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(oral) 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 \( \text{msgs} \) function; and might perform arbitrary state transitions, not necessarily those specified by its \( \text{trans} \) 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:**
- \( \text{rounds} \in \mathbb{N} \), initially 0
- \( \text{decision} \in V \cup \{ \text{unknown} \} \), initially unknown
- \( W \subseteq V \), initially the singleton set consisting of \( i \)'s initial value

**msgs:**
- if \( \text{rounds} \leq f \) then send \( W \) to all other processes

**trans:**
- \( \text{rounds} := \text{rounds} + 1 \)
- let \( X_j \) be the message from \( j \); for each \( j \) from which a message arrives
- \( W := W \cup \bigcup X_j \)
- if \( \text{rounds} = f+1 \) then
  - if \( |W| = 1 \) then \( \text{decision} := v \), where \( W = \{ v \} \)
  - else \( \text{decision} := v_0 \)
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 non-faulty 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.)
6.3 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.)
6.3. ALGORITHMS FOR BYZANTINE FAILURES

Figure 6.4: Execution \( \alpha_1 \)—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.

\textit{Execution}\ \( \alpha_1 \):

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 \( \alpha_1 \). In this execution, the validity condition requires that processes 1 and 2 both decide 1.

Now consider a second execution.
Figure 6.5: Execution $\alpha_2$—false message is circled.

Execution $\alpha_2$:
This is symmetric to $\alpha_1$. 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 $\alpha_2$. In this execution, the validity condition requires that processes 2 and 3 both decide 0.

To get a contradiction, consider a third execution.

Execution $\alpha_3$:
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 $\alpha_3$ as it does in $\alpha_1$, and sends the same messages to 3 in $\alpha_3$ as it does in $\alpha_2$, in both rounds. In fact, it is easy to check that $\alpha_3$ and $\alpha_1$ are indistinguishable to process 1, $\alpha_3 \sim \alpha_1$, and similarly $\alpha_3 \sim \alpha_2$. Since process 1 decides 1 in $\alpha_1$, it also does so in $\alpha_3$, and since process 3 decides 0 in $\alpha_2$, it also does so in $\alpha_3$. But this violates the agreement condition for $\alpha_3$, 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.
6.3. ALGORITHMS FOR BYZANTINE FAILURES

Figure 6.6: Execution $\alpha_3$—conflicting messages are circled.

Note that process 1, for example, can tell that some process is faulty in $\alpha_3$, 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 EIG-Byz. 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 \( \text{val} \) assignment so that any \( \text{null} \) 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 \( \text{newval} \), as follows. For each leaf labelled \( x \), \( \text{newval}(x) := \text{val}(x) \). For each non-leaf node labelled \( x \), \( \text{newval}(x) \) is defined to be the \( \text{newval} \) held by a strict majority of the children of node \( x \), that is, the element \( v \in V \) such that \( \text{newval}(xj) = v \) for a majority of the nodes of the form \( xj \), provided that such a majority exists. If no majority exists, process \( i \) sets \( \text{newval}(x) := v_0 \). Process \( i \)'s final decision is \( \text{newval}(\lambda) \).

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 \( \text{val}(x)_i = \text{val}(x)_j \) for every label \( x \) ending in \( k \).

**Proof.** If \( k \not\in \{i,j\} \), 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. \( \Box \)

The next lemma asserts that all nonfaulty processes agree on the \( \text{newvals} \) 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 \( \text{val}(x)_i = \text{newval}(x)_i = v \) 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 \( \text{val}(x)_i \); call this common value \( v \). Then also \( \text{newval}(x)_i = v \) for every nonfaulty process \( i \), by the definition of \( \text{newval} \) 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 \( \text{val}(x)_i \); call this value \( v \). Therefore, every nonfaulty process \( l \) sends the same value \( v \) for \( x \) to all processes, at round \( r + 1 \), so \( \text{val}(xl)_i = v \) for all nonfaulty \( i \) and \( l \). Then the inductive hypothesis implies that also \( \text{newval}(xl)_i = v \) 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 \( i \), \( \text{newval}(xl)_i = v \) for a majority of children \( xl \) of node \( x \). Then the majority rule used in the algorithm implies that \( \text{newval}(x)_i = v \) for all nonfaulty \( i \). So \( v \) is the required value. \( \square \)

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 \( \text{val}(j)_i = v \) for all nonfaulty processes \( i \) and \( j \). Lemma 6.16 implies that \( \text{newval}(j)_i = v \) for all nonfaulty \( i \) and \( j \). Then the majority rule used in the algorithm implies that \( \text{newval}(\lambda)_i = v \) for all nonfaulty \( i \). Therefore, \( i \)'s decision is \( v \), as needed. \( \square \)

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 \( \alpha \) of the EIGByz algorithm. A tree node \( x \) is said to be common in \( \alpha \) provided that at the end of \( f + 1 \) rounds in \( \alpha \), all the nonfaulty processes \( i \) have the same \( \text{newval}(x)_i \). A set of tree nodes (e.g., a path covering) is said to be common in \( \alpha \) if all the nodes in the set are common in \( \alpha \). 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 $\alpha$ of EIGByz, there exists a path covering that is common in $\alpha$.

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 $x_l$ of $x$. Since $x \notin C$, $C$ induces a common path covering for the subtree rooted at $x_l$. So by the inductive hypothesis, $x_l$ is common. Since $x_l$ 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 $\lambda$ is common.


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. □
6.3. ALGORITHMS FOR BYZANTINE FAILURES

Complexity analysis. The costs are the same as for the \textit{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: \( n > 3f \).

6.4 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 \textit{OptFloodSet} algorithm, which requires \( f + 1 \) rounds, \( 2n^2 \) messages, and \( O \left( n^2b \right) \) bits of communication. For the Byzantine case, the \textit{EIGByz} algorithm uses \( f + 1 \) rounds and an exponential amount of communication, while \textit{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
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 $\alpha$ be the resulting execution of $S$.

We first consider execution $\alpha$ 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 $\alpha_1$ in which process 1 is faulty. That is, $\alpha$ and $\alpha_1$ are indistinguishable to processes 2 and 3, $\alpha \equiv \alpha_1$ and $\alpha \not\equiv \alpha_1$, according to the definition of “indistinguishable” in Section 2.4. See Figure 6.8. In $\alpha_1$, process 1 exhibits a peculiar type of faulty behavior—it behaves like the combination of processes $1'$, $2'$, $3'$, and 1 in $\alpha$. Although it is peculiar, it is an allowable behavior for a faulty process in $A$, under the assumptions for Byzantine faults.

Since $\alpha_1$ 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 $\alpha_1$, processes 2 and 3 must decide 0. Since $\alpha$ is indistinguishable from $\alpha_1$ to processes 2 and 3, both decide 0 in $\alpha$ as well.

Next consider execution $\alpha$ 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 $\alpha_2$ in which process 3 is faulty. That is, $\alpha \not\equiv \alpha_2$ and $\alpha \equiv \alpha_2$. 
Figure 6.8: Executions $\alpha$ and $\alpha_1$ are indistinguishable to processes 2 and 3.

Figure 6.9: Executions $\alpha$ and $\alpha_2$ 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 $\alpha$.

Finally, consider execution $\alpha$ 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 $\alpha_3$ in which process 2 is faulty. That is, $\alpha \sim^3 \alpha_3$ and $\alpha \sim^1 \alpha_3$. See Figure 6.10. By the correctness conditions for Byzantine agreement, processes 3 and 1' must eventually decide in $\alpha_3$, and their decisions must be the same. Because process 3 starts with input 0 and process 1' starts with input 1, there is
no requirement about what value they agree upon, but the agreement condition implies that they agree. Therefore, they decide on the same value in \( \alpha \) also.

But this is a contradiction, because we have already observed that in \( \alpha \), 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 \( \alpha \) of \( B \) with at most one faulty process and let \( \alpha' \) 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 \( \alpha' \). 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 \( \alpha' \).

We argue that these conditions carry over to \( \alpha \). 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 \( \alpha' \) 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 \( \alpha' \) implies that \( v \) is the only decision value for a nonfaulty process in \( \alpha' \). Then \( v \) is the only decision value for a nonfaulty process in \( \alpha \).

For agreement, suppose \( i \) and \( j \) are nonfaulty processes of \( B \). Then they simulate only nonfaulty processes of \( A \). Agreement for \( \alpha' \) 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. \( \Box \)

\(^3\)We invoke the technicality that Byzantine faulty processes are allowed to behave completely correctly, in order to justify this classification.