CHAPTER

14 Consensus and agreement algorithms

14.1 Problem definition

Agreement among the processes in a distributed system is a fundamental requirement for a wide range of applications. Many forms of coordination require the processes to exchange information to negotiate with one another and eventually reach a common understanding or agreement, before taking application-specific actions. A classical example is that of the *commit* decision in database systems, wherein the processes collectively decide whether to *commit* or *abort* a transaction that they participate in. In this chapter, we study the feasibility of designing algorithms to reach agreement under various system models and failure models, and, where possible, examine some representative algorithms to reach agreement.

We first state some assumptions underlying our study of agreement algorithms:

- **Failure models** Among the n processes in the system, at most f processes can be faulty. A faulty process can behave in any manner allowed by the failure model assumed. The various failure models – fail-stop, send omission and receive omission, and Byzantine failures – were discussed in Chapter 5. Recall that in the fail-stop model, a process may crash in the middle of a step, which could be the execution of a local operation or processing of a message for a send or receive event. In particular, it may send a message to only a subset of the destination set before crashing. In the Byzantine failure model, a process may behave arbitrarily. The choice of the failure model determines the feasibility and complexity of solving consensus.
- **Synchronous/asynchronous communication** If a failure-prone process chooses to send a message to process P_i but fails, then P_i cannot detect the non-arrival of the message in an asynchronous system because this scenario is indistinguishable from the scenario in which the message takes a very long time in transit. We will see this argument again when we consider

the impossibility of reaching agreement in asynchronous systems in any failure model. In a synchronous system, however, the scenario in which a message has not been sent can be recognized by the intended recipient, at the end of the round. The intended recipient can deal with the non-arrival of the expected message by assuming the arrival of a message containing some default data, and then proceeding with the next round of the algorithm.

- **Network connectivity** The system has full logical connectivity, i.e., each process can communicate with any other by direct message passing.
- **Sender identification** A process that receives a message always knows the identity of the sender process. This assumption is important – because even with Byzantine behavior, even though the payload of the message can contain fictitious data sent by a malicious sender, the underlying network layer protocols can reveal the true identity of the sender process.

  When multiple messages are expected from the same sender in a single round, we implicitly assume a scheduling algorithm that sends these messages in sub-rounds, so that each message sent within the round can be uniquely identified.
- **Channel reliability** The channels are reliable, and only the processes may fail (under one of various failure models). This is a simplifying assumption in our study. As we will see even with this simplifying assumption, the agreement problem is either unsolvable, or solvable in a complex manner.
- **Authenticated vs. non-authenticated messages** In our study, we will be dealing only with *unauthenticated* messages. With unauthenticated messages, when a faulty process relays a message to other processes, (i) it can forge the message and claim that it was received from another process, and (ii) it can also tamper with the contents of a received message before relaying it. When a process receives a message, it has no way to verify its authenticity. An unauthenticated message is also called an *oral* message or an *unsigned* message.

  Using authentication via techniques such as digital signatures, it is easier to solve the agreement problem because, if some process forges a message or tampers with the contents of a received message before relaying it, the recipient can detect the forgery or tampering. Thus, faulty processes can inflict less damage.
- **Agreement variable** The agreement variable may be boolean or multi-valued, and need not be an integer. When studying some of the more complex algorithms, we will use a boolean variable. This simplifying assumption does not affect the results for other data types, but helps in the abstraction while presenting the algorithms.

Consider the difficulty of reaching agreement using the following example, that is inspired by the long wars fought by the Byzantine Empire in the Middle

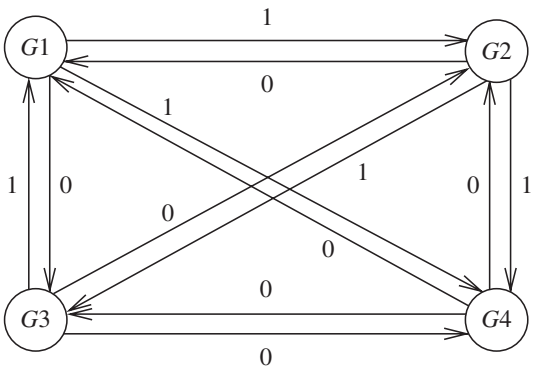

Figure 14.1 Byzantine generals sending confusing messages.

Ages. Four camps of the attacking army, each commanded by a general, are camped around the fort of Byzantium.[1] They can succeed in attacking only if they attack simultaneously. Hence, they need to reach agreement on the time of attack. The only way they can communicate is to send messengers among themselves. The messengers model the messages. An asynchronous system is modeled by messengers taking an unbounded time to travel between two camps. A lost message is modeled by a messenger being captured by the enemy. A Byzantine process is modeled by a general being a traitor. The traitor will attempt to subvert the agreement-reaching mechanism, by giving misleading information to the other generals. For example, a traitor may inform one general to attack at 10 a.m., and inform the other generals to attack at noon. Or he may not send a message at all to some general. Likewise, he may tamper with the messages he gets from other generals, before relaying those messages.

A simple example of Byzantine behavior is shown in Figure 14.1. Four generals are shown, and a *consensus* decision is to be reached about a boolean value. The various generals are conveying potentially misleading values of the decision variable to the other generals, which results in confusion. In the face of such Byzantine behavior, the challenge is to determine whether it is possible to reach agreement, and if so under what conditions. If agreement is reachable, then protocols to reach it need to be devised.

14.1.1 The Byzantine agreement and other problems

The Byzantine agreement problem

Before studying algorithms to solve the agreement problem, we first define the problem formally [20, 25]. The *Byzantine agreement* problem requires a designated process, called the *source process*, with an *initial value*, to reach

[1] Byzantium was the name of present-day Istanbul; Byzantium also had the name of Constantinople.

agreement with the other processes about its initial value, subject to the following conditions:

- **Agreement** All non-faulty processes must agree on the same value.
- **Validity** If the source process is non-faulty, then the agreed upon value by all the non-faulty processes must be the same as the initial value of the source.
- **Termination** Each non-faulty process must eventually decide on a value.

The validity condition rules out trivial solutions, such as one in which the agreed upon value is a constant. It also ensures that the agreed upon value is correlated with the source value. If the source process is faulty, then the correct processes can agree upon any value. It is irrelevant what the faulty processes agree upon – or whether they terminate and agree upon anything at all.

There are two other popular flavors of the Byzantine agreement problem – the *consensus* problem, and the *interactive consistency* problem.

The consensus problem

The consensus problem differs from the Byzantine agreement problem in that each process has an initial value and all the correct processes must agree on a single value [20, 25]. Formally:

- **Agreement** All non-faulty processes must agree on the same (single) value.
- **Validity** If all the non-faulty processes have the same initial value, then the agreed upon value by all the non-faulty processes must be that same value.
- **Termination** Each non-faulty process must eventually decide on a value.

The interactive consistency problem

The interactive consistency problem differs from the Byzantine agreement problem in that each process has an initial value, and all the correct processes must agree upon a set of values, with one value for each process [20, 25]. The formal specification is as follows:

- **Agreement** All non-faulty processes must agree on the same array of values $A[v_1 \ldots v_n]$.
- **Validity** If process i is non-faulty and its initial value is v_i, then all non-faulty processes agree on v_i as the ith element of the array A. If process j is faulty, then the non-faulty processes can agree on any value for $A[j]$.
- **Termination** Each non-faulty process must eventually decide on the array A.

14.1.2 Equivalence of the problems and notation

The three problems defined above are equivalent in the sense that a solution to any one of them can be used as a solution to the other two problems [9]. This equivalence can be shown using a reduction of each problem to the other two problems. If problem A is reduced to problem B, then a solution to problem B can be used as a solution to problem A in conjunction with the reduction. Exercise 14.1 asks the reader to show these reductions.

Formally, the difference between the *agreement problem* and the *consensus problem* is that, in the agreement problem, a single process has the initial value, whereas in the consensus problem, all processes have an initial value. However, the two terms are used interchangably in much of the literature and hence we shall also use the terms interchangably.

14.2 Overview of results

Table 14.1 gives an overview of the results and lower bounds on solving the consensus problem under different assumptions.

It is worth understanding the relation between the consensus problem and the problem of attaining common knowledge of the agreement value. For the "no failure" case, consensus is attainable. Further, in a synchronous system, common knowledge of the consensus value is also attainable, whereas in the asynchronous case, concurrent common knowledge of the consensus value is attainable.

Consensus is not solvable in asynchronous systems even if one process can fail by crashing. To circumvent this impossibility result, weaker variants

Table 14.1 Overview of results on agreement. f denotes number of failure-prone processes. n is the total number of processes.

Failure mode	Synchronous system (message-passing and shared memory)	Asynchronous system (message-passing and shared memory)
No failure	Agreement attainable Common knowledge also attainable	Agreement attainable Concurrent common knowledge attainable
Crash failure	Agreement attainable $f < n$ processes $\Omega(f+1)$ rounds	Agreement not attainable
Byzantine failure	Agreement attainable $f \leq \lfloor (n-1)/3 \rfloor$ Byzantine processes $\Omega(f+1)$ rounds	Agreement not attainable

Table 14.2 Some solvable variants of the agreement problem in an asynchronous system. The overhead bounds are for the given algorithms, and are not necessarily tight bounds for the problem.

Solvable variants	Failure model and overhead	Definition
Reliable broadcast	Crash failures, $n > f$ (MP)	Validity, agreement, integrity conditions (Section 14.5.7)
k-set consensus	Crash failures, $f < k < n$ (MP and SM)	Size of the set of values agreed upon must be at most k (Section 14.5.4)
ϵ-agreement	Crash failures, $n \geq 5f + 1$ (MP)	Values agreed upon are within ϵ of each other (Section 14.5.5)
Renaming	Up to f fail-stop processes, $n \geq 2f + 1$ (MP) Crash failures, $f \leq n - 1$ (SM)	Select a unique name from a set of names (Section 14.5.6)

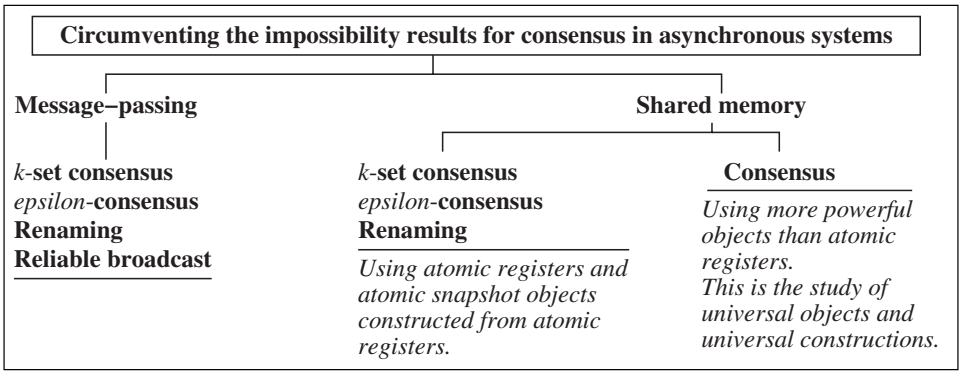

Figure 14.2 Circumventing the impossibility result for consensus in asynchronous systems.

of the consensus problem are defined in Table 14.2. The overheads given in this table are for the algorithms described. Figure 14.2 shows further how asynchronous message-passing systems and shared memory systems deal with trying to solve consensus.

14.3 Agreement in a failure-free system (synchronous or asynchronous)

In a failure-free system, consensus can be reached by collecting information from the different processes, arriving at a "decision," and distributing this decision in the system. A distributed mechanism would have each process broadcast its values to others, and each process computes the same function on the values received. The decision can be reached by using an application-specific function – some simple examples being the *majority*, *max*, and *min* functions. Algorithms to collect the initial values and then distribute the decision may be based on the token circulation on a logical ring, or the three-phase

tree-based broadcast–convergecast–broadcast, or direct communication with all nodes.

- In a synchronous system, this can be done simply in a constant number of rounds (depending on the specific logical topology and algorithm used). Further, common knowledge of the decision value can be obtained using an additional round (see Chapter 8).
- In an asynchronous system, consensus can similarly be reached in a constant number of message hops. Further, concurrent common knowledge of the consensus value can also be attained, using any of the algorithms in Chapter 8.

Reaching agreement is straightforward in a failure-free system. Hence, we focus on failure-prone systems.

14.4 Agreement in (message-passing) synchronous systems with failures

14.4.1 Consensus algorithm for crash failures (synchronous system)

Algorithm 14.1 gives a consensus algorithm for n processes, where up to f processes, where $f < n$, may fail in the fail-stop model [8]. Here, the consensus variable x is integer-valued. Each process has an initial value x_i. If up to f failures are to be tolerated, then the algorithm has $f+1$ rounds. In each round, a process i sends the value of its variable x_i to all other processes if that value has not been sent before. Of all the values received within the round and its own value x_i at the start of the round, the process takes the minimum, and updates x_i. After $f+1$ rounds, the local value x_i is guaranteed to be the consensus value.

(global constants)
integer: f; // maximum number of crash failures tolerated
(local variables)
integer: $x \longleftarrow$ local value;

(1) Process P_i $(1 \leq i \leq n)$ executes the consensus algorithm for up to f crash failures:
(1a) **for** *round* **from** 1 **to** $f+1$ **do**
(1b) **if** the current value of x has not been broadcast **then**
(1c) **broadcast**(x);
(1d) $y_j \longleftarrow$ value (if any) received from process j in this round;
(1e) $x \longleftarrow min_{\forall j}(x, y_j)$;
(1f) **output** x as the consensus value.

Algorithm 14.1 Consensus with up to f fail-stop processes in a system of n processes, $n > f$ [8]. Code shown is for process P_i, $1 \leq i \leq n$.

- The *agreement condition* is satisfied because in the $f+1$ rounds, there must be at least one round in which no process failed. In this round, say round r, all the processes that have not failed so far succeed in broadcasting their values, and all these processes take the minimum of the values broadcast and received in that round. Thus, the local values at the end of the round are the same, say x_i^r for all non-failed processes. In further rounds, only this value may be sent by each process at most once, and no process i will update its value x_i^r.
- The *validity condition* is satisfied because processes do not send fictitious values in this failure model. (Thus, a process that crashes has sent only correct values until the crash.) For all i, if the initial value is identical, then the only value sent by any process is the value that has been agreed upon as per the *agreement condition*.
- The *termination condition* is seen to be satisfied.

Complexity

There are $f+1$ rounds, where $f < n$. The number of messages is at most $O(n^2)$ in each round, and each message has one integer. Hence the total number of messages is $O((f+1) \cdot n^2)$. The worst-case scenario is as follows. Assume that the minimum value is with a single process initially. In the first round, the process manages to send its value to just one other process before failing. In subsequent rounds, the single process having this minimum value also manages to send that value to just one other process before failing.

Algorithm 14.1 requires $f+1$ rounds, independent of the actual number of processes that fail. An *early-stopping* consensus algorithm terminates sooner; if there are f' actual failures, where $f' < f$, then the early-stopping algorithm terminates in $f'+1$ rounds. Exercise 14.2 asks you to design an early-stopping algorithm for consensus under crash failures, and to prove its correctness.

A lower bound on the number of rounds [8]

At least $f+1$ rounds are required, where $f < n$. The idea behind this lower bound is that in the worst-case scenario, one process may fail in each round; with $f+1$ rounds, there is at least one round in which no process fails. In that guaranteed failure-free round, all messages broadcast can be delivered reliably, and all processes that have not failed can compute the common function of the received values to reach an agreement value.

14.4.2 Consensus algorithms for Byzantine failures (synchronous system)

14.4.3 Upper bound on Byzantine processes

In a system of n processes, the Byzantine agreement problem (as also the other variants of the agreement problem) can be solved in a synchronous

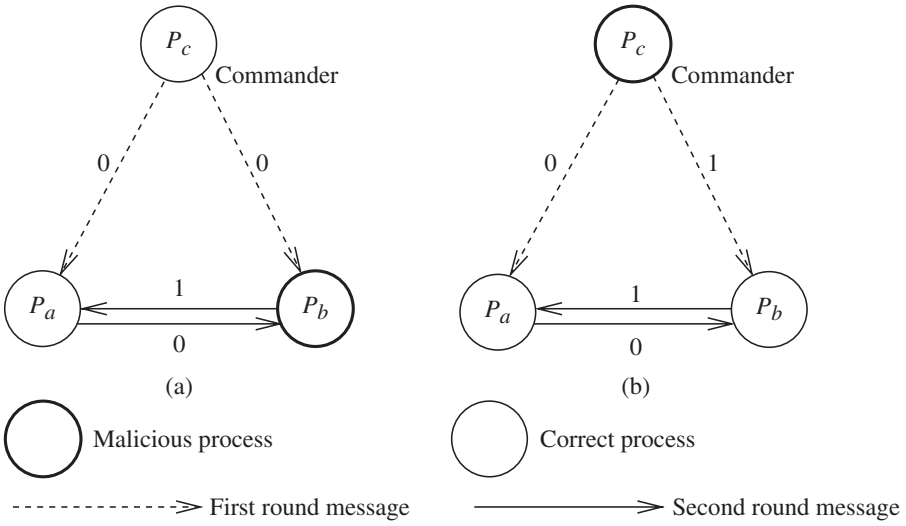

Figure 14.3 Impossibility of achieving Byzantine agreement with $n = 3$ processes and $f = 1$ malicious process.

system only if the number of Byzantine processes f is such that $f \leq \lfloor \frac{n-1}{3} \rfloor$ [20, 25].

We informally justify this result using two steps:

- With $n = 3$ processes, the Byzantine agreement problem cannot be solved if the number of Byzantine processes $f = 1$. The argument uses the illustration in Figure 14.3, which shows a commander P_c and two lieutenant processes P_a and P_b. The malicious process is the lieutenant P_b in the first scenario (Figure 14.3(a)) and hence P_a should agree on the value of the loyal commander P_c, which is 0. But note the second scenario (Figure 14.3(b)) in which P_a receives identical values from P_b and P_c, but now P_c is the disloyal commander whereas P_b is a loyal lieutenant. In this case, P_a needs to agree with P_b. However, P_a cannot distinguish between the two scenarios and any further message exchange does not help because each process has already conveyed what it knows from the third process.

  In both scenarios, P_a gets different values from the other two processes. In the first scenario, it needs to agree on a 0, and if that is the default value, the decision is correct, but then if it is in the second indistinguishable scenario, it agrees on an incorrect value. A similar argument shows that if 1 is the default value, then in the first scenario, P_a makes an incorrect decision. This shows the impossibility of agreement when $n = 3$ and $f = 1$.
- With n processes and $f \geq n/3$ processes, the Byzantine agreement problem cannot be solved. The correctness argument of this result can be shown using reduction. Let $Z(3, 1)$ denote the Byzantine agreement problem for parameters $n = 3$ and $f = 1$. Let $Z(n \leq 3f, f)$ denote the Byzantine agreement problem for parameters $n(\leq 3f)$ and f. A reduction from $Z(3, 1)$ to $Z(n \leq 3f, f)$ needs to be shown, i.e., if $Z(n \leq 3f, f)$ is solvable, then $Z(3, 1)$ is also solvable. After showing this reduction, we can argue that as $Z(3, 1)$ is not solvable, $Z(n \leq 3f, f)$ is also not solvable.

The main idea of the reduction argument is as follows. In $Z(n \leq 3f, f)$, partition the n processes into three sets S_1, S_2, S_3, each of size $\leq n/3$. In $Z(3, 1)$, each of the three processes P_1, P_2, P_3 simulates the actions of the corresponding set S_1, S_2, S_3 in $Z(n \leq 3f, f)$. If one process is faulty in $Z(3, 1)$, then at most f, where $f \leq n/3$, processes are faulty in $Z(n, f)$. In the simulation, a correct process in $Z(3, 1)$ simulates a group of up to $n/3$ correct processes in $Z(n, f)$. It simulates the actions (send events, receive events, intra-set communication, and inter-set communication) of each of the processes in the set that it is simulating.

With this reduction in place, if there exists an algorithm to solve $Z(n \leq 3f, f)$, i.e., to satisfy the validity, agreement, and termination conditions, then there also exists an algorithm to solve $Z(3, 1)$, which has been seen to be unsolvable. Hence, there cannot exist an algorithm to solve $Z(n \leq 3f, f)$.

Byzantine agreement tree algorithm: exponential (synchronous system)

Recursive formulation

We begin with an informal description of how agreement can be achieved with $n = 4$ and $f = 1$ processes [20, 25], as depicted in Figure 14.4. In the first round, the commander P_c sends its value to the other three lieutenants, as shown by dotted arrows. In the second round, each lieutenant relays to the other two lieutenants, the value it received from the commander in the first round. At the end of the second round, a lieutenant takes the majority of the values it received (i) directly from the commander in the first round, and (ii) from the other two lieutenants in the second round. The majority gives a correct estimate of the "commander's" value. Consider Figure 14.4(a) where the commander is a traitor. The values that get transmitted in the two rounds are as

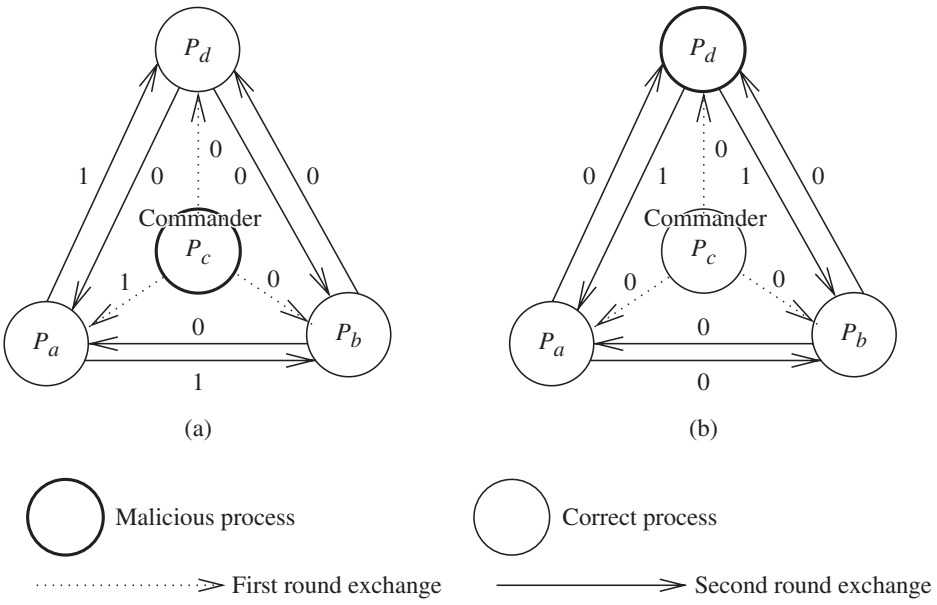

Figure 14.4 Achieving Byzantine agreement when $n = 4$ processes and $f = 1$ malicious process.

shown. All three lieutenants take the majority of $(1, 0, 0)$ which is "0," the agreement value. In Figure 14.4(b), lieutenant P_d is malicious. Despite its behavior as shown, lieutenants P_a and P_b agree on "0," the value of the commander.

(variables)
boolean: $v \longleftarrow$ initial value;
integer: $f \longleftarrow$ maximum number of malicious processes, $\leq \lfloor (n-1)/3 \rfloor$;
(message type)
$OM(v, Dests, List, faulty)$, where
v is a boolean,
$Dests$ is a set of destination process i.d.s to which the message is sent,
$List$ is a list of process i.d.s traversed by this message, ordered from most recent to earliest,
$faulty$ is an integer indicating the number of malicious processes to be tolerated.

$Oral\_Msg$ (f), where $f > 0$:

(1) The algorithm is initiated by the commander, who sends his source value v to all other processes using a $OM(v, N, \langle i \rangle, f)$ message. The commander returns his own value v and terminates.

(2) **[Recursion unfolding:]** For each message of the form $OM(v_j, Dests, List, f')$ received in this round from some process j, the process i uses the value v_j it receives from the source j, and using that value, acts as a *new* source. (If no value is received, a default value is assumed.)

To act as a new source, the process i initiates $Oral\_Msg\,(f'-1)$, wherein it sends
$OM(v_j, Dests - \{i\}, concat(\langle i \rangle, L), (f'-1))$
to destinations not in $concat(\langle i \rangle, L)$
in the next round.

(3) **[Recursion folding:]** For each message of the form $OM(v_j, Dests, List, f')$ received in step 2, each process i has computed the agreement value v_k, for each k not in $List$ and $k \neq i$, corresponding to the value received from P_k after traversing the nodes in $List$, at one level lower in the recursion. If it receives no value in this round, it uses a default value. Process i then uses the value $majority_{k \notin List, k \neq i}(v_j, v_k)$ as the agreement value and returns it to the next higher level in the recursive invocation.

$Oral\_Msg(0)$:

(1) **[Recursion unfolding:]** Process acts as a source and sends its value to each other process.

(2) **[Recursion folding:]** Each process uses the value it receives from the other sources, and uses that value as the agreement value. If no value is received, a default value is assumed.

Algorithm 14.2 Byzantine generals algorithm – exponential number of unsigned messages, $n > 3f$. Recursive formulation.

Table 14.3 Relationships between messages and rounds in the oral messages algorithm for the Byzantine agreement.

Round number	A message has already visited	Aims to tolerate these many failures	Each message gets sent to	Total number of messages in round
1	1	f	$n-1$	$n-1$
2	2	$f-1$	$n-2$	$(n-1)\cdot(n-2)$
...	...	...	...	...
x	x	$(f+1)-x$	$n-x$	$(n-1)(n-2)\ldots(n-x)$
$x+1$	$x+1$	$(f+1)-x-1$	$n-x-1$	$(n-1)(n-2)\ldots(n-x-1)$
...	...	...	...	...
$f+1$	$f+1$	0	$n-f-1$	$(n-1)(n-2)\ldots(n-f-1)$

The first algorithm for solving Byzantine agreement was proposed by Lamport *et al.* [20]. We present two versions of the algorithm.

The recursive version of the algorithm is given in Algorithm 14.2. Each message has the following parameters: a consensus estimate value (v); a set of destinations ($Dests$); a list of nodes traversed by the message, from most recent to least recent ($List$); and the number of Byzantine processes that the algorithm still needs to tolerate ($faulty$). The list $L = \langle P_i, P_{k_1} \ldots P_{k_{f+1-faulty}} \rangle$ represents the sequence of processes (subscripts) in the knowledge expression $K_i(K_{k_1}(K_{k_2} \ldots K_{k_{f+1-faulty}}(v_0)\ldots))$. This knowledge is what $P_{k_{f+1-faulty}}$ conveyed to $P_{k_{f-faulty}}$ conveyed to ... P_{k_1} conveyed to P_i who is conveying to the receiver of this message, the value of the commander ($P_{k_{f+1-faulty}}$)'s initial value.

The commander invokes the algorithm with parameter $faulty$ set to f, the maximum number of malicious processes to be tolerated. The algorithm uses $f+1$ synchronous rounds. Each message (having this parameter $faulty = k$) received by a process invokes several other instances of the algorithm with parameter $faulty = k-1$. The terminating case of the recursion is when the parameter $faulty$ is 0. As the recursion folds, each process progressively computes the majority function over the values it used as a source for that level of invocation in the unfolding, and the values it has just computed as consensus values using the majority function for the lower level of invocations.

There are an exponential number of messages $O(n^f)$ used by this algorithm. Table 14.3 shows the number of messages used in each round of the algorithm, and relates that number to the number of processes already visited by any message as well as the number of destinations of that message.

As multiple messages are received in any one round from each of the other processes, they can be distinguished using the $List$, or by using a scheduling

algorithm within each round. A detailed iterative version of the high-level recursive algorithm is given in Algorithm 14.3. Lines 2a–2e correspond to the unfolding actions of the recursive pseudo-code, and lines 2f–2h correspond to the folding of the recursive pesudo-code. Two operations are defined in the list L: $head(L)$ is the first member of the list L, whereas $tail(L)$

(variables)
boolean: $v \longleftarrow$ initial value;
integer: $f \longleftarrow$ maximum number of malicious processes, $\leq \lfloor (n-1)/3 \rfloor$;
tree of boolean:

- level 0 root is v_{init}^{L}, where $L = \langle\rangle$;
- level h $(f \geq h > 0)$ nodes: for each v_j^L at level $h - 1 = sizeof(L)$, its $n - 2 - sizeof(L)$ descendants at level h are $v_k^{concat(\langle j \rangle, L)}$, $\forall k$ such that $k \neq j, i$ and k is not a member of list L.

(message type)
$OM(v, Dests, List, faulty)$, where the parameters are as in the recursive formulation.

(1) Initiator (i.e., commander) initiates the oral Byzantine agreement:
(1a) **send** $OM(v, N - \{i\}, \langle P_i \rangle, f)$ to $N - \{i\}$;
(1b) **return**(v).

(2) (Non-initiator, i.e., lieutenant) receives the oral message (OM):
(2a) **for** $rnd = 0$ **to** f **do**
(2b) **for** each message OM that arrives in this round, **do**
(2c) **receive** $OM(v, Dests, L = \langle P_{k_1} \ldots P_{k_{f+1-faulty}} \rangle, faulty)$ from P_{k_1};
// $faulty + rnd = f$; $|Dests| + sizeof(L) = n$
(2d) $v_{head(L)}^{tail(L)} \longleftarrow v$; // $sizeof(L) + faulty = f + 1$. fill in estimate.
(2e) **send** $OM(v, Dests - \{i\}, \langle P_i, P_{k_1} \ldots P_{k_{f+1-faulty}} \rangle, faulty - 1)$
to $Dests - \{i\}$ **if** $rnd < f$;
(2f) **for** $level = f - 1$ **down to** 0 **do**
(2g) **for** each of the $1 \cdot (n-2) \cdot \ldots \cdot (n - (level + 1))$ nodes v_x^L in level $level$, **do**
(2h) $v_x^L (x \neq i, x \notin L) = majority_{y \notin concat(\langle x \rangle, L); y \neq i} (v_x^L, v_y^{concat(\langle x \rangle, L)})$;

Algorithm 14.3 Byzantine generals algorithm – exponential number of unsigned messages, $n > 3f$. Iterative formulation. Code for process P_i.

is the list L after removing its first member. Each process maintains a tree of boolean variables. The tree data structure at a non-initiator is used as follows:

- There are $f + 1$ levels from level 0 through level f.
- Level 0 has one root node, $v_{init}^{\langle\rangle}$, after round 1.

- Level h, $0 < h \leq f$ has $1 \cdot (n-2) \cdot (n-3) \cdots (n-h) \cdot (n-(h+1))$ nodes after round $h+1$. Each node at level $(h-1)$ has $(n-(h+1))$ child nodes.
- Node v_k^L denotes the command received from the node $head(L)$ by node k which forwards it to node i. The command was relayed to $head(L)$ by $head(tail(L))$, which received it from $head(tail(tail(L)))$, and so on. The very last element of L is the commander, denoted P_{init}.
- In the $f+1$ rounds of the algorithm (lines 2a–2e of the iterative version), each level k, $0 \leq k \leq f$, of the tree is successively filled to remember the values received at the end of round $k+1$, and with which the process sends the multiple instances of the *OM* message with the fourth parameter as $f-(k+1)$ for round $k+2$ (other than the final terminating round).
- For each message that arrives in a round (lines 2b–2c of the iterative version), a process sets $v_{head(L)}^{tail(L)}$ (line 2d). It then removes itself from *Dests*, prepends itself to L, decrements *faulty*, and forwards the value v to the updated *Dests* (line 2e).
- Once the entire tree is filled from root to leaves, the actions in the folding of the recursion are simulated in lines 2f–2h of the iterative version, proceeding from the leaves up to the root of the tree. These actions are crucial – they entail taking the majority of the values at each level of the tree. The final value of the root is the agreement value, which will be the same at all processes.

Example Figure 14.5 shows the tree at a lieutenant node P_3, for $n = 10$ processes P_0 through P_9 and $f = 3$ processes. The commander is P_0. Only one branch of the tree is shown for simplicity. The reader is urged to work through all the steps to ensure a thorough understanding. Some key steps from P_3's perspective are outlined next, with respect to the iterative formulation of the algorithm.

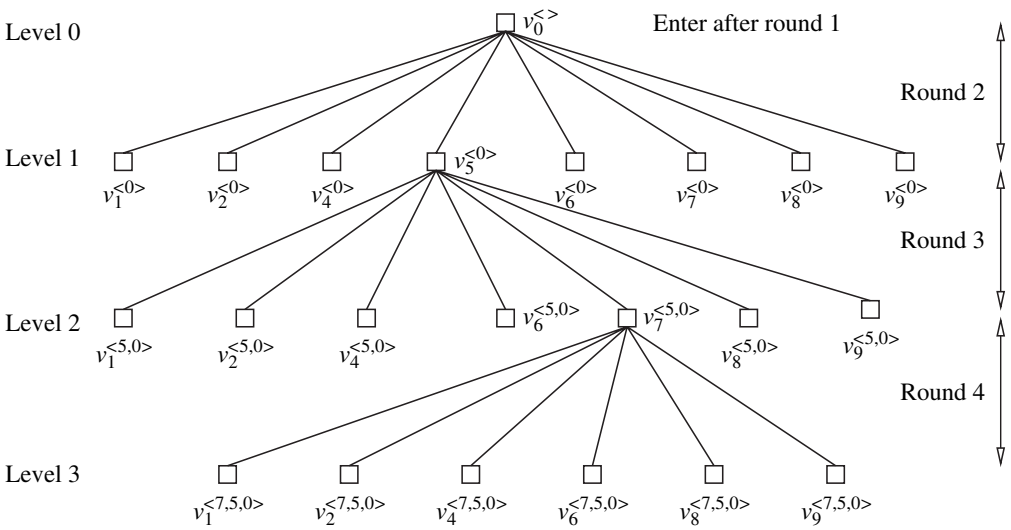

Figure 14.5 Local tree at P_3 for solving the Byzantine agreement, for $n = 10$ and $f = 3$. Only one branch of the tree is shown for simplicity.

- **Round 1** P_0 sends its value to all other nodes. This corresponds to invoking *Oral_Msg* (3) in the recursive formulation. At the end of the round, P_3 stores the received value in $v_0^{\langle\rangle}$.
- **Round 2** P_3 acts as a source for this value and sends this value to all nodes except itself and P_0. This corresponds to invoking *Oral_Msg* (2) in the recursive formulation. Thus, P_3 sends 8 messages. It will receive a similar message from all other nodes except P_0 and itself; the value received from P_k is stored in $v_k^{\langle 0\rangle}$.
- **Round 3** For each of the 8 values received in round 2, P_3 acts as a source and sends the values to all nodes except (i) itself, (ii) nodes visited previously by the corresponding value, as remembered in the superscript list, and (iii) the direct sender of the received message, as indicated by the subscript. This corresponds to invoking *Oral_Msg* (1) in the recursive formulation. Thus, P_3 sends 7 messages for each of these 8 values, giving a total of 56 messages it sends in this round. Likewise it receives 56 messages from other nodes; the values are stored in level 2 of the tree.
- **Round 4** For each of the 56 messages received in round 3, P_3 acts a source and sends the values to all nodes except (i) itself, (ii) nodes visited previously by the corresponding value, as remembered in the superscript list, and (iii) the direct sender of the received message, as indicated by the subscript. This corresponds to invoking *Oral_Msg* (0) in the recursive formulation. Thus, P_3 sends 6 messages for each of these 56 values, giving a total of 336 messages it sends in this round. Likewise, it receives 336 messages, and the values are stored at level 3 of the tree. As this round is *Oral_Msg* (0), the received values are used as estimates for computing the majority function in the folding of the recursion.

An example of the majority computation is as follows:

- P_3 revises its estimate of $v_7^{\langle 5,0\rangle}$ by taking $majority$ $(v_7^{\langle 5,0\rangle}, v_1^{\langle 7,5,0\rangle}, v_2^{\langle 7,5,0\rangle}, v_4^{\langle 7,5,0\rangle}, v_6^{\langle 7,5,0\rangle}, v_8^{\langle 7,5,0\rangle}, v_9^{\langle 7,5,0\rangle})$. Similarly for the other nodes at level 2 of the tree.
- P_3 revises its estimate of $v_5^{\langle 0\rangle}$ by taking $majority$ $(v_5^{\langle 0\rangle}, v_1^{\langle 5,0\rangle}, v_2^{\langle 5,0\rangle}, v_4^{\langle 5,0\rangle}, v_6^{\langle 5,0\rangle}, v_7^{\langle 5,0\rangle}, v_8^{\langle 5,0\rangle}, v_9^{\langle 5,0\rangle})$. Similarly for the other nodes at level 1 of the tree.
- P_3 revises its estimate of $v_0^{\langle\rangle}$ by taking $majority(v_0^{\langle\rangle}, v_1^{\langle 0\rangle}, v_2^{\langle 0\rangle}, v_4^{\langle 0\rangle}, v_5^{\langle 0\rangle}, v_6^{\langle 0\rangle}, v_7^{\langle 0\rangle}, v_8^{\langle 0\rangle}, v_9^{\langle 0\rangle})$. This is the consensus value.

Correctness

The correctness of the Byzantine agreement algorithm (Algorithm 14.3) can be observed from the following two informal inductive arguments. Here we assume that the *Oral_Msg* algorithm is invoked with parameter x, and that there are a total of f malicious processes. There are two cases depending on

whether the commander is malicious. A malicious commander causes more chaos than an honest commander.

Loyal commander

Given f and x, if the commander process is loyal, then $Oral\_Msg(x)$ is correct if there are at least $2f + x$ processes.

This can easily be seen by induction on x:

- For $x = 0$, $Oral\_Msg(0)$ is executed, and the processes simply use the (loyal) commander's value as the consensus value.
- Now assume the above induction hypothesis for any x.
- Then for $Oral\_Msg(x+1)$, there are $2f + x + 1$ processes including the commander. Each loyal process invokes $Oral\_Msg(x)$ to broadcast the (loyal) commander's value v_0 – here it acts as a commander for this invocation it makes. As there are $2f + x$ processes for each such invocation, by the induction hypothesis, there is agreement on this value (at all the honest processes) – this would be at level 1 in the local tree in the folding of the recursion. In the last step, each loyal process takes the majority of the direct order received from the commander (level 0 entry of the tree), and its estimate of the commander's order conveyed to other processes as computed in the level 1 entries of the tree. Among the $2f + x$ values taken in the majority calculation (this includes the commanders's value but not its own), the majority is loyal because $x > 0$. Hence, taking the majority works.

No assumption about commander

Given f, $Oral\_Msg(x)$ is correct if $x \geq f$ and there are a total of $3x + 1$ or more processes.

This case accounts for both possibilities – the commander being malicious or honest. An inductive argument is again useful.

- For $x = 0$, $Oral\_Msg(0)$ is executed, and as there are no malicious processes ($0 \geq f$) the processes simply use the (loyal) commander's value as the consensus value. Hence the algorithm is correct.
- Now assume the above induction hypothesis for any x.
- Then for $Oral\_Msg(x+1)$, there are at least $3x + 4$ processes including the commander and at most $x + 1$ are malicious.
  - (Loyal commander:) If the commander is loyal, then we can apply the argument used for the "loyal commander" case above, because there will be more than $(2(f+1) + (x+1))$ total processes.
  - (Malicious commander:) There are now at most x other malicious processes and $3x + 3$ total processes (excluding the commander). From the induction hypothesis, each loyal process can compute the consensus value using the majority function in the protocol.

Illustration of arguments

In Figure 14.6(a), the commander who invokes $Oral\_Msg(x)$ is loyal, so all the loyal processes have the same estimate. Although the subsystem of $3x$ processes has x malicious processes, all the loyal processes have the same view to begin with. Even if this case repeats for each nested invocation of *Oral_Msg*, even after x rounds, among the processes, the loyal processes are in a simple majority, so the majority function works in having them maintain the same common view of the loyal commander's value. (Of course, had we known the commander was loyal, then we could have terminated after a single round, and neither would we be restricted by the $n > 3x$ bound.) In Figure 14.6(b), the commander who invokes $Oral\_Msg(x)$ may be malicious and can send conflicting values to the loyal processes. The subsystem of $3x$ processes has $x-1$ malicious processes, but all the loyal processes do not have the same view to begin with.

Complexity

The algorithm requires $f+1$ rounds, an exponential amount of local memory, and

$$(n-1)+(n-1)(n-2)+\cdots+[(n-1)(n-2)\cdots(n-f-1)] \text{ messages.}$$

Phase-king algorithm for consensus: polynomial (synchronous system)

The Lamport–Shostak–Pease algorithm [21] requires $f+1$ rounds and can tolerate up to $f \leq \lfloor \frac{n-1}{3} \rfloor$ malicious processes, but requires an exponential number of messages. The *phase-king* algorithm proposed by Berman and Garay [4] solves the consensus problem under the same model, requiring $f+1$ phases, and a polynomial number of messages (which is a huge saving),

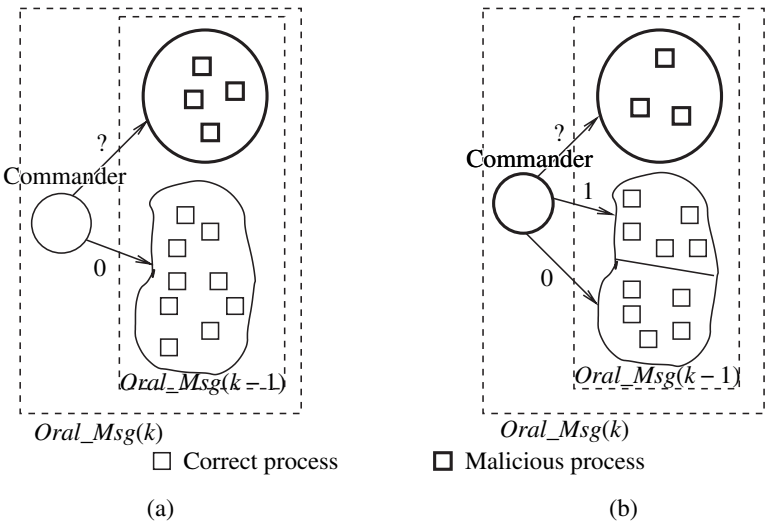

Figure 14.6 The effects of a loyal or a disloyal commander in a system with $n = 14$ and $f = 4$. The subsystems that need to tolerate k and $k-1$ traitors are shown for two cases. (a) Loyal commander. (b) No assumptions about commander.

but can tolerate only $f < \lceil n/4 \rceil$ malicious processes. The algorithm is so called because it operates in $f+1$ phases, each with two rounds, and a unique process plays an asymmetrical role as a leader in each round.

The phase-king algorithm is given in Algorithm 14.4, and assumes a binary decision variable. The message pattern is illustrated in Figure 14.7.

(variables)
boolean: $v \longleftarrow$ initial value;
integer: $f \longleftarrow$ maximum number of malicious processes, $f < \lceil n/4 \rceil$;
(1) Each process executes the following $f+1$ phases, where $f < n/4$:
(1a) **for** $phase = 1$ **to** $f+1$ **do**
(1b) Execute the following round 1 actions:
(1c) **broadcast** v to all processes;
(1d) **await** value v_j from each process P_j;
(1e) $majority \longleftarrow$ the value among the v_j that occurs $> n/2$ times (default value if no majority);
(1f) $mult \longleftarrow$ number of times that $majority$ occurs;
(1g) Execute the following round 2 actions:
(1h) **if** $i = phase$ **then**
(1i) **broadcast** $majority$ to all processes;
(1j) **receive** $tiebreaker$ from P_{phase} (default value if nothing is received);
(1k) **if** $mult > n/2 + f$ **then**
(1l) $v \longleftarrow majority$;
(1m) **else** $v \longleftarrow tiebreaker$;
(1n) **if** $phase = f+1$ **then**
(1o) output decision value v.

Algorithm 14.4 Phase-king algorithm [4] – polynomial number of unsigned messages, $n > 4f$. Code is for process P_i, $1 \leq i \leq n$.

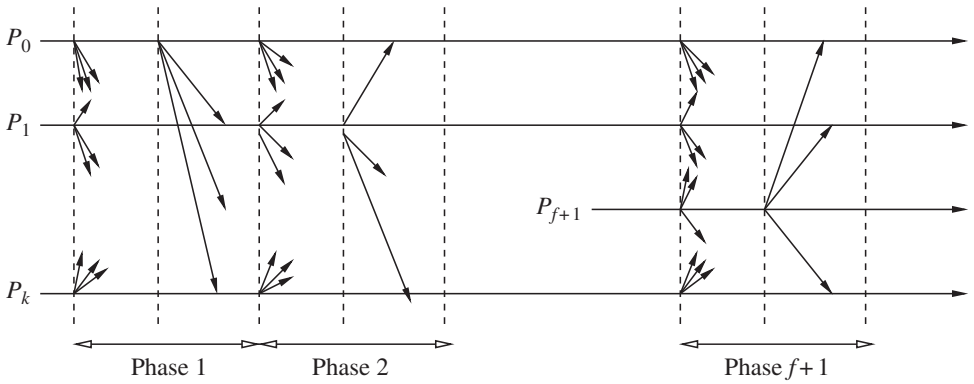

Figure 14.7 Message pattern for the phase-king algorithm.

- **Round 1** In the first round (lines 1b–1f) of each phase, each process broadcasts its estimate of the consensus value to all other processes, and likewise awaits the values broadcast by others. At the end of the round, it counts the number of "1" votes and the number of "0" votes. If either number is greater than $n/2$, then it sets its *majority* variable to that consensus value, and sets *mult* to the number of votes received for the majority value. If neither number is greater than $n/2$, which may happen when the malicious processes do not respond, and the correct processes are split among themselves, then a default value is used for the *majority* variable.
- **Round 2** In the second round (lines 1g–1o) of each phase, the phase king initiates processing – the phase king for phase k is the process with identifier P_k, where $k \in \{1 \ldots n\}$. The phase king broadcasts its majority value *majority*, which serves the role of a tie-breaker vote for those other processes that have a value of *mult* of less than $n/2 + f$. Thus, when a process receives the tie-breaker from the phase king, it updates its estimate of the decision variable v to the value sent by the phase king if its own *mult* variable $< n/2 + f$. The reason for this is that among the votes for its own *majority* value, f votes could be bogus and hence it does not have a clear majority of votes (i.e., $> n/2$) from the non-malicious processes. Hence, it adopts the value of the phase king. However, if $mult > n/2 + f$ (lines 1k–1l), then it has received a clear majority of votes from the non-malicious processes, and hence it updates its estimate of the consensus variable v to its own majority value, irrespective of what tie-breaker value the phase king has sent in the second round.

At the end of $f + 1$ phases, it is guaranteed that the estimate v of all the processes is the correct consensus value.

Correctness

The correctness reasoning is in three steps:

1. Among the $f + 1$ phases, the phase king of some phase k is non-malicious because there are at most f malicious processes.
2. As the phase king of phase k is non-malicious, all non-malicious processes can be seen to have the same estimate value v at the end of phase k. Specifically, observe that any two non-malicious processes P_i and P_j can set their estimate v in three ways:

   (a) Both P_i and P_j use their own *majority* values. Assume P_i's *majority* value is x, which implies that P_i's $mult > n/2 + f$, and of these voters, at least $n/2$ are non-malicious. This implies that P_j must also have received at least $n/2$ votes for x, implying that its majority value *majority* must also be x.

(b) Both P_i and P_j use the phase king's tie-breaker value. As P_k is non-malicious it must have sent the same tie-breaker value to both P_i and P_j.
(c) P_i uses its majority value as the new estimate and P_j uses the phase king's tie-breaker as the new estimate. Assume P_i's $majority$ value is x, which implies that P_i's $mult > n/2 + f$, and of these voters, at least $n/2$ are non-malicious. This implies that phase king P_k must also have received at least $n/2$ votes for x, implying that its majority value $majority$ that it sends as tie-breaker must also be x.

For all three possibilities, any two non-malicious processes P_i and P_j agree on the consensus estimate at the end of phase k, where the phase king P_k is non-malicious.

3. All non-malicious processes have the same consensus estimate x at the start of phase $k+1$ and they continue to have the same estimate at the end of phase $k+1$. This is self-evident because we have that $n > 4f$ and each non-malicious process receives at least $n - f > n/2 + f$ votes for x from the other non-malicious processes in the first round of phase $k+1$. Hence, all the non-malicious processes retain their estimate v of the consensus value as x at the end of phase $k+1$.
The same logic holds for all subsequent phases. Hence, the consensus value is correct.

Complexity
The algorithm requires $f+1$ phases with two sub-rounds in each phase, and $(f+1)[(n-1)(n+1)]$ messages.

14.5 Agreement in asynchronous message-passing systems with failures

14.5.1 Impossibility result for the consensus problem

Fischer *et al.* [12] showed a fundamental result on the impossibility of reaching agreement in an asynchronous (message-passing) system, even if a single process is allowed to have a crash failure. This result, popularly known as the FLP impossibility result, has a significant impact on the field of designing distributed algorithms in a failure-susceptible system. The correctness proof of this result also introduced the important notion of *valency* of global states.

For any global state GS, let $v(GS)$ denote the set of possible values that can be agreed upon in some global state reachable from GS. $|v(GS)|$ is defined as the *valency* of global state GS. For a boolean decision value, a global state can be *bivalent*, i.e., have a valency of two, or *monovalent*, i.e., having a valency of one. A monovalent state GS is *1-valent* if $v(GS) = \{1\}$ and it is *0-valent* if $v(GS) = \{0\}$. Bivalency of a global state captures the idea of uncertainty

in the decision, as either a 0-valent or a 1-valent state may be reachable from this bivalent state.

In an (asynchronous) failure-free system, Section 14.3 showed how to design protocols that can reach consensus. Observe that the consensus value can be solely determined by the inputs. Hence, the initial state is monovalent.

In the face of failures, it can be shown that a consensus protocol necessarily has a bivalent initial state (assuming each process can have an arbitrary initial value from $\{0, 1\}$, to rule out trivial solutions). This argument is by contradiction. Clearly, the initial state where inputs are all 0 is 0-valent and the initial state where inputs are all 1 is 1-valent. Transforming the input assignments from the all-0 case to the all-1 case, observe that there must exist input assignments $\vec{I}_a$ and $\vec{I}_b$ that are 0-valent and 1-valent, respectively, and that they differ in the input value of only one process, say P_i. If a 1-failure tolerant consensus protocol exists, then:

1. Starting from $\vec{I}_a$, if P_i fails immediately, the other processes must agree on 0 due to the termination condition.
2. Starting from $\vec{I}_b$, if P_i fails immediately, the other processes must agree on 1 due to the termination condition.

However, execution 2 looks identical to execution 1, to all processes, and must end with a consensus value of 0, a contradiction. Hence, there must exist at least one bivalent initial state.

Observe that reaching consensus requires some form of exchange of the intial values (either by message-passing or shared memory, depending on the model). Hence, a running process cannot make a unilateral decision on the consensus value. The key idea of the impossibility result is that, in the face of a potential process crash, it is not possible to distinguish between a crashed process and a process or link that is extremely slow. Hence, from a bivalent state, it is not possible to transition to a monovalent state. More specifically, the argument runs as follows. For a protocol to transition from a bivalent global state to a monovalent global state, and using the global time interleaved model for reasoning in the proof, there must exist a *critical step* execution that changes the valency by making a decision on the consensus value. There are two possibilities:

- The *critical step* is an event that occurs at a single process. However, other processes cannot tell apart the two scenarios in which this process has crashed, and in which this process is extremely slow. In both scenarios, the other processes can continue to wait forever and hence the processes may not reach a consensus value, remaining in bivalent state.
- The *critical step* occurs at two or more independent (i.e., not send–receive related) events at different processes. However, as independent events at different processes can occur in any permutation, the *critical step* is not well-defined and hence this possibility is not admissible.

Thus, starting from a bivalent state, it is not possible to transition to a monovalent state. This is the key to the impossibility result for reaching consensus in asynchronous systems.

The impossibility result is significant because it implies that all problems to which the agreement problem can be reduced are also not solvable in any asynchronous system in which crash failures may occur. As all real systems are prone to crash failures, this result has practical significance. We can show that all the problems, such as the following, requiring consensus are not solvable in the face of even a single crash failure:

- The leader election problem.
- The computation of a network-side global function using broadcast–convergecast flows.
- Terminating reliable broadcast.
- Atomic broadcast.

The common strategy is to use a reduction mapping from the consensus problem to the problem X under consideration. We need to show that, by using an algorithm to solve X, we can solve consensus. But as consensus is unsolvable, so must be problem X.

14.5.2 Terminating reliable broadcast

As an example, consider the terminating reliable broadcast problem, which states that a correct process always gets a message even if the sender crashes while sending. If the sender crashes while sending the message, the message may be a null message but it must be delivered to each correct process. The formal specification of reliable broadcast was studied in Chapter 6; here we have an additional termination condition, which states that each correct process must eventually deliver some message.

- **Validity** If the sender of a broadcast message m is non-faulty, then all correct processes eventually deliver m.
- **Agreement** If a correct process delivers a message m, then all correct processes deliver m.
- **Integrity** Each correct process delivers a message at most once. Further, if it delivers a message different from the null message, then the sender must have broadcast m.
- **Termination** Every correct process eventually delivers some message.

The reduction from consensus to terminating reliable broadcast is as follows. A commander process broadcasts its input value using the terminating reliable broadcast. A process decides on a "0" or "1" depending on whether it receives "0" or "1" in the message from this process. However, if it receives the null message, it decides on a default value. As the broadcast is done using the terminating reliable broadcast, it can be seen that the conditions