consistent broadcast here.) This implies that each nonfaulty process decides 0.

Finally, we argue agreement. Suppose that nonfaulty process i decides 1; it is enough to show that every other nonfaulty process also decides 1. Since i decides 1, i must accept messages from at least $2f+1$ different processes by the end of round $2(f+1)$. Let I be the set of nonfaulty processes among these; then $|I| \geq f+1$.

If all the processes in I have initial values of 1, then they broadcast at round 1, and, by property 1 of consistent broadcast, all nonfaulty processes accept these messages by round 2. Then before round 3,[2] each nonfaulty process has accepted messages from at least $f+1$ different processes, which is enough to trigger it to broadcast at round 3; again by property 1 of consistent broadcast, all nonfaulty processes accept these messages by round 4. Thus, each nonfaulty process accepts messages from at least $n-f \geq 2f+1$ different processes by the end of round 4, and so decides 1, as needed.

On the other hand, suppose that one of the processes in I, say j, does not have an initial value of 1. Then it must be that j broadcasts at some round $2s-1$, where $2 \leq s \leq f+1$, which means that j accepts messages from at least $f+s-1$ different processes before round $2s-1$; moreover, none of these messages is from j itself. Then by property 3 of consistent broadcast, messages from all of these $f+s-1$ processes get accepted by all nonfaulty processes by the end of round $2s-1$, and, by property 1, the message broadcast by j gets accepted by all nonfaulty processes by the end of round $2s$. It follows that each nonfaulty process accepts messages from at least $(f+s-1)+1 = f+s$ different processes by the end of round $2s$.

Now there are two cases. If $s = f+1$, then each nonfaulty process accepts messages from at least $2f+1$ different processes by the end of round $2(f+1)$, which is enough to ensure that they all decide 1. On the other hand, if $s \leq f$, then every nonfaulty process accepts sufficiently many messages before round $2s+1$ to broadcast at round $2s+1$, if it has not done so already. Then by property 1 of consistent broadcast, all nonfaulty processes accept messages from

[2]We assume that $f \geq 1$, so that there actually is a round 3.

all the nonfaulty processes by the end of round $2s + 2$. Again, this is enough to ensure that they all decide 1, as needed. □

Complexity analysis. *PolyByz* requires $2f + 2$ rounds. There are at most n broadcasts, each requiring $O(n^2)$ messages; thus, the number of messages is $O(n^3)$. The number of bits in each message is $O(\log n)$, because messages contain process indices. Thus, the total bit complexity is just $O(n^3 \log n)$.

Relationship with the authenticated Byzantine failure model. Adding a consistent broadcast capability to the ordinary Byzantine model produces a model that is somewhat like the authenticated Byzantine failure model discussed informally in Section 6.2.4. However, the two are not exactly the same. For instance, consistent broadcast is just for broadcasting, not for sending individualized messages. More significantly, consistent broadcast does not prevent a process i from broadcasting a message saying (falsely) that a nonfaulty process j has previously sent a particular message; the nonfaulty processes will all accept this message, even though its contents represent a false claim. In the authenticated Byzantine failure model, the use of digital signatures allows processes to reject such messages immediately. However, even though the models are somewhat different, the consistent broadcast capability is strong enough that it can be used to implement, in the ordinary Byzantine model, some algorithms designed for the authenticated Byzantine failure model.

6.4 Number of Processes for Byzantine Agreement

We have presented algorithms to solve the agreement problem in a complete network graph, in the presence of stopping failures, and even in the presence of Byzantine failures. You have probably noticed that these algorithms are quite costly. For stopping failures, the best algorithm we gave was the *OptFloodSet* algorithm, which requires $f + 1$ rounds, $2n^2$ messages, and $O(n^2 b)$ bits of communication. For the Byzantine case, the *EIGByz* algorithm uses $f+1$ rounds and an exponential amount of communication, while *PolyByz* uses $2(f+1)$ rounds and a polynomial amount of communication. Both Byzantine agreement algorithms also require $n > 3f$.

In the rest of this chapter, we show that these high costs are not accidental. First, in this section, we show that the $n > 3f$ restriction is needed for any solution to the Byzantine agreement problem. The next two sections contain related results: Section 6.5 describes exactly the amount of connectivity that is needed in an incomplete network graph in order for Byzantine agreement to

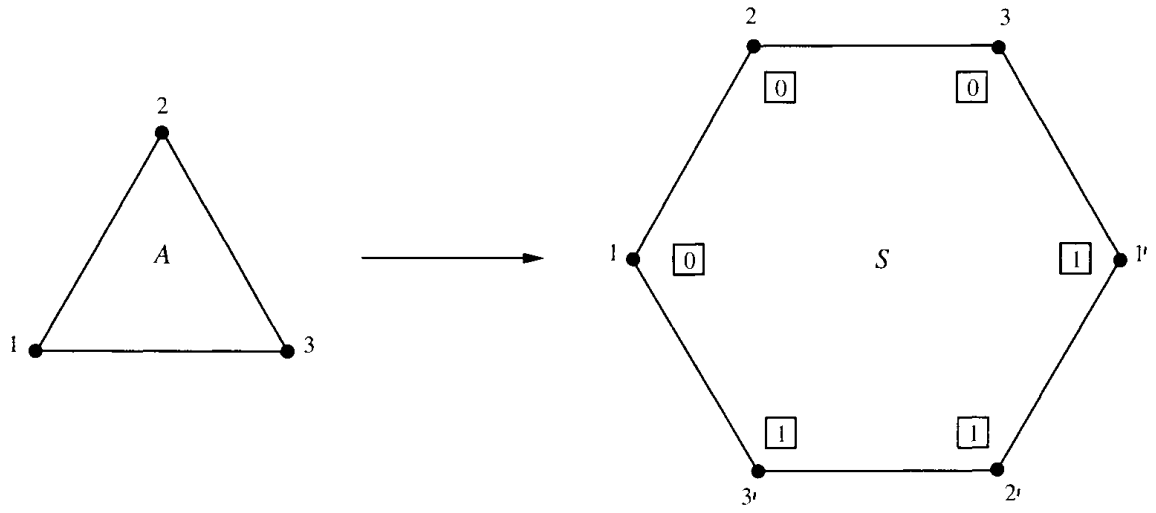

Figure 6.7: Combining two copies of A to get S.

be solvable, while Section 6.6 shows that the $n > 3f$ bound extends to weaker problem statements than Byzantine agreement. The final section of the chapter shows that the lower bound of $f + 1$ on the number of rounds is also necessary, even for the simple case of stopping failures.

In order to prove that $n \leq 3f$ processes cannot solve Byzantine agreement in the presence of f faults, we begin by showing the simplest special case: that three processes cannot solve Byzantine agreement with the possibility of one fault. This result is suggested by the example in Section 6.3.1, although that example does not constitute a proof. We then show the general result, for arbitrary n and f, $n \leq 3f$, by "reducing" the problem to the case of three versus one.

Lemma 6.26 *Three processes cannot solve Byzantine agreement in the presence of one fault.*

Proof. By contradiction. Assume there is a three-process algorithm A that solves the Byzantine agreement problem for the three processes 1, 2, and 3, even if one of these three may be faulty. We construct a new system S using *two copies* of A and show that S must exhibit contradictory behavior. It follows that the assumed algorithm A cannot exist.

Specifically, we take two copies of each process in A and configure them into a single hexagonal system S. We start one copy each of processes 1, 2, and 3 (the unprimed copy) with input value 0, and the other (the primed copy) with input value 1. The arrangement is shown in Figure 6.7.

What is system S, formally? It is a synchronous system, based on a hexagonal network graph, within the general model of Chapter 2. Note that it is *not* a

system that is supposed to solve the Byzantine agreement problem—we don't care what it does, in fact, only that it is a synchronous system of some kind. We will not consider any faulty process behavior in S.

Remember that in the systems we consider as solutions for the Byzantine agreement problem, we assume that the processes all "know" the entire network graph. For example, in A, process 1 knows the names 2 and 3 and presumes that there are exactly three nodes, named 1, 2, and 3, arranged in a triangle. In S, we do not assume that the processes know the entire (hexagonal) network graph, but rather that each process just has local names for its neighbors. For example, in S, process 1 knows that it has two neighbors, which it knows by the names 2 and 3, even though one of them is really $3'$. It does not know that there are duplicate copies of the nodes in the network. The situation is similar to the one considered in Chapter 4, where each process only had local knowledge of its portion of the network graph. In particular, notice that the network in S appears to each process just like the network in A.

System S is not required to exhibit any special type of behavior. However, note that S with any particular input assignment does exhibit *some* well-defined behavior. We will obtain a contradiction by showing that, for the particular input assignment indicated above, no such well-defined behavior is possible.

So suppose that the processes in S are started with the input values indicated in Figure 6.7, that is, the unprimed processes with 0 and the primed processes with 1; let α be the resulting execution of S.

We first consider execution α from the point of view of processes 2 and 3. To processes 2 and 3, it appears as if they are running in the triangle system A, in an execution α_1 in which process 1 is faulty. That is, α and α_1 are indistinguishable to processes 2 and 3, $\alpha \overset{2}{\sim} \alpha_1$ and $\alpha \overset{3}{\sim} \alpha_1$, according to the definition of "indistinguishable" in Section 2.4. See Figure 6.8. In α_1, process 1 exhibits a peculiar type of faulty behavior—it behaves like the combination of processes $1'$, $2'$, $3'$, and 1 in α. Although it is peculiar, it is an allowable behavior for a faulty process in A, under the assumptions for Byzantine faults.

Since α_1 is an execution of A in which only process 1 is faulty and processes 2 and 3 begin with input 0, and since A is assumed to solve Byzantine agreement, the correctness conditions for Byzantine agreement imply that eventually in α_1, processes 2 and 3 must decide 0. Since α is indistinguishable from α_1 to processes 2 and 3, both decide 0 in α as well.

Next consider execution α from the point of view of processes $1'$ and $2'$. To processes $1'$ and $2'$, it appears as if they are running in the triangle system A, in an execution α_2 in which process 3 is faulty. That is, $\alpha \overset{1'}{\sim} \alpha_2$ and $\alpha \overset{2'}{\sim} \alpha_2$.

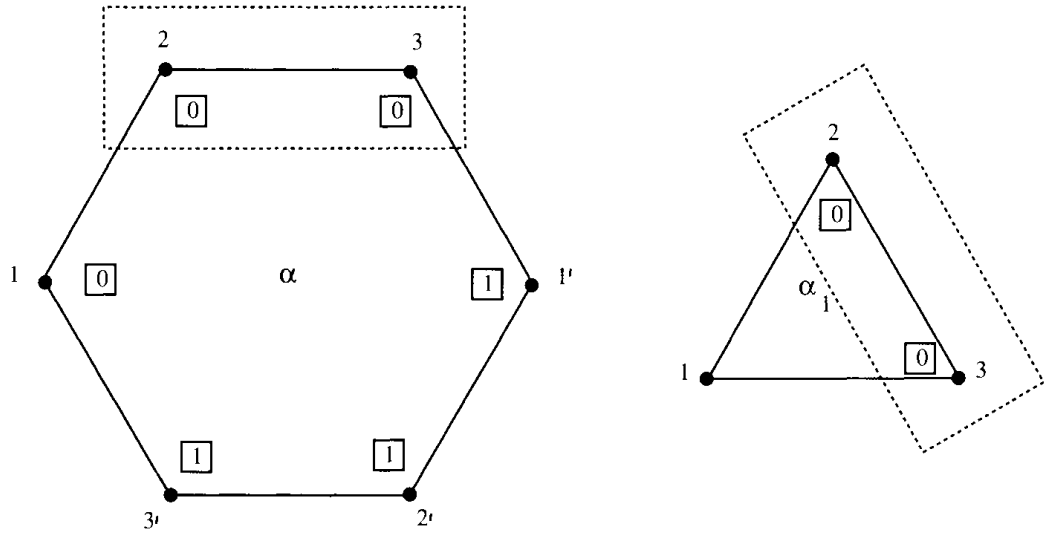

Figure 6.8: Executions α and α_1 are indistinguishable to processes 2 and 3.

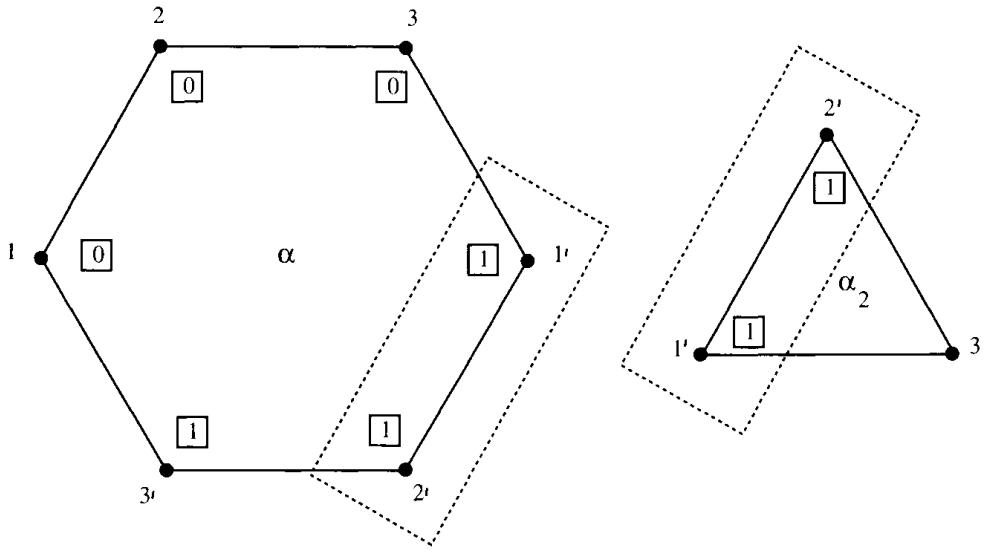

Figure 6.9: Executions α and α_2 are indistinguishable to processes $1'$ and $2'$.

See Figure 6.9. By the same argument as above, processes $1'$ and $2'$ eventually decide 1 in α.

Finally, consider execution α from the point of view of processes 3 and $1'$. To processes 3 and $1'$, it appears as if they are running in the triangle system A, in an execution α_3 in which process 2 is faulty. That is, $\alpha \overset{3}{\sim} \alpha_3$ and $\alpha \overset{1'}{\sim} \alpha_3$. See Figure 6.10. By the correctness conditions for Byzantine agreement, processes 3 and $1'$ must eventually decide in α_3, and their decisions must be the same. Because process 3 starts with input 0 and process $1'$ starts with input 1, there is

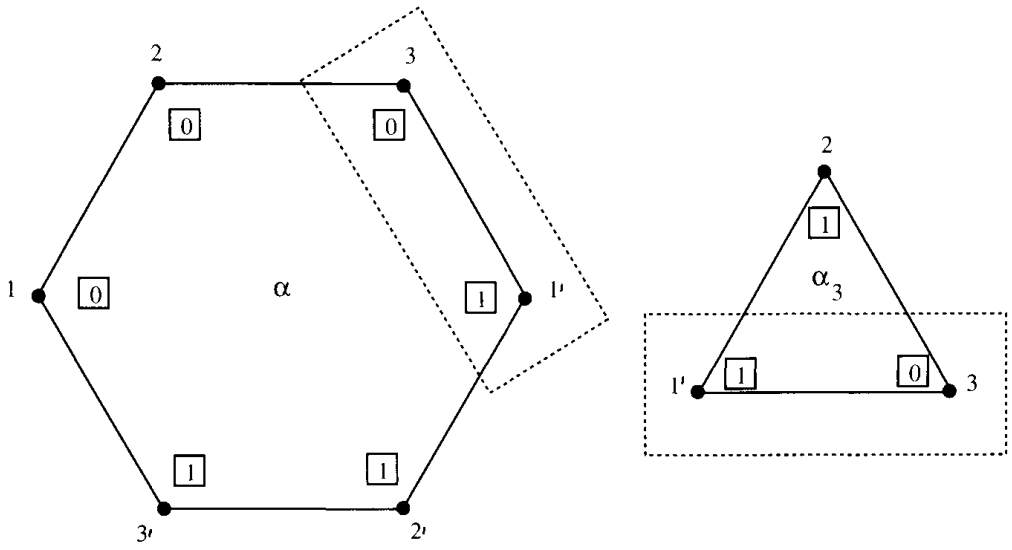

Figure 6.10: Executions α and α_3 are indistinguishable to processes 1 and 3.

no requirement about what value they agree upon, but the agreement condition implies that they agree. Therefore, they decide on the same value in α also.

But this is a contradiction, because we have already observed that in α, process 3 decides 0 and process $1'$ decides 1. □

We now use Lemma 6.26 to show that Byzantine agreement is impossible with $n \leq 3f$ processes. We do this by showing how the existence of an $n \leq 3f$ process solution that can tolerate f Byzantine failures implies the existence of a three-process solution that can tolerate a single Byzantine failure, which contradicts Lemma 6.26.

Theorem 6.27 *There is no solution to the Byzantine agreement problem for n processes in the presence of f Byzantine failures, if $2 \leq n \leq 3f$.*

Proof. For the special case where $n = 2$, it is easy to see that the problem cannot be solved. Informally speaking, suppose that one process starts with 0 and the other with 1. Then each must allow for the possibility that the other is faulty and decide on its own value, in order to ensure the validity property. But if neither is faulty, this violates the agreement property. So we may assume that $n \geq 3$.

Assume for the sake of contradiction that there is a solution A for Byzantine agreement with $3 \leq n \leq 3f$. We show how to transform A into a solution B to Byzantine agreement for three processes, numbered 1, 2, and 3, tolerating one fault. Each of the three processes in B will simulate approximately one-third of the processes of A.

Specifically, we partition the processes of A into three nonempty subsets, I_1, I_2, and I_3, each of size at most f. We let each process i in B simulate the processes in I_i, as follows.

> **B:**
> Each process i keeps track of the states of all the processes in I_i, assigns its own initial value to every member of I_i, and simulates the steps of all the processes in I_i as well as the messages between pairs of processes in I_i. Messages from processes in I_i to processes in another subset are sent from process i to the process simulating that subset. If any simulated process in I_i decides on a value v, then i decides on the value v. (If there is more than one such value, then i can choose any such value.)

We show that B correctly solves Byzantine agreement for three processes. Designate the faulty processes of A to be exactly those that are simulated by faulty processes of B.[3] Fix any particular execution α of B with at most one faulty process and let α' be the simulated execution of A. Since each process of B simulates at most f processes of A, there are at most f faulty processes in α'. Since A is assumed to solve Byzantine agreement for n processes with at most f faults, the usual agreement, validity, and termination conditions for Byzantine agreement hold in α'.

We argue that these conditions carry over to α. For termination, let i be a nonfaulty process of B. Then i simulates at least one process, j, of A, and j must be nonfaulty since i is. The termination condition for α' implies that j must eventually decide; as soon as it does so, i decides (if it has not already done so).

For validity, if all nonfaulty processes of B begin with a value v then all the nonfaulty processes of A also begin with v. Validity for α' implies that v is the only decision value for a nonfaulty process in α'. Then v is the only decision value for a nonfaulty process in α.

For agreement, suppose i and j are nonfaulty processes of B. Then they simulate only nonfaulty processes of A. Agreement for α' implies that all of these simulated processes agree, so i and j also agree.

We conclude that B solves the Byzantine agreement problem for three processes, tolerating one fault. But this contradicts Lemma 6.26. □

[3] We invoke the technicality that Byzantine faulty processes are allowed to behave completely correctly, in order to justify this classification.

6.5 Byzantine Agreement in General Graphs

So far in this chapter, we have considered agreement problems only in complete graphs. For complete graphs with n nodes, we showed in Sections 6.3 and 6.4 that Byzantine agreement can be solved if and only if $n > 3f$. In this section, we consider the problem of Byzantine agreement in general network graphs. We characterize exactly the graphs in which the problem is solvable.

First, if the network graph is a tree with at least three nodes, we cannot hope to solve the Byzantine agreement problem with even one faulty process, for any faulty process that is not a leaf could essentially "disconnect" the processes in one part of the tree from the processes in another. The nonfaulty processes in different components would not even be able to communicate reliably, much less reach agreement. Similarly, it should be plausible that if f nodes can disconnect the graph, then Byzantine agreement is impossible with f faulty processes.

To formalize this intuition, we use the following notion from graph theory. The *connectivity* of a graph G, $conn(G)$, is defined to be the minimum number of nodes whose removal results in either a disconnected graph or a trivial 1-node graph. Graph G is said to be *c-connected* if $conn(G) \geq c$.

Example 6.5.1 Connectivity

Any tree with at least two nodes has connectivity 1, and an n-node complete graph has connectivity $n - 1$. Figure 6.11 shows a graph with connectivity 2. If nodes 2 and 4 are removed, then we are left with two disconnected nodes, 1 and 3.

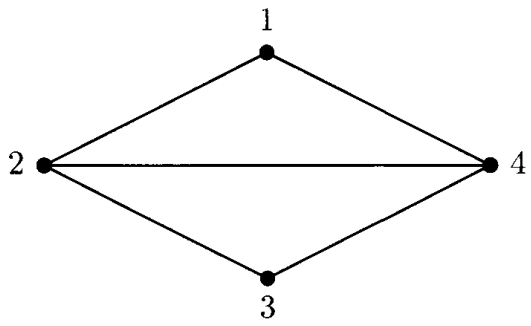

Figure 6.11: A graph G with $conn(G) = 2$.

We use a classical theorem of graph theory known as *Menger's Theorem.*

Theorem 6.28 (Menger's Theorem) *A graph G is c-connected if and only if every pair of nodes in G is connected by at least c node-disjoint paths.*

Now we can characterize those graphs in which it is possible to solve Byzantine agreement with a given number of faults. The characterization is in terms of both the number of nodes in the graph and the connectivity. The proof of the impossibility part of the characterization uses methods similar to those used in Section 6.4 to prove the lower bound for the number of faulty processes.

Theorem 6.29 *The Byzantine agreement problem can be solved in an n-node network graph G, tolerating f faults, if and only if both the following hold:*

1. $n > 3f$
2. $conn(G) > 2f$

Proof. We have already shown, in Theorem 6.27, that $n > 3f$ processes are required to solve Byzantine agreement in a complete graph. It should not be hard to believe that in an arbitrary (not necessarily complete) network graph we still need $n > 3f$; this is because an algorithm for an incomplete graph with $n \leq 3f$ could also be run in an n-node complete graph.

We next show the *if* direction of the proof, namely, that Byzantine agreement is possible if $n > 3f$ and $conn(G) > 2f$. Since G is $2f + 1$-connected, Menger's Theorem, Theorem 6.28, implies that there are at least $2f+1$ node-disjoint paths between any two nodes in G. It is possible to implement reliable communication between any pair of nonfaulty processes, i and j, by having i send a message along $2f+1$ paths between itself and j. Since there are at most f faulty processes, the messages received by j along a majority of these paths must be correct.

Once we have reliable communication between all pairs of nonfaulty processes, we can solve Byzantine agreement just by simulating any algorithm that solves the problem in an n-node complete graph. The implementation given above for reliable communication is used in place of the point-to-point communication in the complete graph. Of course, there is an increase in complexity, but that is not the issue here—the algorithm still works correctly.

We now turn to the most interesting part of the proof, showing that Byzantine agreement can only be solved if $conn(G) > 2f$. We simplify matters by only arguing the case where $f = 1$; we leave the (similar) argument for larger values of f for an exercise.

So, assume there is a graph G with $conn(G) \leq 2$, in which Byzantine agreement can be solved in the presence of one fault, using algorithm A. Then there are two nodes in G that either disconnect G or reduce it to one node. But if they reduce it to one node, it means that G consists of only three nodes, and we already know that Byzantine agreement cannot be solved in a three-node graph in the presence of one fault. So we can assume that the two nodes disconnect G.

Then the picture must be something like Figure 6.11, except that nodes 1 and 3 might be replaced by arbitrary connected subgraphs and there might be several edges between each of processes 2 and 4 and each of the two connected subgraphs. (The link between 2 and 4 could also be missing, but this would only make things harder.) Again for simplicity, we just consider the case where 1 and 3 are single nodes. We construct a system S by combining two copies of A. We start one copy of each process with input value 0 and the other with input value 1, as shown in Figure 6.12. As in the proof of Lemma 6.26, S with the given input assignment does exhibit some well-defined behavior. Again, we will obtain a contradiction by showing that no such behavior is possible.

So suppose that the processes in S are started with the input values indicated in Figure 6.12, that is, the unprimed processes with 0 and the primed processes with 1; let α be the resulting execution of S.

We consider α from the point of view of processes 1, 2, and 3. To these processes, it appears as if they are running in system A, in an execution α_1 in which process 4 is faulty. See Figure 6.13. Then the correctness conditions for Byzantine agreement imply that eventually in α_1, processes 1, 2, and 3 must decide 0. Since α is indistinguishable from α_1 to processes 1, 2, and 3, all three must eventually decide 0 in α as well.

Next consider α from the point of view of processes $1'$, $2'$, and $3'$. To these three processes, it appears as if they are running in A, in an execution α_2 in which process 4 is faulty. See Figure 6.14. By the same argument, processes $1'$, $2'$, and $3'$ must eventually decide 1 in α.

Finally, consider execution α from the point of view of processes 3, 4, and $1'$. To these processes, it appears as if they are running in A, in an execution α_3 in which process 2 is faulty. See Figure 6.15. By the correctness conditions for Byzantine agreement, these three processes must eventually decide in α_3, and their decisions must be the same. Then the same is true in α.

But this is a contradiction, because we have already shown that process 3 must decide 0 and process $1'$ must decide 1 in α. It follows that we cannot solve Byzantine agreement in graphs G with $conn(G) \leq 2$ and $f = 1$.

In order to generalize the result to $f > 1$, we can use the same diagrams, with 2 and 4 replaced by sets I_2 and I_4 of at most f nodes each and 1 and 3 by arbitrary sets I_1 and I_3 of nodes. Removing all the nodes in I_2 and I_4 disconnects I_1 and I_3. The edges of Figure 6.11 can now be considered to represent bundles of edges between the different groups of nodes I_1, I_2, I_3, and I_4. □

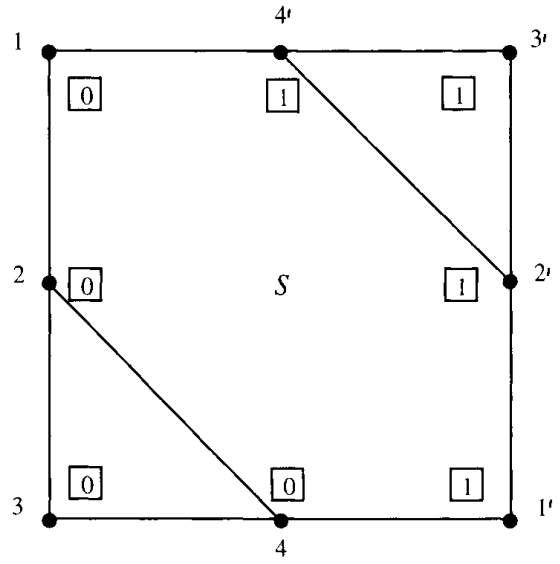

Figure 6.12: Combining two copies of A to get S.

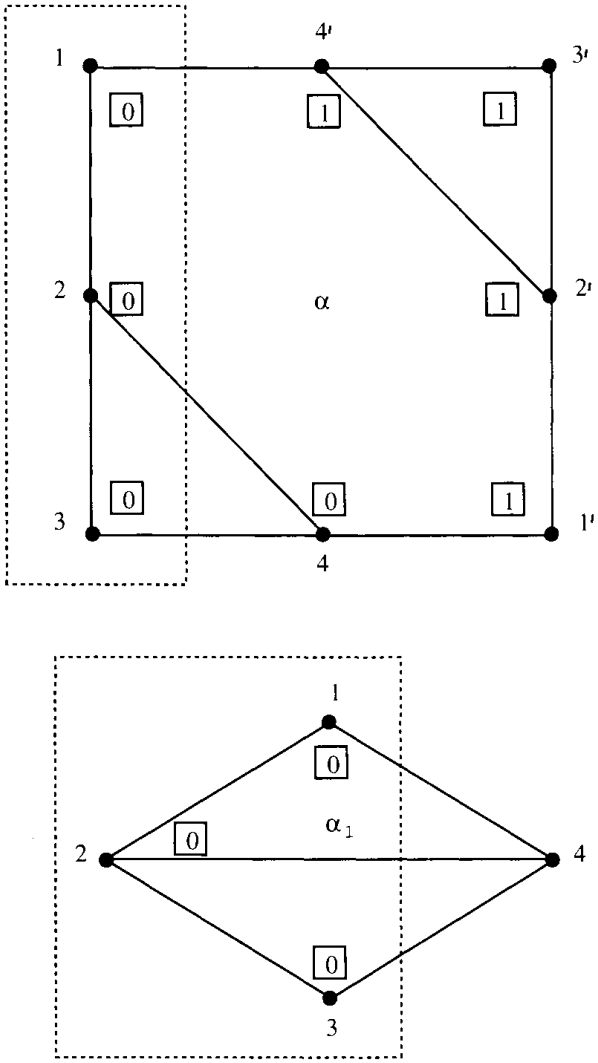

Figure 6.13: Executions α and α_1 are indistinguishable to processes 1, 2, and 3.

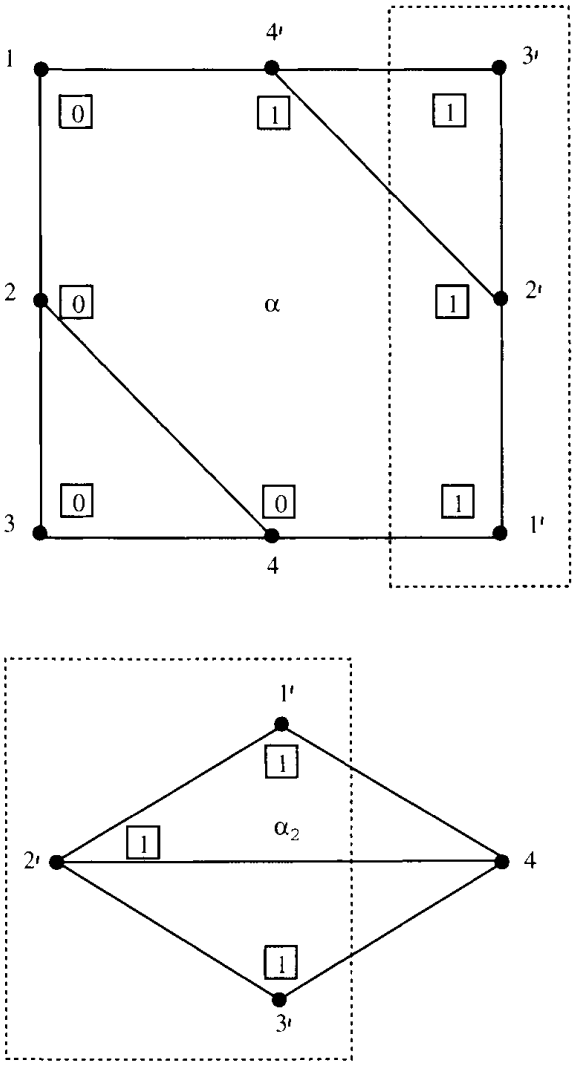

Figure 6.14: Executions α and α_2 are indistinguishable to processes $1'$, $2'$, and $3'$.

6.6 Weak Byzantine Agreement

The same general proof method that we used in Sections 6.4 and 6.5 to prove impossibility for Byzantine agreement with $n \leq 3f$ or $conn \leq 2f$ can also be used to prove impossibility for other consensus problems. As an example, in this section we show how this method can be used to prove impossibility for a weaker variant of the Byzantine agreement problem known as *weak Byzantine agreement.*

The only difference between the problem statement for weak Byzantine agreement and ordinary Byzantine agreement is in the validity condition. The validity condition for weak Byzantine agreement is

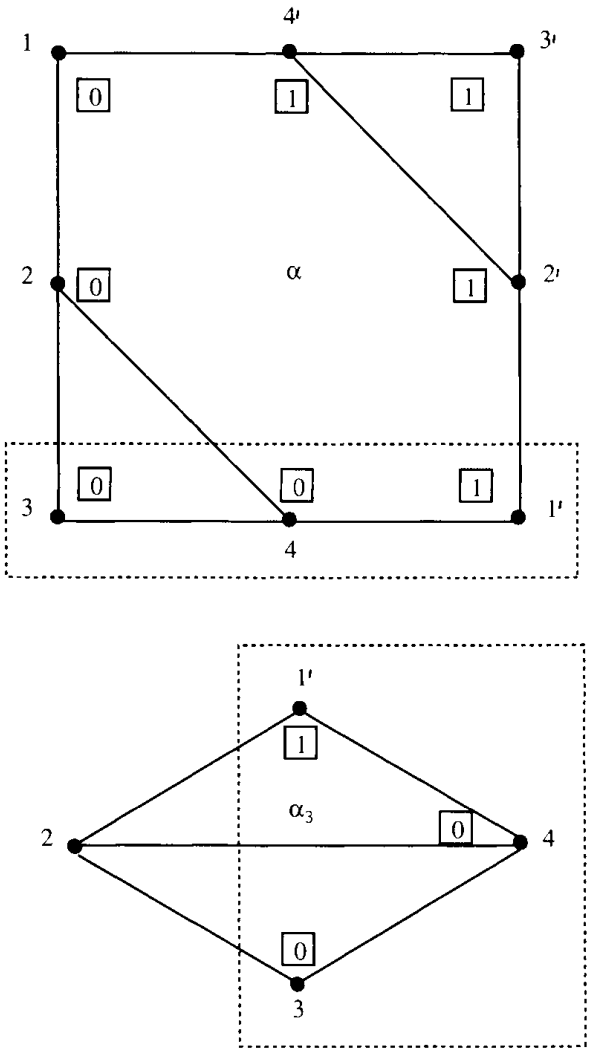

Figure 6.15: Executions α and α_3 are indistinguishable to processes 3, 4, and $1'$.

Validity: If there are no faulty processes and all processes start with the same initial value $v \in V$, then v is the only possible decision value.

In the ordinary Byzantine agreement problem, if all the nonfaulty processes start with the same initial value v, then they must all decide v *even if there are faulty processes.* In weak Byzantine agreement, they are required to decide v only in the case where there are no failures.

Since the new problem statement is weaker than the old one, the algorithms we have described for ordinary Byzantine agreement also work for weak Byzantine agreement. On the other hand, the impossibility results do not immediately carry over; it is plausible that more efficient algorithms might exist for weak Byzantine agreement. However, it turns out that (except for a tiny technicality)

the limitations on the number of processes and the graph connectivity still hold. (The technicality is that now we need to assume that $n \geq 3$, because there is a trivial algorithm for weak Byzantine agreement for the special case where $n = 2$.)

Theorem 6.30 *Assume that $n \geq 3$. The weak Byzantine agreement problem can be solved in an n-node network graph G, tolerating f faults, if and only if both the following hold:*

1. $n > 3f$
2. $conn(G) > 2f$

Proof. The *if* direction follows from the existence of protocols for ordinary Byzantine agreement, as claimed in Theorem 6.29. We give the proof that three processes cannot solve weak Byzantine agreement with one possible fault and leave the extension to $f > 1$ and the connectivity argument for exercises. For simplicity, we assume that $V = \{0, 1\}$.

Assume there is a three-process algorithm A that solves the weak Byzantine agreement problem for the three processes 1, 2, and 3, even if one is faulty. Let α_0 be the execution of A in which all three processes start with 0 and no failures occur. The termination and validity conditions then imply that all three processes eventually decide 0 in α_0; let r_0 be the smallest round number by which all processes decide. Likewise, let α_1 be the execution in which all processes start with 1 and no failures occur, so all processes eventually decide 1 in α_1. Let r_1 be the number of rounds required and choose $r \geq \max\{r_0, r_1, 1\}$.

We construct a new system S by pasting $2r$ copies of A into a ring with $6r$ processes, $3r$ in the "top half" and $3r$ in the "bottom half." We start all the processes in the top half with input value 0 and those in the bottom half with input value 1. The arrangement is shown in Figure 6.16. (This time, we have not bothered to include prime symbols or other distinguishing notation for the multiple copies of the same process of A.) Let α be the resulting execution of S.

By arguing as in the proof of Lemma 6.26, we can show that any two adjacent processes in S must decide on the same value in execution α; this is because it looks to the two processes as if they are in the triangle, interacting with a third process that is faulty. It follows that all processes in S must reach the *same* decision in α. Suppose (without loss of generality) that they all decide 1.

Now to get a contradiction, we argue that some process in the top half of S must decide 0. Let B be any "block" of $2r + 1$ consecutive processes in the top half of S; these all start with initial value 0 in α. Now, all the processes in B begin in the same state in α as the same-named processes do in α_0, and send the same messages at round 1. Thus, at round 1, all the processes in B

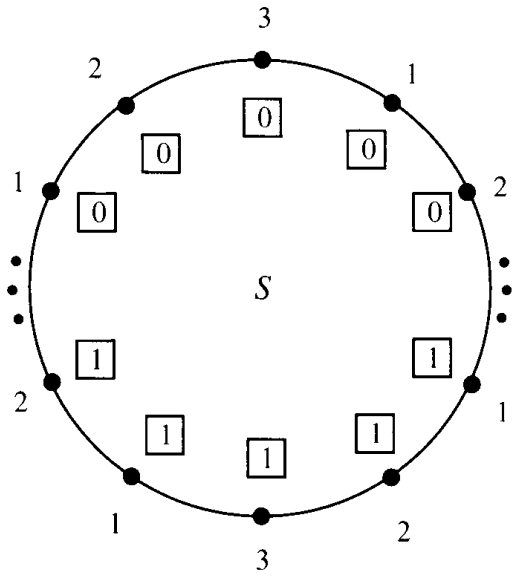

Figure 6.16: Combining $2r$ copies of A to get S.

except possibly for the one at each end receive the same messages in α as their namesakes do in α_0 and so remain in the same states and send the same messages at round 2, in the two executions. At round 2, all processes in B except the two at each end receive the same messages and remain in the same states, in the two executions. Continuing in this way, we see that at round k, $1 \leq k \leq r$, all processes in B except the k at each end receive the same messages and remain in the same states, in α and α_0. In other words, α and α_0 are indistinguishable to all processes in B except the k at each end, for k rounds. Informally speaking, this is because information does not have time to propagate to those processes from outside the block B.

In particular, α and α_0 are indistinguishable to the middle process, process i, of block B for r rounds. But since process i decides 0 by the end of round r in α_0, it also does so in α. This contradicts the fact that process i decides 1 in α. □

6.7 Number of Rounds with Stopping Failures

We complete this chapter by showing that the agreement problem cannot be solved in fewer than $f + 1$ rounds, either for Byzantine or stopping failures. In other words, there does not exist an agreement protocol, for either type of failure, in which all the nonfaulty processes decide by the end of f rounds.

We will proceed by assuming that an f-round agreement algorithm exists and

obtaining a contradiction. It is convenient for us to impose some restrictions on the assumed algorithm, none of which causes any loss of generality. First, we assume that the network graph is completely connected; a fast algorithm for an incomplete graph could also be run in a complete graph, so there is certainly no loss of generality in this restriction. We also assume that all processes that decide do so exactly at the end of round f, then immediately halt. In this case, an algorithm for Byzantine agreement is necessarily an algorithm for stopping agreement (see the remark on the relationship between the two problems in Section 6.1). So, for the purpose of obtaining an impossibility result, we can restrict attention to the stopping agreement problem only. Also, we assume that every process sends a message to every other process at every round k, $1 \leq k \leq f$ (unless and until it fails). Finally, we restrict attention to the case where the value set $V = \{0, 1\}$.

As for the coordinated attack problem in Chapter 5, it is convenient to carry out the proof using the notion of a communication pattern, which is an indication of which processes send messages to which other processes at each round. Specializing the previous definition to the case of a complete graph, we define a *communication pattern* to be any subset of the set

$$\{(i, j, k) : 1 \leq i, j \leq n, i \neq j, 1 \leq k\}.$$

A communication pattern does not describe the contents of messages, but only which processes send to which others, at which rounds.

We consider three restrictions on communication patterns. First, because the algorithm we consider has f rounds, we consider only communication patterns in which all triples (i, j, k) have $k \leq f$. Second, because we are working with the stopping failure model, all the communication patterns that arise satisfy the following restriction: if any triple (i, j, k) is missing from the pattern, then so is every triple of the form (i, j', k'), where $k' > k$. That is, if process i fails to send any of its messages at round k, then it sends no messages at subsequent rounds. Third, because we consider executions with at most f failures, all the communication patterns that arise contain at most f faulty processes. (We define a process i to be *faulty* in a communication pattern if some triple of the form (i, j, k), $k \leq f$, is missing from the pattern.) We say (in the rest of this chapter only) that a communication pattern that satisfies these three restrictions is *good*.

Example 6.7.1 Good communication pattern

An example of a good communication pattern (for $n = f = 4$) is depicted in Figure 6.17. In this pattern, process 3 sends a message to process 4 but fails to send messages to processes 1 and 2 at round 1.

Thus, it must be that process 3 stops in round 1 and sends nothing in later rounds. Also, process 2 stops just at the end of round 2. Processes 1 and 4 are nonfaulty.

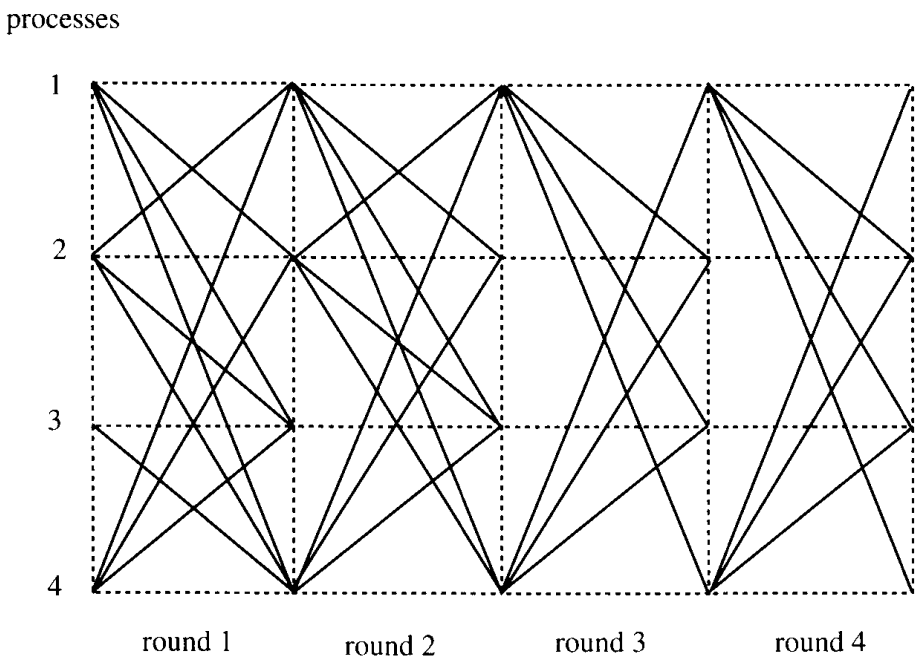

Figure 6.17: A good communication pattern.

Now we define a *run* to be a combination of

1. An assignment of input values to all the processes
2. A good communication pattern

(This is similar to what we called an adversary in Section 5.2.1.)

For a particular agreement algorithm A, each run ρ defines a corresponding execution, $exec(\rho)$, in a natural way. Namely, the initial states of the processes are defined by setting the input state components according to the input assignment given in ρ; the messages that are sent are determined from the communication pattern of ρ, using the message transition function of A applied to the prior state of the sender process; and states after the initial states are determined using the transition function of A. (But after any process fails to send a message, we stop applying its state-transition function.)

In order to give some intuition for the lower bound, we begin by proving the theorem for the special case where $f = 1$.

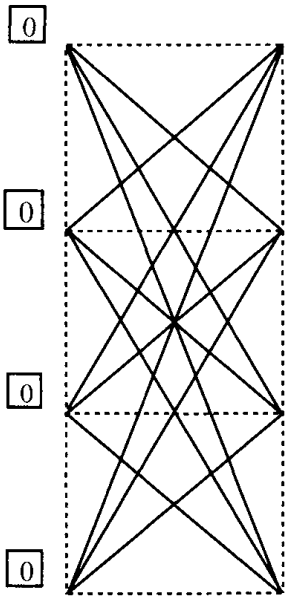

Figure 6.18: Run ρ_0—all inputs are 0, and there are no failures.

Theorem 6.31 *Suppose that $n \geq 3$. Then there is no n-process stopping agreement algorithm that tolerates one fault, in which all nonfaulty processes always decide by the end of round 1.*

Proof. Suppose, to obtain a contradiction, that there is such an algorithm, A; we assume that A satisfies all the restrictions listed at the beginning of this section.

The idea is to construct a chain of executions of A, each with at most one faulty process, such that (a) the first execution in the chain contains 0 as its unique decision value, (b) the last execution in the chain contains 1 as its unique decision value, and (c) any two consecutive executions in the chain are indistinguishable to some process that is nonfaulty in both. Then, since any two consecutive executions look the same to some nonfaulty process, say i, process i must make the same decision in both executions; therefore, the two executions must have the same unique decision value. It follows that *every* execution in the chain must have the same unique decision value, which contradicts the combination of properties (a) and (b).

We start the chain with the execution $exec(\rho_0)$ determined from the run ρ_0 in which all processes have input value 0 and no process is faulty. This run is depicted in Figure 6.18. By validity, the unique decision value in $exec(\rho_0)$ must be 0. Starting from execution $exec(\rho_0)$, we form the next execution by removing a single message—the one from process 1 to process 2. The result is depicted in Figure 6.19. This execution is indistinguishable from $exec(\rho_0)$ to every process

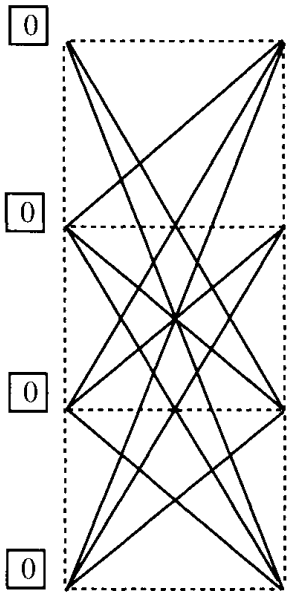

Figure 6.19: The result of removing one message from ρ_0.

except for 1 and 2. Since $n \geq 3$, there is at least one such process. This process is nonfaulty in both executions.

Next we remove the message from 1 to 3; this and the previous execution are indistinguishable to each process except for 1 and 3, and there is at least one such process. We continue in this manner, removing one message from process 1 at a time, in such a way that every two consecutive executions are indistinguishable to some nonfaulty process.

Once we have removed all the messages sent by 1, we continue by changing process 1's input value from 0 to 1. Of course, the resulting execution is indistinguishable from the previous one to every process except 1, since 1 sends no messages in either execution. Next, we replace process 1's messages one by one, and again every consecutive pair of executions is indistinguishable to some nonfaulty process. In this way, we end up with $exec(\rho_1)$, where ρ_1 is defined to be the run in which process 1 has input value 1, all the rest have input value 0, and there are no failures.

Next, we repeat this construction for process 2, first removing its messages one by one, then changing 2's input value from 0 to 1, and then replacing its messages. The resulting execution is $exec(\rho_2)$, where ρ_2 is the run in which processes 1 and 2 have input value 1, the others have input value 0, and there are no failures. Repeating this construction for processes $3, \ldots, n$, we end up with $exec(\rho_n)$, where ρ_n is the run in which all processes start with 1 and there are no failures.

So we have constructed a chain from $exec(\rho_0)$ to $exec(\rho_n)$ satisfying property (c). But validity implies that the unique decision value in $exec(\rho_0)$ is 0 and the unique decision value in $exec(\rho_n)$ is 1, which yields (a) and (b). So we have the needed chain, which gives a contradiction. □

Before moving to the general case, we will do one more preliminary case—the case where $f = 2$.

Theorem 6.32 *Suppose that $n \geq 4$. Then there is no n-process stopping agreement algorithm that tolerates two faults, in which all nonfaulty processes always decide by the end of round 2.*

Proof. Again suppose that there is such an algorithm. We construct a chain with the same properties (a), (b), (c) as in the previous proof, using a similar construction. For each i, $0 \leq i \leq n$, let ρ_i denote the (two-round) run in which processes $1, \ldots, i$ have input 1, processes $i+1, \ldots, n$ have input 0, and there are no faults. The chain starts with $exec(\rho_0)$, ends with $exec(\rho_n)$, and passes through all the executions $exec(\rho_i)$ along the way.

Starting with $exec(\rho_0)$, we want to work toward killing process 1 at the beginning. When we were only dealing with one round, we could simply remove messages from process 1 one by one. Now there is no problem in removing process 1's round 2 messages one by one. But if we remove a round 1 message from 1 to some other process i in one step of the chain, it is no longer the case that the two consecutive executions must look the same to some nonfaulty process. This is because in round 2, i is able to tell all other processes whether or not it received a message from process 1 in round 1.

We solve this problem by using several steps to remove the round 1 message from 1 to i. In the intermediate executions that occur along the way, processes 1 and i are both faulty; this is permissible since $f = 2$. In particular, we start with an execution in which 1 sends a message to i at round 1 and i is nonfaulty. We remove round 2 messages sent by i, one by one, until we obtain an execution in which 1 sends to i at round 1 and i sends no messages at round 2. Next, we remove the round 1 message from 1 to i; the resulting execution is indistinguishable from the preceding one to all processes other than 1 and i. Then we replace round 2 messages sent by i one by one, until we obtain an execution in which 1 does not send to i at round 1 and i is nonfaulty. This achieves our goal of removing a round 1 message from 1 to i, while ensuring that each consecutive pair of executions are indistinguishable to some nonfaulty process.

In this way, we remove round 1 messages from 1 one by one until 1 sends no messages. Then we change process 1's input from 0 to 1 as before. We continue

this procedure "in reverse," replacing process 1's round 1 messages one by one. Repeating this for processes $2, \ldots, n$ gives the needed chain. □

We now prove the general theorem:

Theorem 6.33 *Suppose that $n \geq f + 2$. Then there is no n-process stopping-agreement algorithm that tolerates f faults, in which all nonfaulty processes always decide by the end of round f.*

The proofs of Theorems 6.31 and 6.32 contain the main ideas for the proof of Theorem 6.33. In the general proof, a longer chain is constructed, using f process failures. We proceed more formally than we did in the proofs of Theorems 6.31 and 6.32. We need some notation.

First, if ρ and ρ' are runs in both of which process i is nonfaulty, then we write $\rho \overset{i}{\sim} \rho'$ to mean that $exec(\rho) \overset{i}{\sim} exec(\rho')$—that is, the executions generated by runs ρ and ρ' are indistinguishable to process i. We write $\rho \sim \rho'$ if $\rho \overset{i}{\sim} \rho'$ for some process i that is nonfaulty in both ρ and ρ'. And we write $\rho \approx \rho'$ for the transitive closure of the $\sim$ relation.

Next, notice that all the communication patterns that occur in the chains in the proofs of Theorems 6.31 and 6.32 have a particularly simple form. We capture this form with the following definition. We define a good communication pattern to be *regular* if for every k, $0 \leq k \leq f$, there are at most k processes that fail (to send at least one message) by the end of k rounds. We say that a run or execution is *regular* if its communication pattern is regular.

Finally, if ρ is any run and $0 \leq k \leq f$, we define the run $ff(\rho, k)$—the variant of ρ that is *failure-free* after time k—to be the run that has the same input assignment as ρ, and whose communication pattern is the same as that of ρ for the first k rounds and contains no new failures thereafter. Here are some obvious facts involving ff runs.

Lemma 6.34 *If ρ is a regular run, then*

1. *For any k, $0 \leq k \leq f$, $ff(\rho, k)$ is regular.*
2. *If ρ' is identical to ρ except that some process i that fails in ρ fails at a later round in ρ', then ρ' is regular.*
3. *If no process fails at round $k + 1$, then $ff(\rho, k) = ff(\rho, k + 1)$.*

The heart of the proof of Theorem 6.33 is the following strong lemma, which says that it is possible to construct a chain between *any two* regular executions having the same input assignment.

Lemma 6.35 *Suppose that A is an n-process stopping agreement algorithm that tolerates f faults, in which all nonfaulty processes always decide by the end of round f. Let ρ and ρ' be two regular runs of A with the same input assignment. Then $\rho \approx \rho'$.*

Proof. We show this by proving the following parameterized claim. The case where $k = 0$ immediately implies the lemma.

Claim 6.36 *Let k be an integer, $0 \leq k \leq f$. Let ρ and ρ' be two regular runs of A with the same input assignment and with identical communication patterns through k rounds. Then $\rho \approx \rho'$.*

Proof. The proof of Claim 6.36 is by reverse induction on k, starting with $k = f$ and ending with $k = 0$.

Basis: $k = f$. This case is trivial because the assumption that ρ and ρ' have the same inputs and same communication patterns through f rounds implies that ρ and ρ' are identical.

Inductive step: $0 \leq k \leq f - 1$ and the claim is true for $k + 1$. In this case, it is enough to show that any regular run ρ satisfies $\rho \approx ff(\rho, k)$, because we can apply this result twice to obtain the required claim. So fix some regular run ρ. By Lemma 6.34, $ff(\rho, k)$ is regular.

By inductive hypothesis, $ff(\rho, k+1) \approx \rho$, so it is enough to show that $ff(\rho, k) \approx ff(\rho, k+1)$. If no process fails at round $k+1$ in ρ, then Lemma 6.34 implies that $ff(\rho, k) = ff(\rho, k + 1)$ and we are done. So we assume that at least one process fails at round $k + 1$ in ρ. Let I be the set of processes that do so.

Let ρ_0 be the run that is identical to $ff(\rho, k)$ except that all processes in I fail at the end of round $k + 1$. Then Lemma 6.34, part 2 (applied to ρ), implies that ρ_0 is regular.

Since ρ_0 and $ff(\rho, k)$ are regular runs that are identical through $k + 1$ rounds, we can apply the inductive hypothesis to show that $\rho_0 \approx ff(\rho, k)$. Therefore, to show that $ff(\rho, k) \approx ff(\rho, k + 1)$, it is enough to show that $\rho_0 \approx ff(\rho, k + 1)$.

Now we construct a chain of regular runs spanning from ρ_0 to $ff(\rho, k + 1)$. The only difference between ρ_0 and $ff(\rho, k + 1)$ is that some messages sent by processes in I at round $k + 1$ in ρ_0 are missing in $ff(\rho, k + 1)$. So we remove those messages one at a time, while keeping the runs otherwise unchanged.

For instance, consider the removal of a message from i to j, where $i \in I$. Let σ be the run including the message and τ be the run without the message; we must argue that $\sigma \approx \tau$. If $k + 1 = f$, then σ and τ are indistinguishable to all processes except for i and j; since $n \geq f + 2$ and i is faulty, this must include at least one nonfaulty process. So $\sigma \approx \tau$, as needed.

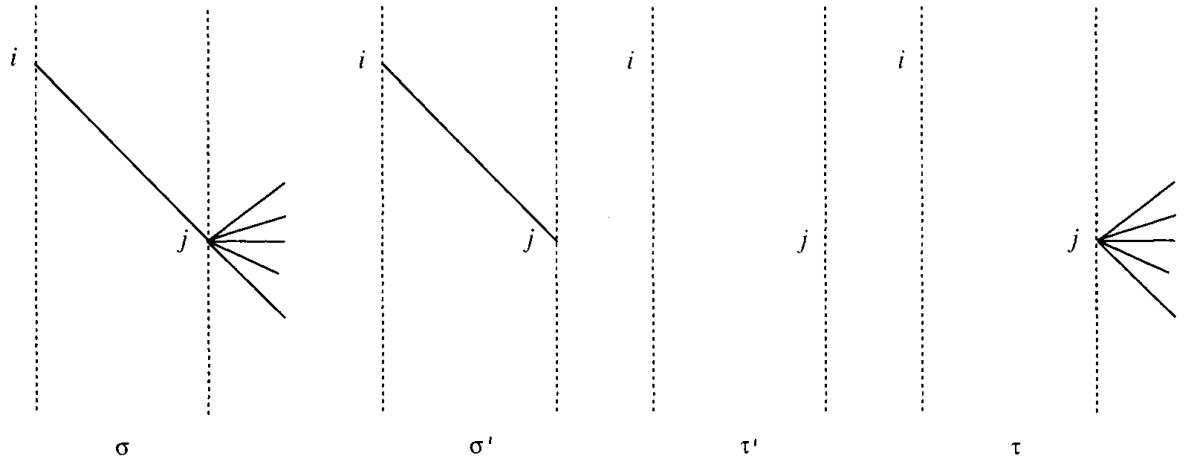

Figure 6.20: Removal of round $k+1$ message from i to j, in proof of Claim 6.36.

On the other hand, if $k+1 \leq f-1$, then define σ' and τ' to be the same as σ and τ respectively, but with j failing just at the start of round $k+2$ (if it has not previously failed). See Figure 6.20.

Both σ' and τ' are regular, since each of σ and τ involves at most $k+1 \leq f-1$ failures, and we only introduce one new failure for the new round $k+2$. Then $\sigma \approx \sigma'$ and $\tau \approx \tau'$, by inductive hypothesis. And $\sigma' \approx \tau'$, because they are indistinguishable to all processes except for i and j. So again, $\sigma \approx \tau$.

This shows that the needed chain from ρ_0 to $ff(\rho, k+1)$ can be constructed, so $\rho_0 \approx ff(\rho, k+1)$, so $\rho \approx ff(\rho, k)$, as needed. □

As we noted earlier, Claim 6.36 immediately implies Lemma 6.35.

Now we extend Lemma 6.35 to apply to different input assignments.

Lemma 6.37 *Suppose that A is an n-process stopping agreement algorithm that tolerates f faults, in which all nonfaulty processes always decide by the end of round f. If ρ and ρ' are two regular runs of A, then $\rho \approx \rho'$.*

Proof. By Lemma 6.35, each run ρ is related to its failure-free version, that is, $\rho \approx ff(\rho, 0)$. So we can assume without loss of generality that ρ and ρ' in the statement of the lemma are both failure-free.

If ρ and ρ' have the same input assignment, then they are identical and there is nothing to prove.

Suppose that ρ and ρ' differ in the input of exactly one process i; say i has input 0 in ρ and input 1 in ρ'. Then define σ and σ' to be the runs that are identical to ρ and ρ', respectively, except that i fails right at the start. Then

Lemma 6.35 implies that $\rho \approx \sigma$ and $\rho' \approx \sigma'$. Also, $\sigma \approx \sigma'$, because σ and σ' are indistinguishable to all processes except for i. It follows that $\rho \approx \rho'$, as needed.

Finally, suppose that ρ and ρ' differ in the input of more than one process. Then we can construct a chain of failure-free runs, spanning from ρ to ρ', changing exactly one process's input at each step in the chain. The previous case applies to each step in this chain. So again, we obtain $\rho \approx \rho'$. □

Using Lemma 6.37, it is easy to prove Theorem 6.33. We already know that all regular runs are related by chains; now we consider the decision values that arise in these runs. Assuming that $n > f$, the termination and agreement properties imply that for every run ρ, there is a unique decision value, $dec(\rho)$, that arises in $exec(\rho)$. The following lemma says that runs that are related by $\sim$ or $\approx$ necessarily give rise to the same decision values.

Lemma 6.38

1. *If* $\rho \sim \rho'$, *then* $dec(\rho) = dec(\rho')$.

2. *If* $\rho \approx \rho'$, *then* $dec(\rho) = dec(\rho')$.

Proof. For part 1, recall that $\rho \sim \rho'$ means that there is a process i that is nonfaulty in both ρ and ρ', such that $exec(\rho) \overset{i}{\sim} exec(\rho')$. This implies that process i decides on the same value in $exec(\rho)$ and $exec(\rho')$. Therefore, $dec(\rho) = dec(\rho')$.

Part 2 follows from part 1. □

Proof (of Theorem 6.33). Suppose there is such an algorithm, A; we assume that A satisfies the restrictions listed at the beginning of the section.

Let ρ_0 be the run of A in which all processes start with 0 and there are no faults, and let ρ_1 be the run in which all processes start with 1 and there are no faults. Lemma 6.37 implies that $\rho_0 \approx \rho_1$. Then Lemma 6.38, part 2, implies that $dec(\rho) = dec(\rho')$. But the validity condition implies that $dec(\rho_0) = 0$ and $dec(\rho_1) = 1$, a contradiction. □

Weaker validity condition. Notice that this impossibility proof still works if we weaken the validity condition to the one that we used in Section 6.6 for the weak Byzantine agreement problem. That is, we have shown that the weak Byzantine agreement problem also requires at least $f + 1$ rounds, under the assumption that $n \geq f + 2$.

6.8 Bibliographic Notes

Many of the ideas in this chapter originated in the two seminal papers by Pease, Shostak, and Lamport [237] and by Lamport, Shostak, and Pease [187]. These two papers contain upper and lower bounds of $3f+1$ for the number of processes required for Byzantine agreement, plus an algorithm for agreement with authentication, all for the case of a completely connected graph. The presentation in the second paper is in terms of attacking generals rather than processes. It is the second paper that coined the term *Byzantine* for this fault model.

In more detail, these two papers define the Byzantine agreement problem and motivate it as an abstraction of a problem arising in the SIFT (Software-Implemented Fault Tolerance) aircraft control system [289]. The algorithms in [237] use an exponential data structure similar to an *EIG* tree; the Byzantine agreement algorithm is similar to *EIGByz*, while the algorithm using authentication is similar to *EIGStop*. The algorithms in [187] are very much the same but are formulated recursively. The impossibility proof for $n \leq 3f$ processes in [237] involves the explicit construction of detailed scenarios. The impossibility proof in [187] introduces the reduction to the case of three versus one that appears in the proof of Theorem 6.27.

Dolev and Strong [93] developed algorithms similar to *FloodSet* and *OptFloodSet* for Byzantine agreement in the case where authentication is available. Dolev [94] considered the Byzantine agreement problem in graphs that are not necessarily completely connected. He proved the connectivity bounds represented in Theorem 6.29, using explicit construction of scenarios. Dolev, Reischuk, and Strong [99] developed algorithms with "early stopping" for certain favorable communication patterns. Other early stopping algorithms were developed by Dwork and Moses [105] and by Halpern, Moses, and Waarts [145].

Bar-Noy, Dolev, Dwork, and Strong defined the *EIG* tree data structure and presented the *EIGByz* algorithm in essentially the form given in this book [39]. The *TurpinCoan* algorithm is from [279].

The first polynomial communication algorithm for Byzantine agreement was provided by Dolev and Strong [101]; it was subsequently improved by Dolev, Fischer, Fowler, Lynch, and Strong [96] to yield a time bound of $2f+3$. Coan [82] developed a tradeoff algorithm, which decreased the number of rounds to $(1+\epsilon)f$, for any $\epsilon > 0$; the communication is polynomial, but the degree of the polynomial depends on ϵ. The consistent broadcast primitive and the *ConsistentBroadcast* algorithm are due to Srikanth and Toueg [269]. The *PolyByz* algorithm is based on algorithms by Srikanth and Toueg [269] and by Dolev et al. [96]. Subsequent research by Moses and Waarts [231], Berman and Garay [49], and Garay and Moses [133] has produced $f+1$ round Byzantine agreement algorithms with

polynomial communication; the last of these also achieves the $n = 3f + 1$ minimum bound on the number of processes. Unfortunately, these algorithms are complicated.

As already noted, the $n > 3f$ lower bound on the number of processes required for Byzantine agreement was originally proved in [237, 187], while the connectivity lower bound was originally proved in [94]. However, the proofs presented in this book were developed by Fischer, Lynch, and Merritt [122]. Menger's Theorem was originally proved by Menger [225] and appears in Harary's book [147].

The weak Byzantine agreement problem was defined by Lamport [178]. The lower bound result for the number of processes needed for weak Byzantine agreement is due to Lamport [178], but the proof given here is due to Fischer, Lynch, and Merritt [122].

The first lower bound result for the number of rounds required to reach agreement was proved by Fischer and Lynch [119], for the case of Byzantine failures. The result was subsequently extended by Dolev and Strong [93] and by DeMillo, Lynch, and Merritt [88] to the case of Byzantine failures with authentication. The extension to the case of stopping failures seems to have first been carried out by Merritt [226], using ideas of Dolev and Strong [101]. Another proof of this result was presented by Dwork and Moses [105]; their proof provides a finer analysis of the time requirements for different runs. Feldman and Micali [113] obtained a constant time randomized solution using "secret-sharing" techniques.

A paper by Fischer [117] surveys much of the early work on the agreement problem.

There has been a considerable amount of work at Draper Laboratories involving the design of fault-tolerant multiprocessors and processor fault-diagnosis algorithms, using Byzantine agreement [172, 173]. These designs have been used for safety-critical applications such as unmanned undersea vehicles, nuclear attack submarines, and nuclear power plant control.

6.9 Exercises

6.1. Prove that any algorithm that solves the Byzantine agreement problem also solves the stopping agreement problem, if the validity condition for stopping failures is modified to require only that nonfaulty processes agree.

6.2. Prove that any algorithm that solves the Byzantine agreement problem, and in which all nonfaulty processes always decide at the same round, also solves the stopping agreement problem.

6.3. Prove Lemma 6.2.

6.4. Trace the execution of the *FloodSet* algorithm for four processes and two failures, where the processes have initial values 1, 0, 0, and 0, respectively. Suppose that processes 1 and 2 are faulty, with process 1 failing in the first round after sending to 2 only and process 2 failing in the second round after sending to 1 and 3 but not 4.

6.5. Consider the *FloodSet* algorithm for f failures. Suppose that instead of running for $f + 1$ rounds, the algorithm runs for only f rounds, with the same decision rule. Describe a particular execution in which the correctness requirements are violated.

6.6. (a) Describe another alternative decision rule that works correctly for the *FloodSet* algorithm, besides the ones discussed in the text.

(b) Give an exact characterization of the set of decision rules that work correctly.

6.7. Extend the *FloodSet* algorithm, its correctness proof, and its analysis to arbitrary (not necessarily complete) connected graphs.

6.8. Give code for *OptFloodSet*. Complete the proof given in the text by proving Lemmas 6.5, 6.6, and 6.7.

6.9. Consider the following simple algorithm for agreement with stopping failures, for a value domain V. Each process maintains a variable *min-val*, originally set to its own initial value. For each of $f+1$ rounds, the processes all broadcast their *min-vals*, then each resets its *min-val* to the minimum of its old *min-val* and all the values it receives in messages. At the end, the decision value is *min-val*. Give code for this algorithm, and prove (either directly or via a simulation proof) that it works correctly.

6.10. Trace the execution of the *EIGStop* algorithm for four processes and two failures, where the processes have initial values 1, 0, 0, and 0, respectively. Suppose that processes 1 and 2 are faulty, with process 1 failing in the first round after sending to 2 only, and process 2 failing in the second round after sending to 1 and 3 but not to 4.

6.11. Prove Lemma 6.11.

6.12. Prove Lemma 6.12, part 1.

6.13. Consider the *EIGStop* algorithm for f failures. Suppose that instead of running for $f + 1$ rounds, the algorithm only runs for f rounds, with the same decision rule. Describe a particular execution in which the correctness requirements are violated.

6.14. An alternative way to prove the correctness of *FloodSet* is by relating it to *EIGStop* by a simulation relation. In order to do this, it is convenient to first extend *EIGStop* by allowing each process i to broadcast all values at all rounds, not just values associated with nodes whose labels do not contain i. It must be argued that this extension does not affect correctness. Also, some details in the description of *EIGStop* must be filled in, for example, explicit *rounds* and *decision* variables, manipulated in the obvious ways. Then *FloodSet* and the modified *EIGStop* can be run side by side, starting with the same set of initial values, and with failures occurring at the same processes at exactly the same times.

Prove the correctness of *FloodSet* in this way. The heart of the proof should be the following simulation relation, which involves the states of both algorithms after the same number of rounds.

> **Assertion 6.9.1** *For any r, $0 \leq r \leq f + 1$, the following are true after r rounds.*
>
> *(a) The values of the rounds and decision variables are the same in the states of both algorithms.*
>
> *(b) For each i, the set W_i in FloodSet is equal to the set of vals that decorate nodes of i's tree in EIGStop.*

Be sure to include the statement and proof of any additional invariants of *EIGStop* that you need to establish the simulation.

6.15. Prove the correctness of *OptEIGStop*, in either of the following two ways:

(a) By a simulation of *EIGStop*, using a proof analogous to the simulation proof relating *OptFloodSet* to *FloodSet*.

(b) By relating it to *OptFloodSet*.

6.16. Prove the correctness of the *EIGStop* and *OptEIGStop* algorithms for the authenticated Byzantine failure model. Some key facts that can be used in the proof of *EIGStop* are expressed by the following assertion, analogous to the statement of Lemma 6.12:

Assertion 6.9.2 *After $f + 1$ rounds:*

(a) If i and j are nonfaulty processes, $val(y)_i = v \in V$, and xj is a prefix of y, then $val(x)_j = v$.

(b) If v is in the set of vals at any nonfaulty process, then v is an initial value of some process.

(c) If i is a nonfaulty process, and $v \in V$ is in the set of vals at process i, then there is some label y that does not contain i such that $v = val(y)_i$.

These facts follow from the properties of digital signatures.

6.17. *Research Question*: Define the authenticated Byzantine failure model formally and prove results about its power and limitations.

6.18. Give an example of an execution of *EIGStop* that shows that *EIGStop* does not solve the agreement problem for Byzantine faults.

6.19. Consider the *EIGByz* algorithm with seven processes and three rounds. Arbitrarily select two of the processes as faulty and provide random choices for the inputs of all processes and for the message values of the faulty processes. Calculate all the information produced in the execution and verify that the correctness conditions are satisfied.

6.20. In the *EIGByz* algorithm, show that not every node in the *EIG* tree need be common.

6.21. Consider the *EIGByz* algorithm. Construct explicit executions to show that the algorithm can give wrong results if it is run with

(a) Seven nodes, two faults, and two rounds.

(b) Six nodes, two faults, and three rounds.

6.22. The *TurpinCoan* algorithm uses the threshold $n - f$ at rounds 1 and 2. What other pairs of thresholds would also allow the algorithm to work correctly?

6.23. Suppose we consider the *TurpinCoan* algorithm with *two* sets of faulty processes, F and G, rather than just one. Each set has at most f processes. Processes in F behave correctly except that they can send incorrect messages during rounds 1 and 2. Processes in G are allowed to behave incorrectly during execution of the binary Byzantine agreement subroutine (and only then). What correctness conditions are guaranteed by the combined algorithm under these failure assumptions? Prove.

6.24. Now you may assume that $n > 4f$. Design an algorithm that uses a subroutine for binary Byzantine agreement and solves multivalued Byzantine agreement. The algorithm should improve on the *TurpinCoan* algorithm by only requiring one additional round rather than two.

6.25. Show that there is no upper bound on the time until a nonfaulty process might accept a message (m, i, r) in the *ConsistentBroadcast* algorithm. That is, for any t, produce an execution of *ConsistentBroadcast* in which some nonfaulty process accepts the message at a round $r' \geq r + t$.

6.26. Can you design an algorithm to implement the consistent broadcast mechanism in the Byzantine failure model, with $f \gg 1$ faults, having the additional property that no nonfaulty process ever accepts a message (m, i, r) strictly after round $r + 1$?

Either give such an algorithm and prove its correctness, or argue why no such algorithm exists.

6.27. Describe a worst-case execution of *PolyByz*, that is, one in which there is some nonfaulty process i such that the earliest round by which process i accepts messages from $2f + 1$ distinct processes is exactly round $2(f + 1)$.

6.28. A programmer at the Flaky Computer Corporation has modified his implementation of the *PolyByz* algorithm so that the acceptance threshold for each round of the form $2s - 1$ is $s - 1$ rather than $f + s - 1$, and the decision threshold is $f + 1$ rather than $2f + 1$. Is his modification correct? Prove or give a counterexample.

6.29. Design a polynomial communication algorithm for Byzantine agreement for a general input value set, without using a subroutine for binary Byzantine agreement. Your algorithm should use the consistent broadcast mechanism, but you might have to design a better implementation than the *ConsistentBroadcast* algorithm.

6.30. Design an algorithm for stopping agreement that satisfies the following *early stopping* property: If in an execution of the algorithm only $f' < f$ processes fail, then the time until all the nonfaulty processes decide is at most kf', for some constant k. Do the same for Byzantine agreement.

6.31. Design a protocol for four processes in a completely connected graph that tolerates *either* one Byzantine fault or three stopping faults. Try to minimize the number of rounds.

6.32. *Research Question*: Devise a *simple* $f+1$ round protocol solving Byzantine agreement, requiring only $3f+1$ processes and polynomial communication.

6.33. This exercise is designed to explore the construction in the proof of Lemma 6.26, which pastes together two triangle systems to yield a hexagon system.

(a) Carefully describe an algorithm A for a three-process complete graph that solves the *no-fault agreement problem*, that is, the Byzantine agreement problem in the special case where no processes are faulty.

(b) Now construct system S by pasting together two copies of your algorithm A, as in the proof of Lemma 6.26. Describe carefully the execution of S in which processes 1, 2, and 3 start with input 0, and $1'$, $2'$, and $3'$ start with input 1.

(c) Does S solve the no-fault agreement problem (for the hexagon network)? Either prove that it does or give an execution that shows that it does not.

(d) Does there exist a three-process algorithm A such that *arbitrarily many* copies of A can be pasted together into a ring, and the resulting ring will always solve the no-fault agreement problem?

6.34. What is the largest number of faulty processes that can be tolerated by Byzantine agreement algorithms that run in the following network graphs?

(a) A ring of size n.

(b) A three-dimensional cube, m nodes on a side, in which nodes are connected only to their neighbors in the three dimensions.

(c) A complete bipartite graph with m nodes in each of its two components.

6.35. Give a more careful impossibility proof for Byzantine agreement when $n = 2$ and $f = 1$.

6.36. Analyze the time, number of messages, and number of communication bits for the Byzantine agreement algorithm for general graphs, described in the proof of Theorem 6.29. Can you improve on any of these?

6.37. Show carefully that the simplifications assumed in the proof of Theorem 6.29 to prove that Byzantine agreement is impossible with $f = 1$ and $conn(G) \leq 2$ are in fact justified. That is, show that the existence of an algorithm for the case where nodes 1 and 3 are replaced by arbitrary connected subgraphs implies the existence of an algorithm for the case where they are just single nodes.

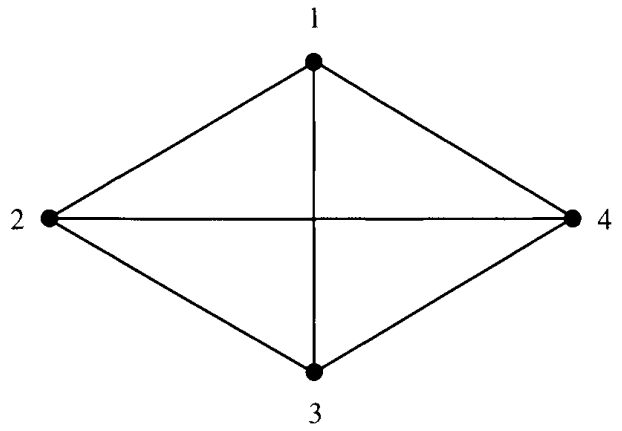

Figure 6.21: Network graph for Exercise 6.38.

6.38. Reconsider the proof that Byzantine agreement cannot be reached in the graph in Figure 6.11. Why does this proof fail to extend to the graph in Figure 6.21?

6.39. Prove that Byzantine agreement for f failures, where $f > 1$, cannot be solved in a graph G with $conn(G) \leq 2f$. This can be done either using the process grouping argument sketched at the end of the proof of Theorem 6.29, or else using a reduction similar to the one in Theorem 6.27.

6.40. Give a simple algorithm for weak Byzantine agreement in a network graph consisting of two nodes connected by a single link.

6.41. Complete the proof of Theorem 6.30, by showing impossibility

(a) When $n \leq 3f$ and $f > 1$.

(b) When $conn(G) \leq 2f$.

6.42. Consider the *Byzantine Firing Squad* problem, defined as follows. There are n processes in a fully connected network with no input values and with variable start times. That is, each process begins in a *quiescent* state containing no information and from which it sends only *null* messages. It does not change state until and unless it receives a special *wakeup* message from the outside or a non-*null* message from another process. A process does not know the current round number when it awakens. The model is similar to the one in Section 2.1, except that we do not assume here that all processes must receive *wakeup* messages—only some arbitrary subset of the processes. Also, we permit Byzantine faults.

The problem is for processes to issue *fire* signals, subject to the following conditions:

Agreement: If any nonfaulty process issues a *fire* signal at some round, then all nonfaulty processes issue a *fire* signal at that same round and no nonfaulty process issues a *fire* signal at any other round.

Validity: If all nonfaulty processes receive *wakeup* messages, then all nonfaulty processes eventually *fire*; if no nonfaulty process receives a *wakeup* message, then no nonfaulty process ever *fires*.

(a) Design an algorithm to solve the Byzantine Firing Squad problem for $n > 3f$.

(b) Prove that the problem cannot be solved if $n \leq 3f$.

6.43. State and give a direct proof of the special case of Theorem 6.33 for $f = 3$.

6.44. Does Lemma 6.37 still hold if the runs are not required to be regular? Give a proof or a counterexample.

6.45. In Section 6.7, it is shown that stopping agreement tolerating f faults cannot be solved in f rounds. The construction involves the construction of a long chain connecting the two runs in which all the processes are nonfaulty and have the same inputs. The chain, however, is only constructed implicitly.

(a) How long is the chain of runs?

(b) By how much can you shorten this chain using Byzantine faults rather than stopping faults?

6.46. *Research Question*: Obtain upper and lower bound results about the time required to solve the stopping agreement problem and/or the Byzantine agreement problem, in general (not necessarily complete) network graphs.