CHAPTER 8

AGREEMENT PROTOCOLS

8.1 INTRODUCTION

In distributed systems, where sites (or processors) often compete as well as cooperate to achieve a common goal, it is often required that sites reach mutual agreement. For example, in distributed database systems, data managers at sites must agree on whether to commit or to abort a transaction [11]. Reaching an agreement typically requires that sites have knowledge about the values of other sites. For example, in *distributed commit*, a site should know the outcome of *local commit* at each site.

When the system is free from failures, an agreement can easily be reached among the processors (or sites). For example, processors can reach an agreement by communicating their values to each other and then by taking a majority vote or a minimum, maximum, mean, etc. of those values. However, when the system is prone to failure, this method does not work. This is because faulty processors can send conflicting values to other processors preventing them from reaching an agreement. In the presence of faults, processors must exchange their values with other processors and relay the values received from other processors several times to isolate the effects of faulty processors. A processor refines its value as it learns of the values of other processors (This entire process of reaching an agreement is called an *agreement protocol*).

In this chapter, we study agreement protocols for distributed systems under processor failures. A very general model of faults is assumed. For example, a faulty processor may send spurious messages to other processors, may lie, may not respond to received messages correctly, etc. Also, nonfaulty processors do not know which processors are faulty.

In agreement problems, nonfaulty processors in a distributed system should be able to reach a common agreement, even if certain components in the system are faulty. The agreement is achieved through an agreement protocol that involves several rounds of message exchange among the processors.

8.2 THE SYSTEM MODEL

Agreement problems have been studied under the following system model:

- There are n processors in the system and at most m of the processors can be faulty.
- The processors can directly communicate with other processors by message passing. Thus, the system is logically fully connected.
- A receiver processor always knows the identity of the sender processor of the message.
- The communication medium is reliable (i.e., it delivers all messages without introducing any errors) and only processors are prone to failures.

For simplicity, we assume that agreement is to be reached between only two values, 0 and 1. Results can easily be extended to multivalue agreement [23].

Early solutions to agreement problems assumed that only processors could be faulty and that communication links did not fail. Limiting faults solely to the processors simplifies the solution to agreement problems. Recently, agreement problems have been studied under the failure of communication links only [24] and under the failure of both processors and communication links [25] In this chapter, we limit the treatment of agreement problems solely to processor failures.

8.2.1 Synchronous vs. Asynchronous Computations

In a synchronous computation, processes in the system run in lock step manner, where in each step, a process receives messages (sent to it in the previous step), performs a computation, and sends messages to other processes (received in the next step). A step of a synchronous computation is also referred to as a *round*. In synchronous computation, a process knows all the messages it expects to receive in a round. A message delay or a slow process can slow down the entire system or computation.

In an asynchronous computation, on the other hand, the computation at processes does not proceed in lock steps. A process can send and receive messages and perform computation at any time.

In this chapter, synchronous models of computation are assumed. The assumption of synchronous computation is critical to agreement protocols. In fact, the agreement problem is not solvable in an asynchronous system, even for a single processor failure [10].

8.2.2 Model of Processor Failures

In agreement problems, we consider a very general model of processor failures. A processor can fail in three modes: *crash fault, omission fault*, and *malicious fault.* In a crash fault, a processor stops functioning and never resumes operation. In an omission fault, a processor "omits" to send messages to some processors. (These are the messages that the processor should have sent according to the protocol or algorithm it is executing.) For example, a processor is supposed to broadcast a message to all other processors, but it sends the message to only a few processors. In a malicious fault, a processor behaves randomly and arbitrarily. For example, a processor may send fictitious messages to other processors to confuse them. Malicious faults are very broad in nature and thus most other conceivable faults can be treated as malicious faults. Malicious faults are also referred to as *Byzantine faults.*

Since a faulty processor can refuse to send a message, a nonfaulty processor may never receive an expected message from a faulty processor. In such a situation, we assume that the nonfaulty processor simply chooses an arbitrary value and acts as if the expected message has been received [16]. Of course, we assume that such situations, where a processor refuses to send a message, can be detected by the respective receiver processors. In synchronous systems, if the duration of each round is known, then this detection is simple—all the expected messages not received by the end of a round were not sent.

8.2.3 Authenticated vs. Non-Authenticated Messages

Note that to reach an agreement, processors have to exchange their values and relay the received values to other processors several times. The capability of faulty processors to distort what they receive from other processors greatly depends upon the type of underlying messages.

There are two types of messages: *authenticated and non-authenticated.* In an authenticated message system, a (faulty) processor cannot forge a message or change the contents of a received message (before it relays the message to other processors). A processor can verify the authenticity of a received message. An authenticated message is also called a *signed* message [14].

In a non-authenticated message system, a (faulty) processor can forge a message and claim to have received it from another processor or change the contents of a received message before it relays the message to other processors. A processor has no way of verifying the authenticity of a received message. A non-authenticated message is also called an *oral* message [14]. It is easier to reach agreement in an authenticated message system because faulty processors are capable of doing less damage.

8.2.4 Performance Aspects

The performance (or the computational complexity) of agreement protocols is generally determined by the following three metrics: *time, message traffic*, and *storage overhead.* Time refers to the time taken to reach an agreement under a protocol. The time is usually expressed as the number of rounds needed to reach an agreement. Message traffic is measured by the number of messages exchanged to reach an agreement. Sometimes, the message traffic is also measured by the total number of bits exchanged to reach an agreement [5]. Storage overhead measures the amount of information that needs to be stored at processors during the execution of a protocol.

Next, we discuss three agreement problems for non-authenticated messages under processor failures.

8.3 A CLASSIFICATION OF AGREEMENT PROBLEMS

There are three well known agreement problems in distributed systems: the *Byzantine agreement problem*, the *consensus problem*, and the *interactive consistency problem.* In the Byzantine agreement problem, a single value, which is to be agreed on, is initialized by an arbitrary processor and all nonfaulty processors have to agree on that value. In the consensus problem, every processor has its own initial value and all nonfaulty processors must agree on a single common value. In the interactive consistency problem, every processor has its own initial value and all nonfaulty processors must agree on a *set* of common values.

In all three problems, all nonfaulty processors must reach a common agreement. In the Byzantine agreement and the consensus problems, the agreement is about a single value. Whereas in the interactive consistency problem, the agreement is about a set of common values. In the Byzantine agreement problem, only one processor initializes the initial value. Whereas in the consensus and the interactive consistency problems, every processor has its own initial value. Table 8.1 summarizes the starting values and final outcomes of the three problems.

Next, we define these three agreement problems in a precise manner.

8.3.1 The Byzantine Agreement Problem

In the Byzantine agreement problem, an arbitrarily chosen processor, called the *source processor*, broadcasts its initial value to all other processors. A solution to the Byzantine agreement problem should meet the following two objectives:

Agreement. All nonfaulty processors agree on the same value.

Validity. If the source processor is nonfaulty, then the common agreed upon value by all nonfaulty processors should be the initial value of the source.

TABLE 8.1
The three agreement problems

Problem →	**Byzantine** Agreement	**Consensus**	**Interactive** Consistency
Who initiates the value	One processor	All processors	All processors
Final agreement	Single value	Single value	A vector of values

Two points should be noted: (1) If the source processor is faulty, then all nonfaulty processors can agree on *any* common value. (2) It is irrelevant what value faulty processors agree on or whether they agree on a value at all.

8.3.2 The Consensus Problem

In the consensus problem, every processor broadcasts its initial value to all other processors. Initial values of the processors may be different. A protocol for reaching consensus should meet the following conditions:

Agreement All nonfaulty processors agree on the same single value.

Validity If the initial value of every nonfaulty processor is v, then the agreed upon common value by all nonfaulty processors must be v.

Note that if the initial values of nonfaulty processors are different, then all nonfaulty processors can agree on *any* common value. Again, we don't care what value faulty processors agree on.

8.3.3 The Interactive Consistency Problem

In the interactive consistency problem, every processor broadcasts its initial value to all other processors. The initial values of the processors may be different. A protocol for the interactive consistency problem should meet the following conditions:

Agreement. All nonfaulty processors agree on the same vector, $(v_1, v_2, ..., v_n)$.

Validity. If the ith processor is nonfaulty and its initial value is v_i, then the ith value to be agreed on by all nonfaulty processors must be v_i.

Note that if the jth processor is faulty, then all nonfaulty processors can agree on *any* common value for v_j. It is irrelevant what value faulty processors agree on.

8.3.4 Relations Among the Agreement Problems

All three agreement problems are closely related [7]. For example, the Byzantine agreement problem is a special case of the interactive consistency problem, in which the initial value of only one processor is of interest. Conversely, if each of the n processors runs a copy of the Byzantine agreement protocol, the interactive consistency problem is solved. Likewise, the consensus problem can be solved using the solution of the interactive consistency problem. This is because all nonfaulty processors can compute the value that is to be agreed upon by taking the majority value of the common vector that is computed by an interactive consistency protocol, or by choosing a default value if a majority does not exist.

Thus, solutions to the interactive consistency and consensus problems can be derived from the solutions to the Byzantine agreement problem. In other words, the Byzantine agreement problem is primitive to the other two agreement problems. For this reason, we will focus solely on the Byzantine agreement problem for the rest of the chapter.

However, it should by no means be concluded that the Byzantine agreement problem is weaker than the interactive consistency problem or that the interactive consistency problem is weaker than the consensus problem. In fact, there is no linear ordering of this sort among these agreement problems. For example, the Byzantine agreement problem can be solved using a solution to the consensus problem in the following manner [7]:

1. The source processor sends its value to all other processors, including itself.
2. All the processors, including the source, run an algorithm for the consensus problem using the values received in the first step as their initial values.

The above two steps solve the Byzantine agreement problem because (1) if the source processor is nonfaulty, then all the processors will receive the same value in the first step and all nonfaulty processors will agree on that value as a result of the consensus algorithm in the second step, and (2) if the source processor is faulty, then the other processors may not receive the same value in the first step; However, all nonfaulty processors will agree on the same value in the second step as a result of the consensus algorithm. Thus, the Byzantine agreement is reached in both cases. However, the $n - 1$ extra messages are sent at the first step.

8.4 SOLUTIONS TO THE BYZANTINE AGREEMENT PROBLEM

The Byzantine agreement problem was first defined and solved (under processor failures) by Lamport et al. [14, 16]. Recall that in this problem, an arbitrarily chosen processor (called the source processor) broadcasts its initial value to all other processors. A protocol for the Byzantine agreement should guarantee that all nonfaulty processors agree on the same value and if the source processor is nonfaulty, then the common agreed upon value by all nonfaulty processors should be the initial value of the source.

It is obvious that all the processors must exchange the values through messages to reach a consensus. Processors send their values to other processors and relay received values to other processors [16]. During the execution of the protocol, faulty processors may confuse other processors by sending them conflicting values or by relaying to them fictitious values.

The Byzantine agreement problem is also referred to as the Byzantine *generals* problem ([4, 14]) because the problem resembles a situation where a team of generals in an army is trying to reach agreement on an attack plan. The generals are located at geographically distant positions and communicate only through messengers. Some of the generals are traitors (equivalent to faulty processers) and try to prevent loyal generals from reaching an agreement by deliberately transmitting erroneous information.

8.4.1 The Upper Bound on the Number of Faulty Processors

In order to reach an agreement on a common value, nonfaulty processors need to be free from the influence of faulty processors. If faulty processors dominate in number, they can prevent nonfaulty processors from reaching a consensus. Thus, the number of faulty processors should not exceed a certain limit if a consensus is to be reached.

Pease et al. [16] showed that in a fully connected network, it is impossible to reach a consensus if the number of faulty processors, m, exceeds $\lfloor (n-1)/3 \rfloor$. Lamport et al. [14] were the first to give a protocol to reach Byzantine agreement that requires $m+1$ rounds of message exchanges (m is the maximum number of faulty processors). Fischer et al. [8] showed that $m+1$ is the lower bound on the number of rounds of message exchanges to reach a Byzantine agreement in a fully connected network where only processors can fail.

However, if authenticated messages are used, this bound is relaxed and a consensus can be reached for any number of faulty processors.

8.4.2 An Impossibility Result

We now show that a Byzantine agreement cannot be reached among three processors, where one processor is faulty [14]. For a rigorous treatment of this impossibility result for a higher number of processors, readers are referred to [9] and [16].

Consider a system with three processors, p_0, p_1, and p_2. For simplicity, we assume that there are only two values, 0 and 1, on which processors agree and processor p_0 initiates the initial value. There are two possibilities: (1), p_0 is not faulty or (2) p_0 is faulty.

Case I: p_0 is not faulty. Assume p_2 is faulty. Suppose that p_0 broadcasts an initial value of 1 to both p_1 and p_2. Processor p_2 acts maliciously and communicates a value of 0 to processor p_1. Thus, p_1 receives conflicting values from p_0 and p_2. (This scenario is shown in Fig. 8.1. A faulty processor is depicted by an oval and a nonfaulty processor is denoted by a circle.) However, since p_0 is nonfaulty, processor p_1 must accept 1 as the agreed upon value if condition 2 (of Sec. 8.3.1) is to be satisfied.

Case II: p_0 is faulty. Suppose that processor p_0 sends an initial value of 1 to p_1 and 0 to p_2. Processor p_2 will communicate the value 0 to p_1. (This scenario is shown in Fig. 8.2). As far as p_1 is concerned, this case will look identical to Case I. So any agreement protocol which works for three processors cannot distinguish between the two cases and must force p_1 to accept 1 as the agreed upon value whenever p_1 is faced with such situations (to satisfy condition 2). However, in Case II, this will work only if p_2 is also made to accept 1 as the agreed upon value.

Using a similar argument, we can show that if p_2 receives an initial value of 0 from p_0, then it must take 0 as the agreed upon value, even if p_1 communicates a value

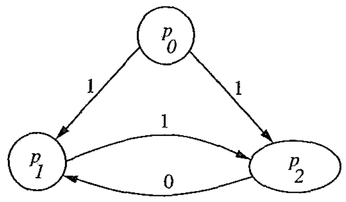

FIGURE 8.1
Processor p_0 is Non-Faulty.

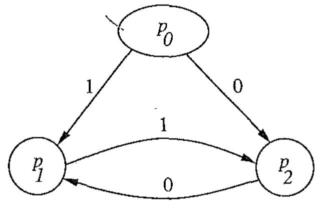

FIGURE 8.2
Processor p_0 is Faulty.

of 1. However, if this is followed in Case II, p_1 will agree on a value of 1 and p_2 will agree on a value of 0, which will violate condition 1 (Sec. 8.3.1) of the solution.

Therefore, no solution exists for the Byzantine agreement problem for three processors, which can work under single processor failure.

8.4.3 Lamport-Shostak-Pease Algorithm

Lamport et al.'s algorithm [14], referred to as the Oral Message algorithm OM(m), $m > 0$, solves the Byzantine agreement problem for $3m + 1$ or more processors in the presence of at most m faulty processors. Let n denote the total number of processors (clearly, $n \geq 3m + 1$). The algorithm is recursively defined as follows:

Algorithm OM(0).

1. The source processor sends its value to every processor.
2. Each processor uses the value it receives from the source. (If it receives no value, then it uses a default value of 0.)

Algorithm OM(m), $m > 0$.

1. The source processor sends its value to every processor.
2. For each i, let v_i be the value processor i receives from the source. (If it receives no value, then it uses a default value of 0.). Processor i acts as the new source and initiates **Algorithm OM(m-1)** wherein it sends the value v_i to each of the $n - 2$ other processors.
3. For each i and each j ($\neq i$), let v_j be the value processor i received from processor j in Step 2. using Algorithm OM(m−1). (If it receives no value, then it uses a default value of 0.). Processor i uses the value *majority*(v_1, v_2, ..., v_{n-1}).

This algorithm is evidently quite complex. The processors are successively divided into smaller and smaller groups and the Byzantine agreement is recursively achieved within each group of processors (Step 2 of "Algorithm OM($m-1$)"). Step 3 is executed during the folding phase of the recursion, where a *majority* function is applied to select the majority value out of the values received in a round of message exchange (Step 2). The function *majority* (v_1, v_2, ..., v_{n-1}) computes a majority value of the values v_1, v_2, ..., v_{n-1} if it exists (otherwise, it returns the default value 0).

The execution of the algorithm OM(m) invokes $n-1$ separate executions of the algorithm OM($m-1$), each of which invokes $n-2$ executions of the algorithm OM(m-2), and so on. Therefore, there are $(n-1)(n-2)(n-3)$ ... $(n-m+\cancel{1})^{k}$ separate executions of the algorithm OM(k), k = $n-1$, $n-2$, $n-3$, ..., $n-m+1$. The message complexity of the algorithm is O(n^m).

Example 8.1. Consider a system with four processors, p_0, p_1, p_2, and p_3. For simplicity, we assume that there are only two values 0 and 1, furthermore, we assure that processor p_0 initiates the initial value and that processor p_2 is faulty.

To initiate the agreement, processor p_0 executes algorithm OM(1) wherein it sends its value 1 to all other processors (Fig. 8.3). At Step 2 of the algorithm OM(1), after having received the value 1 from the source processor p_0, processors p_1, p_2, and p_3 execute the algorithm OM(0). These executions are shown in Fig. 8.4. Processors p_1 and p_3 are nonfaulty and send value 1 to processors $\{p_2, p_3\}$ and $\{p_1, p_2\}$, respectively. Faulty processor p_2 sends value 1 to p_1 and a value 0 to p_3 (see Fig. 8.4).

After having received all the messages, processors p_1, p_2, and p_3 execute Step 3 of the algorithm OM(1) to decide on the majority value. Processor p_1 has received values (1, 1, 1), whose majority value is 1, processor p_2 has received values (1, 1, 1), whose majority value is 1, and processor p_3 has received values (1, 1, 0), whose majority value is 1. Thus, both conditions of the Byzantine agreement are satisfied.

Example 8.2. Figure 8.5 shows a situation where processor p_0 is faulty and sends conflicting values to the other three processors. These three processors, under Step 1 of OM(0), send the received values to the other two processors.

After having received all the messages, processors p_1, p_2, and p_3 execute Step 3 of the algorithm OM(1) to decide on the majority value. Note that all three processors have received values (1, 0, 1), whose majority value is 1, Thus, all three nonfaulty processors agree on the same value and the required conditions of the Byzantine agreement are satisfied.

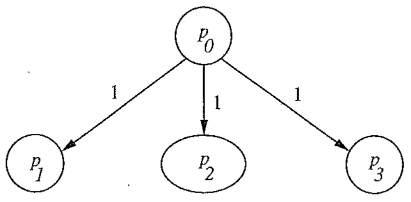

FIGURE 8.3
Processor p_0 executes the algorithm OM(1)

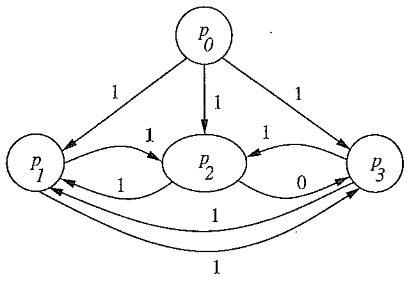

FIGURE 8.4
Processors p_1, p_2, and p_3 execute the algorithm OM(0)

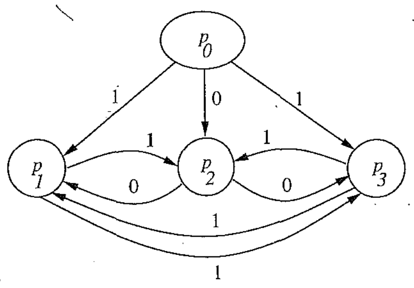

FIGURE 8.5
Processors p_0, p_1, p_2, and p_3 execute OM(1) and OM(0)

8.4.4 Dolev et al.'s Algorithm

Dolev et al. have given a polynomial algorithm for reaching Byzantine agreement [5]. The algorithm is polynomial in message complexity. However, the algorithm requires up to $2m+3$ rounds to reach an agreement (more than what is needed in the Lamport-Shostak-Pease algorithm). Thus, there is a trade off between message complexity and time delay (rounds).

DATA STRUCTURES. The algorithm uses two thresholds: LOW and HIGH, where LOW:= $m+1$ and HIGH:= $2m+1$. The basic idea is that any subset of processors of size LOW will have at least one nonfaulty processor. Therefore, we can prevent faulty processors from introducing erroneous values by confirming an assertion from at least LOW number of processors. Note that if an assertion is supported by LOW number of processors, then it must be supported by at least one nonfaulty processor. Also, any subset of processors of size HIGH includes a majority of processors, that is, $m+1$, that are nonfaulty. Therefore, an assertion must be confirmed by at least HIGH number of processors before assuming an agreement on that assertion.

The algorithm uses two types of messages: a "$*$" message and a message consisting of the name of a processor. The "$*$" denotes the fact that the sender of the message is sending a value of 1 and the name in a message denotes the fact that the sender of the message received a "$*$" from the named processor. (Note that every message contains the name of its sender processor.)

A processor keeps a record of all the messages it has received. Let W_x^i be the set of processors that have sent message x to processor i. (Note that x is either a "$*$" or a processor name.) Thus, each process maintains $n+1$ number of W sets. We refer to W_x^i as the set of *witnesses* to message x for processor i. A processor j is a *direct supporter* for a processor k if j directly receives "$*$" from k. When a nonfaulty processor j directly receives "$*$" from processor k, it sends message "k" to all other processors. When processor i receives the message "k" from processor j, it adds j into W_k^i because j is a witness to message "k". Process j is an *indirect supporter* for processor k if $|W_k^j| \geq$ LOW; That is, processor j has received message "k" from at least LOW number of processors. A processor j *confirms* processor k if $|W_k^j| \geq$ HIGH; That is, at least HIGH number of processors told processor j that they received the value of a 1 from processor k. A process i maintains a set, C_i, of confirmed processors.

THE ALGORITHM. In the first round, the source processor sends a "$*$" message to all processors (including itself) if its value is 1. If its value is 0, it sends nothing in the first round. (The default value is 0.) If the processors finally agree on "$*$", then the agreed upon value is 1. Otherwise, the agreed upon value is 0.

In subsequent rounds, a processor sends its messages to all other processors, receives messages from other processors, and then decides what messages to send in the next round. Recall that when a nonfaulty processor j receives a "$*$" message from processor k, it sends message "k" to all other processors in the next round indicating that it is a direct supporter of processor "k". Next, we introduce the *initiation* operation, which is tantamount to sending a "$*$" message to all other processors. A processor *initiates* under the following conditions:

- It initiates in the second round if it receives a "$*$" from the source in round 1.
- It initiates in the $K+1$st round if at the end of Kth round the cardinality of the set of the confirmed processors (not including the source) is at least LOW+ max(0, $\lfloor K/2 \rfloor - 2$) (referred to as the *condition of initiation*).

The following four rules describe the operation of the Dolev et al.'s algorithm [5]:

1. In the first round, the source broadcasts its value to all other processors.
2. In a round $k > 1$, a processor broadcasts the names of all processes for which it is either a direct or indirect supporter and which it has not previously broadcast. If the condition of initiation was true at the end of the previous round, it also broadcasts the "$*$" message unless it has previously done so.
3. If a processor confirms HIGH number of processors, it commits to a value of 1.
4. After round $2m+3$, if the value 1 is committed, the processors agree on 1; otherwise, they agree on 0.

DISCUSSION. The algorithm has two interesting features: initiation and committing. A processor commits if it confirms HIGH number of processors; That is, each of these HIGH confirmed processors has been witnessed to have sent a "$*$" by HIGH number of processors. Since there are at least $m+1$ nonfaulty processors in any HIGH number of processors, a processor commits if it determines that at least $m+1$ nonfaulty processors have witnessed that at least $m+1$ nonfaulty confirmed processors sent a "$*$". This means $m+1$ nonfaulty processors have indeed initiated (i.e., have broadcast a "$*$").

The initiation is a very interesting concept as it causes a chain reaction of the initiation operation to make nonfaulty processors eventually dominate the faulty processors. A processor initiates in the second round if it receives a "$*$" from the source in the first round. A processor can initiate after the first round, only if it has confirmed sufficiently many processors. This number is LOW for the first four rounds and after that it increases by one for every two rounds. The algorithm and the condition for initiation are so designed that if LOW number of nonfaulty processors initiate, an avalanche of initiation begins, causing all nonfaulty processors to initiate and commit.

Example 8.3. Consider a situation with $3m+1$ processors out of which m processors are faulty. Suppose the source is nonfaulty and broadcasts a "$*$" in the first round. In the second round, $2m$ nonfaulty processors will initiate (i.e., broadcast "$*$"). In the third round, $2m + 1$ nonfaulty processors (including the source) will broadcast messages containing the name of the processors (direct supporter messages) informing that they have witnessed a "$*$" from $2m$ other nonfaulty processors. Thus, in the fourth round, the witness set of all $2m + 1$ nonfaulty processors will contain all $2m + 1$ nonfaulty processors and they all will commit to a value of 1 in the fourth round. Thus, if the source is nonfaulty, an agreement is reached in four rounds [5].

If the source is faulty, it may send a "$*$" to only a few processors. In this case, at least all the nonfaulty processors which received a "$*$" in the first round will initiate in the second round. Note that until LOW number of nonfaulty processors initiate, there is no guarantee that the confirmed set at a processor will reach size LOW, triggering an initiation at some other nonfaulty processor. Some faulty processors can always behave like nonfaulty processors that receive "$*$" from the source and collaborate with nonfaulty processors to help them trigger initiation at other nonfaulty processors. However, there is no guarantee of this. On the other hand, if a faulty source sends a "$*$" to at least LOW number of nonfaulty processors in the first round, then an avalanche of initiation at nonfaulty processors occurs, resulting in the commit of all nonfaulty processors.

EXTENSION TO CASE N $>$ 3M + 1. So far, it has been assumed that out of $n = 3m + 1$ processors, exactly m processors are faulty. Now we extend the result for the case where $n > 3m + 1$; that is, the number of nonfaulty processors is more than the lower bound.

When the number of processors n is greater than $3m + 1$, the application of the above algorithm will exchange more messages than necessary. To reduce the total number of messages exchanged to reach a consensus, $3m+1$ processors are designated as *active* processors (including the source) and the rest of the processors are called *passive* processors. The passive processors do not send any messages and a processor ignores a message about or from a passive processor. (Note that some faulty processors can forge messages.)

All active processors follow the above described algorithm. They send "$*$" messages to all processors (active as well as passive), but send all other messages containing names only to all other active processors. A passive processor agrees on 1 if it receives a "$*$" from HIGH number of active processors; Otherwise, it agrees on 0. All the active processors will reach the Byzantine agreement after $2m + 3$ rounds. It is shown in [5] that both active and passive processors together reach the Byzantine agreement as a result of the above algorithm.

8.5 APPLICATIONS OF AGREEMENT ALGORITHMS

Algorithms for agreement problems find applications in problems where processors should reach an agreement regarding their values in the presence of malicious failures. We next discuss two such applications, viz., clock synchronization in distributed systems and atomic commit in distributed databases.

8.5.1 Fault-Tolerant Clock Synchronization

In distributed systems, it is often necessary that sites (or processes) maintain physical clocks that are synchronized with one another. Since physical clocks have a drift problem, they must be periodically resynchronized. Such periodic synchronization becomes extremely difficult if the Byzantine failures are allowed. This is because a faulty process can report different clock values to different processes. The description of clock synchronization in this section is based on the work of Lamport and Melliar-Smith [13]. We make the following assumptions regarding the system:

A1: All clocks are initially synchronized to approximately the same values.

A2: A nonfaulty process's clock runs at approximately the correct rate. (The correct rate means one second of clock time per second of real time. No assumption is made about a faulty clock.)

A3: A nonfaulty process can read the clock value of another nonfaulty process with at most a small error ϵ.

A clock synchronization algorithm should satisfy the following two conditions:

- At any time, the values of the clocks of all nonfaulty processes must be approximately equal.
- There is a small bound on the amount by which the clock of a nonfaulty process is changed during each resynchronization.

The latter condition implies that resynchronization does not cause a clock value to jump arbitrarily far, thereby preventing the clock rate from being too far from the real time.

We discuss two clock synchronization algorithms, namely, the interactive convergence algorithm and the interactive consistency algorithm, which are fault-tolerant to the Byzantine failures. The former gets its name because it causes the nonfaulty clocks to converge and the latter gets its name because all nonfaulty processes obtain mutually consistent views of all the clocks. Both the algorithms can tolerate up to m process failures in a network of at least $3m + 1$ processes.

The Interactive Convergence Algorithm

The interactive convergence algorithm assumes that the clocks are initially synchronized and that they are resynchronized often enough so that two nonfaulty clocks never differ by more than δ. The algorithm works as follows:

The Algorithm. Each process reads the value of all other processes' clocks and sets its clock value to the average of these values. However, if a clock value differs from its own clock value by more than δ, it replaces that value by its own clock value when taking the average.

Clearly, the algorithm is conceptually very simple. It does not safeguard against the problem of *two-faced* clocks wherein a faulty clock reports different values to different processes. We next show that, nonetheless, the algorithm brings the clocks of nonfaulty processes closer. Let two processes p and q, respectively, use c_{pr} and c_{qr} as the clock values of a third process r when computing their averages. If r is nonfaulty, then $c_{pr} = c_{qr}$. If r is faulty, then $|c_{pr} - c_{qr}| \leq 3\delta$ (because the difference in the clock values of any two processes is bounded by δ). When p and q compute their averages for the n clocks values, they both use identical values for the clocks of $n-m$ nonfaulty processes and the difference in the clock values of m faulty processes they use is bounded by 3δ. Consequently, the averages computed by p and q differ by at most $(3m/n)\delta$. Since $n > 3m$, clearly $(3m/n)\delta < \delta$. Thus, each resynchronization brings the clocks closer by a factor of $(3m/n)$. This implies that we can keep the clocks synchronized within any desired degree by resynchronizing them often enough using the algorithm.

In the above discussion of this algorithm, two assumptions have been made:

- All processes execute the algorithm instantaneously at exactly the same time.
- The error in reading another process's clock is zero.

Since a process may not execute the algorithm instantaneously, it may read other processes' clocks at different time instants. This problem can be circumvented by having a process compute the average of the difference in clock values (rather than using absolute clock values) and incrementing its clock by the average increment. (Clock differences larger than δ are replaced by 0.) However, this requires the following assumption:

A3′: A nonfaulty process can read the difference between the clock value of another nonfaulty process and its own with at most a small error ϵ.

If the clock-reading error is ϵ, then the difference in the clock values read by a process can be as large as $\delta + \epsilon$. Therefore, only clock differences larger than $\delta + \epsilon$ are replaced by 0 while computing the average increment.

The Interactive Consistency Algorithm

The interactive consistency algorithm adds two improvements: first, it takes the median of the clock values rather than the mean. The median provides a good estimate of the clock value, as the number of bad clocks will be low. Second, it avoids the problem of two-faced clocks (which report different values to different processes) by using a more sophisticated technique to obtain clock values of the processes.

Two processes will compute approximately the same median if they obtain approximately the same set of clock values for other processes. Therefore, the following conditions apply:

C1: Any two processes obtain approximately the same value for a process p's clock (even if p is faulty).

Not only should all processes compute the same value, but their values should be close to the clock values of nonfaulty processes. Therefore, the following condition:

C2: If q is a nonfaulty process, then every nonfaulty process obtains approximately the correct value for process q's clock.

Thus, if a majority of the processes are nonfaulty, the median of all the clock values is either approximately equal to a good clock's value or it lies between the values of two good clocks.

Note that conditions C1 and C2 are very similar to the Agreement and Validity conditions of the interactive consistency problem of Sec. 8.3.3. Therefore, the interactive consistency algorithm for clock synchronization works in the following manner: first, all the processes execute an algorithm for the interactive consistency problem to collect the values of the clocks of other processes, which satisfy conditions C1 and C2. Second, every process uses the median of the collected values to compute its new clock value. The first step can be executed by having every process independently run an instance of the oral message protocol, OM(m), for the Byzantine problem, where a process sends a copy of its clock to every process as its initial value. When the algorithm OM(m) has stopped at every process, every process has clock values for all other processes, which satisfy conditions C1 and C2. At this point, every process computes the median of these values and sets its clock to the median.

8.5.2 Atomic Commit in DDBS

In the problem of atomic commit, sites of a DDBS must agree whether to commit or abort a transaction. In the first phase of the atomic commit, sites execute their part of a distributed transaction and broadcast their decisions (commit or abort) to all other sites. In the second phase, each site, based on what it received from other sites in the first phase, decides whether to commit or abort its part of the distributed transaction.

Since every site receives an identical response from all other sites, they will reach the same decision. However, if some sites behave maliciously, they can send a conflicting response to other sites, causing them to make conflicting decisions.

In these situations, we can use algorithms for the Byzantine agreement to insure that all nonfaulty processors reach a common decision about a distributed transaction. It works as follows: In the first phase, after a site has made a decision, it starts the Byzantine agreement. In the second phase, processors determine a common decision based on the agreed vector of values.

8.6 SUMMARY

In distributed systems, it is often required that sites (or processors) reach a mutual agreement. However, when Byzantine faults are permitted, solutions to the agreement problem are nontrivial because faulty processors may behave maliciously, preventing other processors from reaching a common agreement. In Byzantine failures, a faulty processor may send spurious messages to other processors, may lie, may not respond to received messages correctly, etc.

In this chapter, we studied agreement problems under a synchronous model of computation, where processors run in a lock step manner. The agreement problem is not solvable in an asynchronous system even for a single processor failure. In an asynchronous computation, computation at processors does not proceed in lock steps. There are two types of messages: authenticated vs. non-authenticated. In the authenticated message system, a faulty processor cannot forge a message or change the contents of a received message before it relays the message to other processors. In a non-authenticated message system, a faulty processor can forge a message and claim to have received it from another processor or change the contents of a received message before it relays the message to other processors. This chapter discussed agreement problems for non-authenticated messages under processor failures.

The agreement problems can be classified into three classes, namely, the Byzantine agreement problem, the consensus problem, and the interactive consistency problem. In the Byzantine agreement problem, a source processor initializes a value and all nonfaulty processors must agree on that value (if the source is non-faulty). In the consensus problem, every processor has its own initial value and all nonfaulty processors must agree on a common single value. In the interactive consistency problem, every processor has its own initial value and all non-faulty processors must agree on a *set* of common values.

In all three problems, all nonfaulty processors must reach a common agreement. In the Byzantine agreement and consensus problems, the agreement concerns a single value. Whereas in the interactive consistency problem, the agreement concerns a set of common values. In the Byzantine agreement problem, only one processor can initialize the initial value. Whereas in the consensus and the interactive consistency problems, every processor has its own initial value. The three agreement problems are related and the Byzantine agreement problem is primitive to the other two agreement problems.

An algorithm for Byzantine agreement must guarantee that all nonfaulty processors agree on the same value and if the source processor is nonfaulty, the common agreed upon value by all nonfaulty processors should be the initial value of the source. Obviously, all the processors must exchange values through messages to reach a consensus. Processors send their values to other processors and relay received values to other processors. A major problem is that during the execution of the protocol, faulty processors may confuse other processors by sending them conflicting values or may not relay the correct value.

In order to reach an agreement on a common value, nonfaulty processors should be free from the influence of faulty processors. If faulty processors dominate in number, they can prevent nonfaulty processors from reaching a consensus. Thus, the number of faulty processors should not exceed a limit if a consensus is to be reached. It is impossible to reach a consensus if the number of faulty processors, m, exceeds $\lfloor (n-1)/3 \rfloor$. (If authenticated messages are used, this bound is less rigid and a consensus can be reached for any number of faulty processors.) It has been shown that $m+1$ is the lower bound on the number of rounds of message exchanges to reach a Byzantine agreement in a fully connected network where only processors can fail.

Lamport et al. [14] were the first to present an algorithm to solve the Byzantine agreement problem for $3m+1$ or more processors in the presence of at most m faulty processors. In the algorithm, processors are recursively divided into smaller and smaller groups and the Byzantine agreement is recursively achieved within each group of processors. The message complexity of this algorithm is O(n^m) and it requires $m+1$ rounds to reach the consensus. Dolev et al. gave a polynomial algorithm for reaching the Byzantine agreement [5]. Their algorithm is polynomial in message complexity, but it requires up to $2m + 3$ rounds to reach an agreement (which is more than double the rounds needed in the Lamport et al.'s algorithm). Thus, there is a tradeoff between message complexity and time delay (rounds).

Algorithms for agreement problems find applications in problems where processors should reach an agreement in the presence of malicious failures. Example of these applications include clock synchronization, atomic commit in DDBS, and fault tolerance.

8.7 FURTHER READING

Two surveys on the Byzantine agreement problem appear in papers by Fischer [7] and Strong-Dolev [21]. For the extension of binary-value Byzantine agreement to multivalue agreement, readers should see a paper by Turpin and Coan [23]. Yan et al. [24, 25], have extended the agreement protocols to Byzantine link failures. Chor and Coan [2] and Rabin [17] discuss the problem of randomized Byzantine generals. Turek and Shasha [22] present an up-to-date survey of the consensus problem.

The problem of fault-tolerant clock synchronization in distributed systems has been widely studied. Ramanathan and Shin [18] give a comprehensive survey of clock synchronization techniques in distributed systems. Techniques for fault-tolerant clock synchronization appear in papers by Cristian [3], Halpern et al. [12], Ramanathan et al. [19], and Srikanth and Toueg [20]. Consensus-based fault-tolerant distributed systems have been studied by Babaoglu [1]. Application of the Byzantine agreement in distributed transaction commit can be found in papers by Dolev and Strong [6] and Mohan et. al. [15].

PROBLEMS

8.1. Show that Byzantine agreement cannot always be reached among four processors if two processors are faulty.

8.2. Show how a solution to the consensus problem can be used to solve the interactive consistency problem.

8.3. Prove that in Dolev et al.'s algorithm for case $n > 3m + 1$, if the active processors agree on the value 1, then the passive processors will also agree on the value of 1.

REFERENCES

1. Babaoglu, O., "On the Reliability of Consensus-Based Fault-Tolerant Distributed Computing Systems," *ACM Transactions on Computer Systems*, Nov. 1987.
2. Chor, B., and B. Coan, "A Simple and Efficient Randomized Byzantine Agreement Algorithm," *IEEE Transactions on Software Engineering*, June 1985.
3. Cristian, F., "A Probabilistic Approach to Distributed Clock Synchronization," *Proc. of the 9th International Conf. on Distributed Computing Systems*, June 1989.
4. Dolev, D., "The Byzantine Generals Strike Again," *Journal of Algorithms*, Jan. 1982.

5. Dolev, D., M. Fischer, Rob Fowler, Nancy Lynch, and Ray Strong, "An Efficient Algorithm for Byzantine Agreement without Authentication," *Information and Control*, 1982.
6. Dolev, D., and R. Strong, "Distributed Commit with Bounded Waiting," *Proc. of the 2nd IEEE Symposium on Reliability in Distributed Software and Database Systems*, July 1982.
7. Fischer, M. J., "The Consensus Problems in Unreliable Distributed Systems (a Brief Survey)," *Proc. of the 1983 Intl. FTC-Conference*, Aug. 1983.
8. Fischer, M. J., and N. A. Lynch, "A Lower Bound on Time to Assure Interactive Consistency," *Information Processing Letters*, June 1982, pp. 183–186.
9. Fischer, M. J., N. A. Lynch, and M. Merritt, "Easy Impossibility Proofs for Distributed Consensus Problems," *Distributed Computing*, Jan. 1986.
10. Fischer, M. J., N. A. Lynch, and M. S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *Proc. of 2nd ACM Symposium on Principles of Database Systems*, Mar. 1983.
11. Gray, J. N., "Notes on Database Operating Systems," *Operating Systems: An Advanced Course*, Springer-Verlag, New York, 1978, pp. 393–481.
12. Halpern, J., B. Simons, and R. Strong, "An Efficient Fault-Tolerant Algorithm for Clock Synchronization," *Proc. of the 3rd ACM Symposium on Principles of Distributed Computing*, 1984.
13. Lamport,, and P. M. Melliar-Smith, "Synchronizing Clocks in the Presence of Faults," *Journal of the ACM*, Jan. 1985.
14. Lamport, L., R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems*, July 1982.
15. Mohan, C., R. Strong, and S. Finkelstein, "Method for Distributing Transaction Commit and Recovery Using Byzantine Agreement within Clusters of Processors," *Proc. of the 2nd ACM Symposium on Distributed Computing*, Aug. 1983.
16. Pease, M., R. Shostak, and L. Lamport, "Reaching Agreement in the Presence of Faults," *Journal of the ACM*, Apr. 1980.
17. Rabin, M., "Randomized Byzantine Generals," *Proc. of the 24th Symposium on Foundations of Computer Science*, 1983.
18. Ramanathan, P., D. Kandlur, and K. G. Shin, "Hardware-Assisted Software Clock Synchronization for Homogeneous Distributed Systems," *IEEE Transactions on Computers*, Apr. 1990.
19. Ramanathan, P., and K. G. Shin, "Fault-Tolerant Clock Synchronization in Distributed Systems," *IEEE Computer*, Oct. 1990.
20. Srikant, T. K., and S. Toueg, "Optimal Clock Synchronization," *Journal of the ACM*, Jan. 1987.
21. Strong, R., and D. Dolev, "Byzantine Agreement," *Proc. of the Spring Compcon '83*, Mar. 1983.
22. Turek, J., and D. Shasha, "The Many Faces of Consensus in Distributed Systems," *IEEE Computer*, June 1992.
23. Turpin, R., and B. Coan, "Extending Binary Byzantine Agreement to Multivalued Byzantine Agreement," *Information Processing Letters*, Feb. 1984.
24. Yan, K. Q, and Y. H. Chin, "An Optimal Solution for Consensus Problem in an Unreliable Communication System," *Proc. of the Intl. Conf. on Parallel Processing*, Aug. 1988.
25. Yan, K. Q, Y. H. Chin, and S. C. Wang, "Optimal Agreement Protocol in Malicious Faulty Processors and Faulty Links," *to appear in IEEE Trans. on Data and Knowledge Engineering.*