Chapter 7

More Consensus Problems

The past two chapters have been devoted to consensus problems—Chapter 5 to the coordinated attack problem and Chapter 6 to the agreement problem. In this chapter, we finish our study of synchronous distributed consensus by considering three other consensus problems: the *k-agreement problem*, the *approximate agreement problem*, and the *distributed database commit problem*. As in Chapter 6, we consider process failures only.

7.1 k-Agreement

The first problem we consider is the k-agreement problem, where k is some nonnegative integer. The k-agreement problem is a natural generalization of the ordinary agreement problem considered in Chapter 6. But now, instead of requiring that all processes decide on exactly the same value, we insist only that they limit their decisions to a small number, k, of distinct values.

The original motivation for this problem was purely mathematical—it is interesting to try to determine how the results of Chapter 6 change when the problem requirements are varied in this simple way. But it is possible to imagine practical situations in which such an algorithm could be useful. For example, consider the problem of allocating shareable resources, such as broadcast frequencies in a communication network. It might be desirable for a number of processes to agree on a small number of frequencies to use for the broadcast of a large amount of data (say, a videotape). Because the communication is by broadcast, any number of processes could receive the data using the same frequency. In order to minimize the total communication load, it is preferable to keep the number k of frequencies that are used small.

In this section, we prove exactly matching upper and lower bounds on the

number of rounds required to solve the k-agreement problem, in a complete network graph and for the case of stopping failures only. These bounds are given in terms of n, the number of processes; f, the number of failures tolerated; and k, the allowed number of decision values.

7.1.1 The Problem

In the k-agreement problem, just as for the ordinary agreement problem, we assume that the network is an n-node connected undirected graph with processes $1, \ldots, n$, where each process knows the entire graph. Each process starts with an input from a fixed set V and is supposed to eventually output a decision from the set V. (Again, we assume that for each process, there is exactly one start state containing each input value.) We assume that at most f processes might fail. We consider *stopping failures* only. The required conditions are as follows.

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

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

Termination: All nonfaulty processes eventually decide.

The agreement condition is the natural generalization of the agreement condition for the ordinary agreement problem. Notice that we use the stronger validity condition for stopping failures given near the end of Section 6.1 rather than the weaker one we used in most of Chapter 6; we need this stronger condition for the lower bound proof in Section 7.1.3. The ordinary agreement problem with the stronger validity condition is exactly the k-agreement problem for $k = 1$.

For the results we present in this section, we consider the special case of a complete network graph only. We also assume that V comes equipped with a total ordering.

As in Section 6.2.1, we define a process to be *active* after r rounds, $0 \leq r$, if it does not fail by the end of r rounds.

7.1.2 An Algorithm

We present a very simple algorithm, called *FloodMin*; in fact, it is exactly the algorithm sketched in Exercise 6.9, but it runs for a smaller number of rounds. As we claimed in Exercise 6.9, when this algorithm runs for $f + 1$ rounds, it guarantees ordinary stopping agreement. It turns out that it still guarantees k-agreement when it runs for only $\lfloor \frac{f}{k} \rfloor + 1$ rounds. Thus, roughly speaking, allowing k decision values rather than just one divides the running time by k.

***FloodMin* algorithm (informal):**

Each process maintains a variable *min-val*, originally set to its own initial value. For each of $\lfloor \frac{f}{k} \rfloor + 1$ rounds, the processes all broadcast their *min-val*s, then each process resets its *min-val* to the minimum of its old *min-val* and all the values in its incoming messages. At the end, the decision value is *min-val*.

The code follows. (Compare its structure with that of *FloodSet* in Section 6.2.1.)

***FloodMin* algorithm (formal):**

The message alphabet is V.

***states*$_i$:**
rounds $\in \mathbb{N}$, initially 0
decision $\in V \cup \{unknown\}$, initially *unknown*
min-val $\in V$, initially i's initial value

***msgs*$_i$:**
if *rounds* $\leq \lfloor \frac{f}{k} \rfloor$ then send *min-val* to all other processes

***trans*$_i$:**
rounds := *rounds* + 1
let m_j be the message from j, for each j from which a message arrives
min-val := $\min(\{min\text{-}val\} \cup \{m_j : j \neq i\})$
if *rounds* $= \lfloor \frac{f}{k} \rfloor + 1$ then *decision* := *min-val*

We argue correctness; the proof is similar to that for the *FloodSet* algorithm in Section 6.2.1. Let $M(r)$ denote the set of *min-val* values of active processes after r rounds. We first observe that the set $M(r)$ can only decrease at successive times.

Lemma 7.1 *$M(r) \subseteq M(r-1)$ for all r, $1 \leq r \leq \lfloor \frac{f}{k} \rfloor + 1$.*

Proof. Suppose that $m \in M(r)$. Then m is the value of *min-val*$_i$ after r rounds, for some process i that is active after r rounds. Then either $m =$ *min-val*$_i$ just before round r or else m arrives at i in some round r message, say from j. But in this case, *min-val*$_j = m$ after $r-1$ rounds, and j must be active after $r-1$ rounds because it sends a message at round r. It follows that $m \in M(r-1)$. □

Lemma 7.2 *Let $d \in \mathbb{N}^+$. If at most $d-1$ processes fail during a particular round r, $1 \leq r \leq \lfloor \frac{f}{k} \rfloor + 1$, then $|M(r)| \leq d$, that is, there are at most d different min-vals for active processes after round r.*

Proof. Suppose for the sake of contradiction that at most $d-1$ processes fail during round r, yet $|M(r)| > d$. Let m be the maximum element of $M(r)$ and let $m' \neq m$ be any other element of $M(r)$. Then m' is an element of $M(r-1)$, by Lemma 7.1; let i be any process that is active and has $m' = \mathit{min\text{-}val}_i$ after $r-1$ rounds. If i does not fail in round r, then every process receives a message containing m' from i at round r. But this cannot occur, because some active process has $m > m'$ as its *min-val* after r rounds. It follows that i fails during round r.

But m' was chosen to be any arbitrary element of $M(r)$ other than the maximum, m. Thus, for every element $m' \neq m$ of $M(r)$, there is some process that is active, has its *min-val* equal to m' after $r-1$ rounds, and fails during round r. By assumption, there are at most $d-1$ processes that fail at round r, so there can be at most $d-1$ elements of $M(r)$ other than m. Therefore, $|M(r)| \leq d$; this is a contradiction. □

Now we can prove the main correctness theorem.

Theorem 7.3 *The FloodMin algorithm solves the k-agreement problem for the stopping failure model.*

Proof. Termination and validity are straightforward. We prove the new agreement condition. Suppose, to obtain a contradiction, that the number of distinct decision values is greater than k in a particular execution having at most f failures. Then the number of *min-vals* for active processes after $\lfloor \frac{f}{k} \rfloor + 1$ rounds is at least $k+1$, that is, $|M(\lfloor \frac{f}{k} \rfloor + 1)| \geq k+1$. By Lemma 7.1, $|M(r)| \geq k+1$ for all r, $0 \leq r \leq \lfloor \frac{f}{k} \rfloor + 1$. Then Lemma 7.2 implies that at least k processes fail in each round r, $1 \leq r \leq \lfloor \frac{f}{k} \rfloor + 1$. This yields a total number of failures that is at least $(\lfloor \frac{f}{k} \rfloor + 1)k$. But this is strictly greater than f, which yields a contradiction. □

Complexity analysis. The number of rounds is $\lfloor \frac{f}{k} \rfloor + 1$. The number of messages is at most $(\lfloor \frac{f}{k} \rfloor + 1)n^2$, and the number of message bits is at most $(\lfloor \frac{f}{k} \rfloor + 1)n^2 b$, where b is an upper bound on the number of bits needed to represent a single element of V.

7.1.3 Lower Bound*

In this section, we show that the upper bound of $\lfloor \frac{f}{k} \rfloor + 1$ is tight, by proving that it is also a lower bound, provided that $|V| \geq k+1$. This gives an exact characterization of the speedup that is achievable by allowing k output values

rather than just 1—essentially, the time is divided by k. As you might expect, the ideas of the proof are derived from those used for the proof of the lower bound for ordinary agreement, Theorem 6.33, but they are a good deal more advanced and more interesting. In fact, they take us into the realm of algebraic topology.

For the remainder of this section, fix A to be an n-process algorithm that solves the k-agreement problem, tolerating the stopping failure of at most f processes. Suppose that A halts in $r < \lfloor \frac{f}{k} \rfloor + 1$ rounds; that is, that $r \leq \lfloor \frac{f}{k} \rfloor$. In order to obtain a contradiction, we need the additional assumption that $n \geq f + k + 1$, which means that at least $k + 1$ processes never fail.

Without loss of generality, we may assume that all processes that decide do so exactly at the end of round r and immediately halt. We also assume that every process sends a message to every other process at every round k, $1 \leq k \leq r$ (until and unless it fails). Finally, we assume that the value domain V consists of exactly the $k + 1$ elements, $0, 1, \cdots, k$, since that is all we need to obtain a contradiction.

We obtain a contradiction by showing that in one of the executions of A (with at most f failures), there are $k + 1$ processes that choose $k + 1$ distinct values, thus violating k-agreement.

Overview. Recall the proof of Theorem 6.33, the $f + 1$ round lower bound for ordinary stopping agreement. It uses a *chain argument*, producing a chain of executions that spans from one in which 0 is the only allowable decision to one in which 1 is the only allowable decision. We would like to extend this proof to other values of k. Unfortunately, in the k-agreement problem, unlike in the ordinary agreement problem, the decision values in one execution do not determine the decision values in closely related executions. For example, if executions α and α' of an ordinary agreement algorithm are indistinguishable to nonfaulty process i, then not only must i's decision be the same in both, but also the decisions of all the other nonfaulty processes in both α and α' must be the same as i's decision. In a k-agreement algorithm, if α and α' are indistinguishable to i, then i's decision is still guaranteed to be the same in both, but now the decisions of the other processes are not determined. Even if α and α' are indistinguishable to $n - 1$ processes, the decision value of the remaining process is not determined.

The key idea we use is to construct a k-dimensional collection of executions rather than a (one-dimensional) chain. Adjacent executions in this collection are indistinguishable to designated nonfaulty processes. We call the k-dimensional structure used to organize these executions the *Bermuda Triangle* (because any hypothetical k-agreement algorithm vanishes somewhere in its interior).

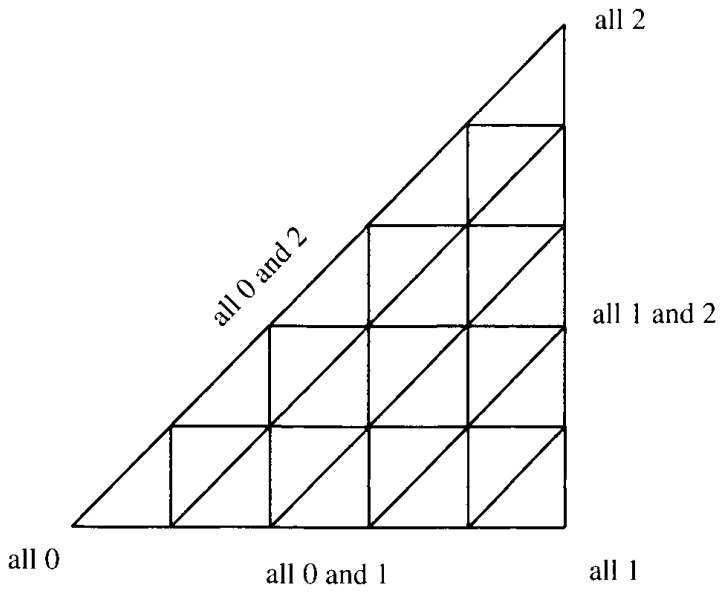

Figure 7.1: Bermuda Triangle for $k = 2$.

Example 7.1.1 Bermuda Triangle

Figure 7.1 is an example of a Bermuda Triangle, for the case where $k = 2$. It consists of a large triangle "triangulated" into a collection of "tiny triangles."

For $k > 2$, we need a k-dimensional version of a triangle. Fortunately, such a generalization already exists in the field of algebraic topology: it is called a k-dimensional *simplex*. For example, a one-dimensional simplex is just an edge, a two-dimensional simplex is a triangle, and a three-dimensional simplex is a tetrahedron. (Beyond three dimensions, the simplices are much harder to imagine.)

So, for an arbitrary k, we start with a k-dimensional simplex in k-dimensional Euclidean space. This simplex contains a number of *grid points*, which are the points in Euclidean space with integer coordinates. The *k-dimensional Bermuda Triangle*, B, is obtained by triangulating this simplex with respect to these grid points, obtaining a collection of tiny k-dimensional simplices.

The proof involves first assigning an execution to each vertex (grid point) of B. The executions in which all processes start with the same input in $\{0, \ldots, k\}$ and there are no failures get assigned to the $k + 1$ corner vertices of B. For instance, in the case where $k = 2$, we assign an execution in which all processes have input 0 to the lower left-hand corner, an execution in which all processes have input 1 to the lower right-hand corner, and an execution in which all processes have input 2 to the upper right-hand corner (see Figure 7.1). Moreover,

for every vertex x on any face of B (of any dimension), the only inputs appearing in the execution assigned to x are those that appear in the executions assigned to the corners of the face. For instance, in the case where $k = 2$, all executions assigned to vertices on the lower edge have inputs chosen from $\{0, 1\}$.

Next, to each vertex in B, we assign the index of some process that is nonfaulty in the execution assigned to that vertex. This process assignment is done in such a way that, for each tiny simplex T, there is a single execution α with at most f faults that is compatible with the executions and processes assigned to the corners of T in the following sense:

1. All the processes assigned to the corners of T are nonfaulty in α.

2. If execution α' and process i are assigned to some corner of T, then α and α' are indistinguishable to i.

This assignment of executions and processes to vertices of B has some nice properties. Suppose α and i are associated with vertex x. If x is a corner of B, then all processes start with the same input in α, so, by the validity condition, i must decide on this value in α. If x is on an external edge of B, then in α each process starts with one of the two input values that are associated with the corners of B at the two ends of the edge; the validity condition then implies that i must decide on one of these two values. More generally, if x is on any face (of any dimension) of B, then in α each process starts with one of the input values that are associated with the corners of the face; the validity condition then implies that i must decide on one of these values. Finally, if x is in the interior of B, then i is allowed to decide on any of the $k + 1$ values.

Our ability to assign executions and indices to the vertices in the manner just described depends on the fact that the number r of rounds in each execution is at most $\lfloor \frac{f}{k} \rfloor$, that is, that $f \geq rk$. This is because the executions are assigned using a k-dimensional generalization of the chain argument in the proof of Theorem 6.33. The construction uses r process failures for each of the k dimensions.

After having assigned executions and indices to vertices, we "color" each vertex with a "color" chosen from the set $\{0, \ldots, k\}$. Namely, we color a vertex x having associated execution α and associated process i with the color that corresponds to i's decision value in α. This coloring has the following properties:

1. The colors of the $k + 1$ corners of B are all different.

2. The color of each point on an external edge of B is the color of one of the corners at the endpoints of the edge.

3. More generally, the color of each point on any external face (of any dimension) of B is the color of one of the corners of the face.

It turns out that colorings of this k-simplex with exactly these properties have been studied in the field of algebraic topology, under the name *Sperner colorings.*

At this point, we can apply a remarkable combinatorial result first proved in 1928: *Sperner's Lemma* says that any Sperner coloring of a triangulated k-dimensional simplex must include at least one tiny simplex whose $k+1$ corners are colored with all $k+1$ distinct colors. In our case, this simplex corresponds to an execution with at most f faults, in which $k+1$ processes choose $k+1$ distinct values. But this contradicts the agreement condition for the k-agreement problem.

It follows that the hypothesized algorithm cannot exist, that is, there is no algorithm for the k-agreement problem tolerating f faults and halting in $r \leq \lfloor \frac{f}{k} \rfloor$ rounds. The rest of this subsection contains more details.

Definitions. We use the definition of a communication pattern from Section 6.7. Now, we redefine a *good* communication pattern to be one in which $k \leq r$ for all triples (i, j, k) and in which the missing triples are consistent with the stopping failure model. (That is, we use the first two conditions in the definition of a good communication pattern in Section 6.7, except that the upper bound for the number of rounds is now r instead of f. For the moment, we do not limit the number of failures.) Based on this new definition of a good communication pattern, we define a *run* and define $exec(\rho)$ for a run ρ in the same way as in Section 6.7. We also say that process i is *silent* after t rounds in a run if i sends no messages in any round numbered $t+1$ or higher.

Bermuda Triangle. We begin with the k-simplex in k-dimensional Euclidean space whose corner vertices are the length k vectors $(0, \ldots, 0)$, $(N, 0, \ldots, 0)$, $(N, N, 0, \ldots, 0), \ldots, (N, \ldots, N)$, where N is a huge integer to be defined shortly. The *Bermuda Triangle* B is this simplex, together with the following triangulation into *tiny simplices.* The vertices of B are the grid points contained in the simplex, that is, the points of the form $x = (x_1, \ldots, x_k)$, where the vector components are integers between 0 and N satisfying $x_1 \geq x_2 \geq \cdots \geq x_k$. The tiny simplices are defined as follows: pick any grid point and walk one step in the positive direction along each dimension, in any order. The $k+1$ points visited by this walk define the vertices of a tiny simplex.

Example 7.1.2 Coordinates of vertices in the Bermuda Triangle

The two-dimensional Bermuda Triangle is illustrated in Figure 7.2.

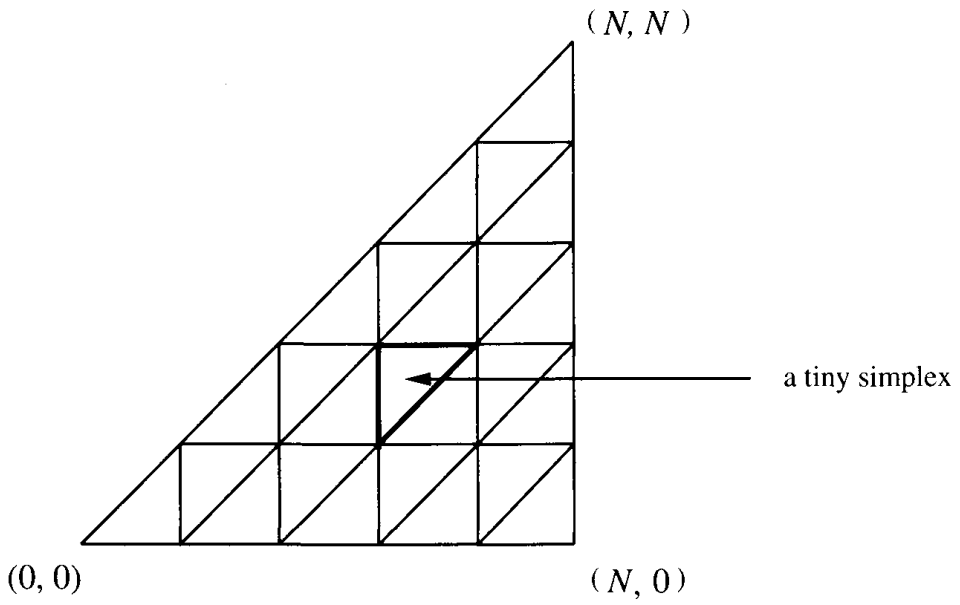

Figure 7.2: Two-dimensional Bermuda Triangle.

Labelling B with executions and runs. In this section, we describe how to assign executions to vertices of B (that is, to "label" the vertices with executions). We do this by first augmenting runs by attaching *tokens* to some of the (process, round number) pairs (i, t) in the runs. Such a token should be thought of as "giving permission" for process i to fail in round t or later. More than one token may be attached to the same pair (i, t).

More specifically, for any $l > 0$, we define an *l-run* to be a run augmented with exactly l tokens for each round number t, $1 \leq t \leq r$, in such a way that if some process i fails at some round t, then there is a token attached to some pair $(i, t'), t' \leq t$. An l-run contains a total of lr tokens. We are really only interested in the two cases where $l = 1$ and $l = k$, that is, 1-runs and k-runs. We define a *failure-free l-run* to be an l-run in which there are no failures and in which all tokens are attached to pairs of the form $(1, t)$ (that is, only process 1 has permission to fail).

Since each augmented run is constructed from a run, each augmented run gives rise to an execution in an obvious way. We extend the notation $exec(\rho)$, which was previously defined for runs, to the case where ρ is an augmented run. In order to label the vertices of B with executions, we label them with k-runs.

We now define four operations on l-runs, each of which makes only minor changes. Each operation can only remove or add a single triple, change the value of a single process's input, or move a single token between processes with

adjacent indices within the same round. These operations are very similar to the ones used in the proof of Theorem 6.33. The operations are defined as follows.

1. *remove*(i, j, t), where i and j are process indices and t is a round number, $1 \leq t \leq r$.
   This operation removes the triple (i, j, t) (which represents the round t message from i to j) if it is there, and has no effect otherwise. It can only be applied if i and j are both silent after t rounds and there is a token attached to some (i, t'), $t' \leq t$.

2. *add*(i, j, t).
   This operation adds the triple (i, j, t) if it is not already there and has no effect otherwise. It can only be applied if i and j are both silent after t rounds and i is active after $t - 1$ rounds.

3. *change*(i, v).
   This operation changes process i's input value to v and has no effect if this input value is already v. It can only be applied if i is silent after 0 rounds and $(i, 1)$ has a token.

4. *move*(i, j, t).
   This operation moves a token from (i, t) to (j, t), where j is either $i + 1$ or $i - 1$. It can only be applied if (i, t) has a token and if all failures have permission from other tokens.

It should be obvious from the definitions that when any of these operations is applied to an l-run, the result is also an l-run.

Now, for any $v \in \{1, \ldots, k\}$, we can define a sequence *seq*(v) of *remove*, *add*, *change*, and *move* operations that can be applied to any failure-free 1-run ρ to transform it into the failure-free 1-run in which all processes have input v. In fact, the same sequence *seq*(v) can be used for all failure-free 1-runs ρ. This can be done using the methods in the proof of Theorem 6.33; the main difference is the explicit movement of the tokens giving permission to fail. In this construction, inputs of processes are changed to v one at a time, starting with process 1. As before, this construction uses r failures in r rounds.

It turns out that the sequences *seq*(v) can be constructed so that they are isomorphic for different v—that is, they are the same except for the choice of v. Now we can (finally) define the parameter N used in defining the size of B: N is simply the length of the sequence *seq*(v) (for any v).

We will use several sequences *seq*(v) to label the vertices of B. Recall that the elements of the value domain are $0, 1, \ldots, k$. For each $v \in \{0, 1, \ldots, k\}$, define τ_v to be the failure-free 1-run in which all processes' initial values are

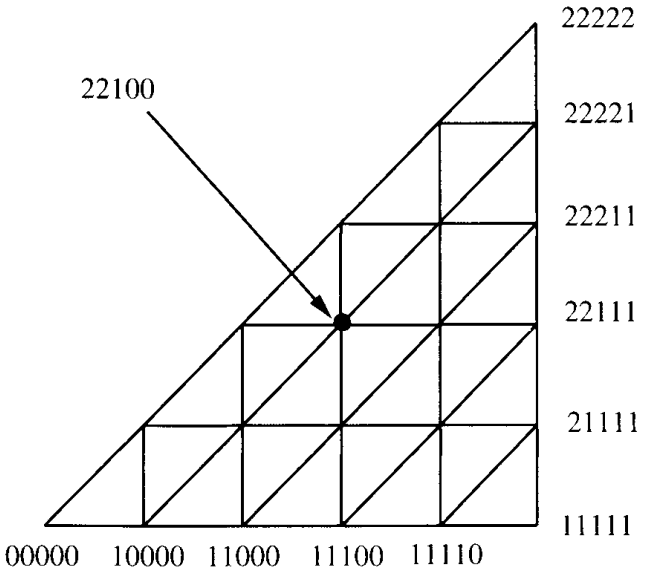

Figure 7.3: Labelling the Bermuda Triangle with k-runs.

equal to v. We will apply each sequence $seq(v)$, $1 \leq v \leq k$, to the failure-free 1-run τ_{v-1}, to generate a sequence of 1-runs to use as *preliminary labels* for the vertices along the edge of B in the vth dimension (the "v-axis"). Then the k-run we assign to each vertex x of B will be obtained by "merging" the k 1-runs that are preliminary labels of the projections of x on the k axes.

Example 7.1.3 Labelling Bermuda Triangle with k-runs

To give some intuition for how this merging works, we give a simplified diagram for the case where $k = 2$ (so $V = \{0, 1, 2\}$) and $n = 5$. See Figure 7.3.

The diagram does not depict all the vertices—only those labelled by failure-free k-runs. Thus, the only interesting information we need to provide is the vector of input values for each depicted vertex. The k-run labelling each corner of B is a failure-free k-run in which all inputs are equal and where the 0s appear at the lower left, the 1s at the lower right, and the 2s at the upper right. The chain along the horizontal axis is constructed by using $seq(1)$ to span from all 0s to all 1s, while the chain along the vertical axis is constructed using $seq(2)$ to span from all 1s to all 2s.

Note the pattern of inputs appearing in B. Along the horizontal axis, the processes' inputs are changed from 0 to 1 one at a time, starting from process 1. Along the vertical axis, the processes' inputs are changed from 1 to 2 one at a time, starting from process 1.

In the interior of B, changes take place in both directions at the same time. For example, consider the indicated interior vertex with input vector 22100. The vectors labelling its projections on the horizontal and vertical axes are 11100 and 22111, respectively. The vector 22100 can be changed into 22111 by changing the inputs of the last two processes from 0 to 1, moving horizontally in B. Similarly, the vector 11100 can be changed into 22100 by changing the inputs of the first two processes from 1 to 2, moving vertically in B. The vector labelling each node of B consists of values in $\{0, 1, 2\}$ occurring in nonincreasing order.

Now we give a formal definition for merging. The *merge* of the sequence $\sigma_1, \ldots, \sigma_k$ of 1-runs is the k-run ρ defined as follows:

1. Process i has the input value v in ρ, where v is the maximum value in $\{1, \ldots, k\}$ such that i has input value v in σ_v, or 0 if no such v exists.

2. A triple (i, j, t) is included in ρ exactly if it is included in all the σ_v, $1 \leq v \leq k$.

3. The number of tokens assigned to a pair (i, t) in ρ is the sum of the number of tokens assigned to the pair (i, t) in all the σ_v.

To motivate the first condition, we reconsider the way that the merge operation is to be used. Each σ_v will be obtained by applying some prefix of $seq(v)$ to τ_{v-1}. At some point in this sequence, the input value for process i is changed from $v-1$ to v. If this has already happened in σ_v then let us say that process i has "converted" in dimension v. The first condition just chooses the largest v (if any) such that process i has converted in dimension v.

The second condition says that a message is missing in the new run ρ if and only if it is missing in any of the runs σ_v being merged. The third condition just accumulates the tokens. It is not hard to see that the merge of a sequence of 1-runs is in fact a k-run.

Now we put the pieces together and define the labelling of vertices of B with k-runs. Let $x = (x_1, \ldots, x_k)$ be an arbitrary vertex of B. For each $v \in \{1, \ldots, k\}$, let σ_v be the 1-run that results from applying the first x_v operations of $seq(v)$ to τ_{v-1}. Then the k-run labelling x is the merge of $\sigma_1, \ldots, \sigma_k$. Note that there are at most $rk \leq f$ tokens in the merged run, and hence at most f failures. For the rest of this proof, we fix the labelling of B with k-runs (and executions).

We end this subsection by giving some close connections among the k-runs labelling the vertices of any single tiny simplex T in B. Let $y_0, \ldots, y_k$ be the vertices of T, in the order determined by the "walk" that generates T (as described

in the definition of the Bermuda Triangle). Let $\rho_0, \ldots, \rho_k$ be the respective k-runs labelling these vertices.

The first lemma says that any process that is faulty in one of these k-runs must have a token in all of them.

Lemma 7.4 *If process i is faulty in some ρ_v, $0 \leq v \leq k$, then i has a token in every ρ_v.*

Proof Sketch. This is because the changes in each sequence *seq* are so gradual, in particular because movement of a token and removal of a triple occur in two separate steps. The detailed proof is left as an exercise. □

The second lemma limits the number of total failures in all of the runs.

Lemma 7.5 *For any $v \in \{0, \ldots, k\}$, let F_v denote the set of processes that fail in ρ_v. Let $F = \cup_v F_v$. Then $|F| \leq rk \leq f$.*

Proof. Left as an exercise. The proof uses Lemma 7.4. □

Finally, we consider labelling vertices of T with process indices. A *local process labelling* of T is an assignment of distinct process indices $i_0, \ldots, i_k$ to the vertices $y_0, \ldots, y_k$ of T in such a way that, for every v, i_v has no tokens in ρ_v. The final important property of the k-runs labelling the vertices of T is that if there is a local process labelling of T, then T is consistent with a single execution.

Lemma 7.6 *Let $i_0, \ldots, i_k$ be a local process labelling of T. Then there is a run ρ with at most f failures such that for all v, i_v is nonfaulty in ρ and $exec(\rho_v)$ and $exec(\rho)$ are indistinguishable to process i_v.*

Proof Sketch. We define ρ as follows. We define the initial value for each process i in ρ to be i's initial value from any one of the ρ_v. For $1 \leq t \leq r-1$, we include the triple (i, j, t) in ρ exactly if it is in all the ρ_v. Likewise, we include (i, j, r), where the recipient j is not one of the processes i_v, exactly if it is in all the ρ_v. Finally, (i, j, r), where $j = i_v$ (for a specific v) is included exactly if it is in ρ_v (for the same v).

We leave it as an exercise to show that ρ has all the needed properties: that it is indeed a run, that it has at most f failures, and that for each v, i_v is nonfaulty in ρ and $exec(\rho_v)$ and $exec(\rho)$ are indistinguishable to process i_v. The proof uses Lemma 7.5 to bound the number of failures. □

Labelling B with process indices. Recall that we are supposed to assign process indices to the vertices of B so that for each tiny simplex T, there is an execution that is compatible with the executions and processes labelling the vertices of T. Lemma 7.6 suggests a way of doing this: for each vertex of B, we pick a process that has no tokens in the corresponding k-run, in such a way that the processes chosen for the vertices of any tiny simplex are all distinct. Lemma 7.6 then implies the needed compatibility condition for each tiny simplex.

We define a *global process labelling* for B to be an assignment of processes to vertices of B such that for every vertex x, the process assigned to x has no tokens in the k-run labelling x, and such that for each tiny simplex T, all the processes assigned to vertices of T are distinct. A global process labelling for B yields a local process labelling for each tiny simplex of B.

We now construct a global process labelling for B. (Since the construction is technical, you might prefer to skip it on a first reading and proceed directly to Lemma 7.10.) We begin the construction by associating a set $live(\rho)$ of processes with each k-run ρ labelling a vertex of B and then choosing one process from each set $live(\rho)$. The sets $live(\rho)$ will satisfy the following properties:

1. Each set $live(\rho)$ consists of exactly $n - rk$ processes. (Since we have assumed that $n \geq f + k + 1$ and $f \geq rk$, this means that each set $live(\rho)$ contains at least $k + 1$ processes.)

2. The processes in $live(\rho)$ are chosen from among those that do not have tokens in ρ.

3. If ρ and ρ' are two k-runs labelling two vertices of the same tiny simplex in B, and if process $i \in live(\rho) \cap live(\rho')$, then i has the same rank in both sets.[1]

So fix some k-run ρ. It contains exactly rk tokens; let *tokens* be the multiset of process indices describing the number of tokens associated with each process. We "flatten" the multiset *tokens* to obtain a new multiset *newtokens* with the same number of tokens, but in which no more than one token is associated with any process. Also, any process that has a token in *tokens* also has a token in *newtokens*. The *Flatten* procedure works as follows:

***Flatten* procedure:**

newtokens := *tokens*
while *newtokens* has a duplicate element do
    select such an element, say i

[1]The *rank* of an element i within a finite totally ordered set L is the number of elements of L that are less than or equal to i.

if there exists $j < i$ such that $newtokens(j) = 0$, then
move a token from i to the largest such j
else move a token from i to the smallest $j > i$ such that $newtokens(j) = 0$

Then we define $live(\rho)$ to be all processes i such that $newtokens(i) = 0$.

It is easy to see that this definition of *live* satisfies the first two properties required above. To see the third property, fix a tiny simplex T, let $y_0, \ldots, y_k$ be the vertices of T, in the order determined by the walk that generates T, and let $\rho_0, \ldots, \rho_k$ be the respective k-runs labelling these vertices. First, we note that, when we walk the vertices of T in order, if process i ever acquires a token, then it always has tokens later in the walk.

Lemma 7.7 *Let $v < v' < v''$. If process i has no tokens in ρ_v but has a token in $\rho_{v'}$, then i has a token in $\rho_{v''}$.*

Proof. Left as an exercise. □

Now we can prove the third property for the *live* sets.

Lemma 7.8 *If $i \in live(\rho_v) \cap live(\rho_w)$, then i has the same rank in $live(\rho_v)$ and $live(\rho_w)$.*

Proof. Assume without loss of generality that $v < w$. Since $i \in live(\rho_v)$ and $i \in live(\rho_w)$, i has no tokens in either ρ_v or ρ_w. Then Lemma 7.7 implies that i has no tokens in any of the runs $\rho_v, \ldots, \rho_w$.

Since token placements in adjacent k-runs differ by at most the movement of one token from one process to an adjacent process, and since i has no tokens in any of these runs, it follows that the total number of tokens on processes smaller than i is the same, say s, in all of the runs $\rho_v, \ldots, \rho_w$. Since $i \in live(\rho_v)$, the way the *Flatten* procedure works implies that $s < i$. (If $s \geq i$, then the tokens that start on processes smaller than i would "overflow" in the *Flatten* procedure so that one would end up on i.) Therefore, i is guaranteed to have the same rank, $i - s$, in $live(\rho_v)$ and $live(\rho_w)$. □

Now we are ready to label the vertices of B with process indices. Let $x = (x_1, \ldots, x_k)$ be any vertex of B, and let ρ be its k-run; we choose a particular process index from the set $live(\rho)$. Namely, let $plane(x) = \sum_{i=1}^{k} x_i \pmod{k+1}$; we label x with the process having rank $plane(x)$ in $live(\rho)$. This choice is motivated by the following fact about B:

Lemma 7.9 *If x and y are distinct vertices of the same tiny simplex, then $plane(x) \neq plane(y)$.*

Now we obtain our goal:

Lemma 7.10 *This labelling of B with process indices is a global process labelling.*

Proof. Because the index for each vertex x is chosen from the set $live(\rho)$, where ρ is the associated k-run, it must be that that index has no tokens in ρ. For any fixed tiny simplex T, Lemmas 7.8 and 7.9 together imply that the chosen indices are all distinct. □

We summarize what we know about the labellings we have produced:

Lemma 7.11 *The given labellings of B with k-runs and processes have the following property. For every tiny simplex T with run labels $\rho_0, \ldots, \rho_k$ and process labels $i_0, \ldots, i_k$, there is a run ρ with at most f failures, such that for all v, i_v is nonfaulty in ρ and $exec(\rho_v)$ and $exec(\rho)$ are indistinguishable to process i_v.*

Proof. This follows from Lemmas 7.6 and 7.10. □

Sperner's Lemma. We are nearly done! It remains only to state Sperner's Lemma (for the special case of the Bermuda Triangle) and to apply it to obtain a contradiction. This will yield the lower bound on the number of rounds required to solve k-agreement. A *Sperner coloring* of B assigns one of a set of $k+1$ colors to each vertex of B so that

1. The colors of the $k+1$ corners of B are all different.
2. The color of each point on an external edge of B is the color of one of the corners at the endpoints of the edge.
3. More generally, the color of each interior point on an external face (of any dimension) of B is the color of one of the adjacent corners of B.

Sperner colorings have a remarkable property: there must be at least one tiny simplex whose $k+1$ vertices are colored with all $k+1$ colors.

Lemma 7.12 (Sperner's Lemma for B) *For any Sperner coloring of B, there is at least one tiny simplex in B whose $k+1$ corners are all colored with distinct colors.*

Now recall that A is the hypothesized k-agreement algorithm, assumed to tolerate f faults and halt in at most $\lfloor \frac{f}{k} \rfloor$ rounds. We define a coloring C_A of B as follows. Given a vertex x labelled with run ρ and process i, color x with process i's decision in the execution $exec(\rho)$ of A.

Lemma 7.13 *If A is an algorithm for k-agreement tolerating f faults and halting in $\lfloor \frac{f}{k} \rfloor$ rounds, then C_A is a Sperner coloring of B.*

Proof. By the validity condition of k-agreement. □

Now we can prove the main theorem:

Theorem 7.14 *Suppose that $n \geq f+k+1$. Then there is no n-process algorithm for k-agreement that tolerates f faults, in which all nonfaulty processes always decide within $\lfloor \frac{f}{k} \rfloor$ rounds.*

Proof. Lemma 7.13 implies that C_A is a Sperner coloring, so Sperner's Lemma, Lemma 7.12, implies that there is a tiny simplex T, all of whose vertices are colored distinctly by C_A.

Suppose that T's k-run labels are $\rho_0, \ldots, \rho_k$ and its process labels are $i_0, \ldots, i_k$. By the definition of C_A, this means that all $k+1$ different decisions are produced by the $k+1$ processes i_v in their respective executions $exec(\rho_v)$. But Lemma 7.11 implies that there is a single run ρ with at most f failures, such that for all v, i_v is nonfaulty in ρ and $exec(\rho_v)$ and $exec(\rho)$ are indistinguishable to process i_v. But this implies that in ρ, the $k+1$ processes $i_0, \ldots, i_k$ decide on $k+1$ distinct values, violating the agreement condition for the k-agreement problem. □

7.2 Approximate Agreement

Now we consider the *approximate agreement* problem in the presence of Byzantine failures. In this problem, the processes start with real-valued inputs and are supposed to eventually decide on real-valued outputs. They are permitted to send real-valued data in messages. Instead of having to agree exactly, as in the ordinary agreement problem, this time the requirement is just that they agree to within a small positive real-valued tolerance ϵ. More precisely, the requirements are

Agreement: The decision values of any pair of nonfaulty processes are within ϵ of each other.

Validity: Any decision value for a nonfaulty process is within the range of the initial values of the nonfaulty processes.

Termination: All nonfaulty processes eventually decide.

This problem arises, for example, in clock synchronization algorithms, where processes attempt to maintain clock values that are close but do not necessarily agree exactly. Many real distributed network algorithms work in the presence of approximately synchronized clocks, so approximate agreement on clock values is usually sufficient.

Here, we consider the approximate agreement problem in complete graphs only. One way of solving the problem is by using an ordinary Byzantine agreement algorithm as a subroutine. This solution assumes that $n > 3f$.

***ByzApproxAgreement* algorithm:**

The processes run an ordinary Byzantine agreement algorithm to decide on a value for each process. All these algorithms run in parallel. In the algorithm for process i, i begins by sending its message to all processes in round 1, then all processes use the received values as their inputs in a Byzantine agreement algorithm. When these algorithms terminate, all nonfaulty processes have the same decision values for all processes. Each chooses the $\lfloor \frac{n}{2} \rfloor$th largest value in the multiset of decision values as its own final decision value.

To see that this works, note that if i is nonfaulty, then the validity condition for Byzantine agreement guarantees that the value obtained by all nonfaulty processes for i is i's actual input value. Since $n > 3f$, it follows that the middle value in the multiset must be in the range of the initial values of the nonfaulty processes.

Theorem 7.15 *ByzApproxAgreement solves the approximate agreement problem for an n-node complete graph, if $n > 3f$.*

Now we present a second solution, not using Byzantine agreement. The main reason we present this solution is that it has an easy extension to the asynchronous network model, which we present in Chapter 21. In contrast, the Byzantine agreement problem cannot be solved in asynchronous networks. The second solution also has the property that it sometimes terminates in fewer than the number of rounds required for Byzantine agreement, depending on how far apart the initial values of nonfaulty processes are. The algorithm is based on successive approximation. For simplicity, we describe a nonterminating version of the algorithm, then discuss termination separately. This algorithm again assumes that $n > 3f$.

We need a little notation and terminology: First, if U is a finite multiset of reals with at least $2f$ elements, and $u_1, \ldots, u_k$ is an ordering of the elements of U in nondecreasing order, then let $reduce(U)$ denote the result of removing

the f smallest and f largest elements from U, that is, the multiset consisting of $u_{f+1}, \ldots, u_{k-f}$. Also, if U is a nonempty finite multiset of reals, and $u_1, \ldots, u_k$ is again an ordering of the elements of U in nondecreasing order, then let $select(U)$ be the multiset consisting of $u_1, u_{f+1}, u_{2f+1}, \ldots$, that is, the smallest element of U and every fth element thereafter. Finally, if U is a nonempty finite multiset of reals, then $mean(U)$ is just the mean of the elements in U.

We also say that the *range* of a nonempty finite multiset of reals is the smallest interval containing all the elements, and the *width* of such a multiset is the size of the range interval.

The second solution is as follows:

***ConvergeApproxAgreement* algorithm:**

Process i maintains a variable *val* containing its latest estimate. Initially, val_i contains i's initial value. At each round, process i does the following.

First, it broadcasts its *val* value to all processes, including itself.[2] Then it collects all the values it has received at that round into a multiset W; if i does not receive a value from some other process, it simply picks some arbitrary default value to assign to that process in the multiset, thus ensuring that $|W| = n$.

Then, process i sets *val* to $mean(select(reduce(W)))$. That is, process i throws out the f smallest and f largest elements of W. From what is left, i selects only the smallest element and every fth element thereafter. Finally, *val* is set to the average (mean) of the selected elements.

We claim that at any round, all the nonfaulty processes' *val*s are in the range of the nonfaulty processes' *val*s just prior to the round. Moreover, at each round, the width of the multiset of nonfaulty processes' *val*s is reduced by a factor of at least $\lfloor \frac{n-2f-1}{f} \rfloor + 1$. If $n > 3f$, this is greater than 1.

Lemma 7.16 *Suppose that $val_i = v$ just after round r of an execution of ConvergeApproxAgreement, where i is a nonfaulty process. Then v is in the range of the nonfaulty processes' vals just before round r.*

Proof. If W_i is the multiset collected by process i at round r, then there are at most f elements of W_i that are not values sent by nonfaulty processes. Then all the elements of $reduce(W_i)$ are in the range of nonfaulty processes' *val*s just prior to round r. It follows that the same is true for $mean(select(reduce(W_i)))$, which is the new value of val_i. □

[2] As usual, sending to itself is simulated by a local transition.

Lemma 7.17 *Suppose that $val_i = v$ and $val_{i'} = v'$ just after round r of an execution of ConvergeApproxAgreement, where i and i' are both nonfaulty processes. Then*

$$|v - v'| \leq \frac{d}{\left\lfloor \frac{n-2f-1}{f} \right\rfloor + 1},$$

where d is the width of the range of the nonfaulty processes' vals just before round r.

Proof. Let W_i and $W_{i'}$ be the respective multisets collected by processes i and i' in round r. Let S_i and $S_{i'}$ be the respective multisets $select(reduce(W_i))$ and $select(reduce(W_{i'}))$. Let $c = \lfloor \frac{n-2f-1}{f} \rfloor + 1$; note that c is exactly the number of elements in S_i and in $S_{i'}$. Let the elements of S_i be denoted by $u_1, \ldots, u_c$ and those of $S_{i'}$ by $u'_1, \ldots, u'_c$, both in nondecreasing order. We begin with a claim that says that the reduced multisets differ in at most f elements.

Claim 7.18 $|reduce(W_i) - reduce(W_{i'})| \leq f$.

Proof. Since nonfaulty processes contribute the same value to both W_i and $W_{i'}$, we have that $|W_i - W_{i'}| \leq f$. We can show that removing a smallest element from both multisets does not increase the number of elements in the difference, and we can show the same for removing a largest element. Using these two facts f times apiece yields the result. □

Claim 7.18 can be used to show

Claim 7.19 $u_j \leq u'_{j+1}$ *and* $u'_j \leq u_{j+1}$ *for all* j, $1 \leq j \leq c-1$.

Proof. We show the first claim only; the second is symmetric. Note that u_j is the $((j-1)f+1)$st smallest element of $reduce(W_i)$, and u'_{j+1} is the $(jf+1)$st smallest element of $reduce(W_{i'})$. Since, by Claim 7.18, there are at most f elements of $reduce(W_{i'})$ that are not elements of $reduce(W_i)$, it must be that $u_j \leq u'_{j+1}$. □

Now we finish the proof of Lemma 7.17 by calculating the required bound. We have that

$$\begin{aligned}
|v - v'| &= |mean(S_i) - mean(S_{i'})| \\
&= \frac{1}{c}|(\Sigma_{j=1}^{c}(u_j - u'_j))| \\
&\leq \frac{1}{c}(\Sigma_{j=1}^{c}|u_j - u'_j|) \\
&= \frac{1}{c}(\Sigma_{j=1}^{c}(\max(u_j, u'_j) - \min(u_j, u'_j))).
\end{aligned}$$

By Claim 7.19, $\max(u_j, u'_j) \leq \min(u_{j+1}, u'_{j+1})$ for all j, $1 \leq j \leq c-1$, so this latter expression is less than or equal to

$$\frac{1}{c}(\Sigma_{j=1}^{c-1}(\min(u_{j+1}, u'_{j+1}) - \min(u_j, u'_j))) + \frac{1}{c}(\max(u_c, u'_c) - \min(u_c, u'_c)),$$

which collapses to

$$\frac{1}{c}(\max(u_c, u'_c) - \min(u_1, u'_1)).$$

But all of the values u_c, u'_c, u_1, and u'_1 are in the range of the nonfaulty processes' *vals* just before round r, since all elements of $reduce(W_i)$ and $reduce(W_{i'})$ are in this range. So this last expression is less than or equal to $\frac{d}{c}$, as needed.

Termination. We convert *ConvergeApproxAgreement* to a terminating algorithm, that is, one in which all processes eventually decide. (In fact, all processes eventually halt.) Namely, each nonfaulty process uses the range of all the values it receives at round 1 to compute a round number by which it is sure that the *vals* of any two nonfaulty processes will be at most ϵ apart. Each process can do this because it knows the value of ϵ and the guaranteed rate of convergence, and furthermore, it knows that the range of values it receives at round 1 includes the initial values of all the nonfaulty processes. Different nonfaulty processes might compute different round numbers, however.

Any process i that reaches its computed round decides on its own current *val*. After doing this, process i broadcasts its *val* with a special *halting* tag and then halts. After any process j receives a *val* with a *halting* tag from i, it uses this *val* as its message from i, not only for the current round, but also for all future rounds (until j itself decides to halt, on the basis of j's own computed round number).

Although nonfaulty processes might compute different round numbers, it should be clear that the smallest such estimate is correct. Thus, at the time the first nonfaulty process halts, the range of *vals* is already sufficiently small. At subsequent rounds, the range of *vals* of nonfaulty processes never increases, although there is no guarantee that it continues to decrease.

Theorem 7.20 *ConvergeApproxAgreement, with termination added as above, solves the approximate agreement problem for an n-node complete graph, if* $n > 3f$.

Complexity analysis. There is no upper bound depending only on n, f, ϵ and the width of the multiset of nonfaulty processes' initial values, for the time for all nonfaulty processes to decide in the *ConvergeApproxAgreement* algorithm.

This is because faulty processes can send arbitrary values at round 1, which can cause the nonfaulty processes to compute arbitrarily large round numbers for termination.

The exercises discuss bounds on the number of processes and the connectivity needed to solve the approximate agreement problem. We will revisit this problem in Chapter 21, in the asynchronous network setting.

7.3 The Commit Problem

In this, the final section on distributed consensus problems in synchronous systems, we present some of the key ideas about the *distributed database commit* problem. As discussed in Section 5.1, the problem arises when a collection of processes participate in the processing of a database transaction. After this processing, each process arrives at an initial "opinion" about whether the transaction ought to be *committed* (i.e., its results made permanent and released for the use of other transactions) or *aborted* (i.e., its results discarded). A process will generally favor committing the transaction if all its local computation on behalf of that transaction has been successfully completed, and otherwise will favor aborting the transaction. The processes are supposed to communicate and eventually agree on one of the outcomes, *commit* or *abort*. If possible, the outcome should be *commit*.

Solutions to this problem have been designed for real distributed networks, in which there can be a combination of process and link failures. However, the results in Chapter 5 imply that there can be no solution in the case of unlimited link failures. Some limitation must therefore be assumed on message loss.

7.3.1 The Problem

We consider a simplified version of the commit problem, for networks in which there is no message loss, but only process failures. If you are interested in implementing the algorithms in this chapter in a real network, you will have to add other mechanisms, such as repeated retransmissions, to cope with lost messages. We allow any number of process stopping failures.

We assume that the input domain is $\{0, 1\}$, where 1 represents *commit* and 0 represents *abort*. We restrict attention here to the case where the network is a complete graph. The correctness conditions are

Agreement: No two processes decide on different values.

Validity:

1. If any process starts with 0, then 0 is the only possible decision value.
2. If all processes start with 1 and there are no failures, then 1 is the only possible decision value.

Termination: This comes in two flavors. The *weak termination condition* says that if there are no failures then all processes eventually decide. The *strong termination condition* (also known as the *non-blocking condition*) says that all nonfaulty processes eventually decide.

Commit algorithms that satisfy the strong termination condition are sometimes called *non-blocking commit algorithms*, while commit algorithms that satisfy the weak termination condition but not the strong one are sometimes called *blocking commit algorithms.*

Notice that our agreement condition is that *no two processes* decide on different values. Thus, we do not allow even a failed process to decide differently from other processes. We require this because, in practical uses of a commit protocol, a process might fail and later recover. Suppose, for example, that a process i decides *commit* before it fails, and that later, other processes decide *abort.* If process i recovers and retains its *commit* decision, then there would be an inconsistency.

The formal problem statement is similar to two others we have already considered: the coordinated attack problem in Section 5.1 and the agreement problem for stopping failures in Section 6.1. The most important difference between the commit problem and the coordinated attack problem is that we are here considering process failure and not link failure; there is also a difference in the validity condition. The important differences between the commit problem and the stopping agreement problem are, first, the particular choice of validity condition, and, second, the consideration of a weaker notion of termination. Results in Section 6.7 about the stopping agreement problem imply a lower bound of $n - 1$ on the number of rounds needed to solve the commit problem with the strong termination condition. (Note that the proof of Theorem 6.33 still works with the commit validity conditions.)

In the rest of this section, we give versions of two standard practical commit algorithms (for the simplified setting with only process faults). The first, *two-phase commit*, is a blocking algorithm, while the second, *three-phase commit*, is non-blocking. We then give a simple lower bound on the number of messages needed to solve the problem, even if only weak termination is required.

7.3.2 Two-Phase Commit

The best-known practical commit algorithm is *two-phase commit*; without any embellishments, this simple algorithm guarantees only weak termination.

> ***TwoPhaseCommit* algorithm:**
>
> The algorithm assumes a distinguished process, say process 1.
>
> *Round 1:* All processes except for process 1 send their initial values to process 1, and any process whose initial value is 0 decides 0. Process 1 collects all these values, plus its own initial value, into a vector. If all positions in this vector are filled in with 1s, then process 1 decides 1. Otherwise—that is, if there is some position in the vector that contains 0 or else some position that is not filled in (because no message was received from the corresponding process)—process 1 decides 0.
>
> *Round 2:* Process 1 broadcasts its decision to all the other processes. Any process other than process 1 that receives a message at round 2 and has not already decided at round 1 decides on the value it receives in that message.

See Figure 7.4 for an illustration of the communication pattern used in the failure-free runs of *TwoPhaseCommit*.[3]

Theorem 7.21 *TwoPhaseCommit solves the commit problem with the weak termination condition.*

Proof. Agreement, validity, and weak termination are all easy to show. □

However, *TwoPhaseCommit* does not satisfy the strong termination condition, that is, it is a blocking algorithm. This is because if process 1 fails before beginning its broadcast in round 2, then no nonfaulty process whose initial value is 1 ever decides. In practice, if process 1 fails, then the remaining processes usually carry out some sort of *termination protocol* among themselves and sometimes manage to decide. For example, if process 1 fails but some other process, i, has already decided 0 in round 1, then process i can inform the remaining nonfaulty processes that its decision is 0, and they can also safely decide 0. But the termination protocol cannot succeed in all cases. For example, suppose that all processes except for 1 start with input 1, but process 1 fails before sending any

[3]Our round designation does not correspond exactly to the usual designation of phases for the two-phase commit protocol. Usually, an extra round is added at the beginning, in which process 1 requests the *commit* or *abort* values from the other processes. Phase 1 then consists of this extra round plus our round 1. We do not need the extra round for our simplified model and problem statement.

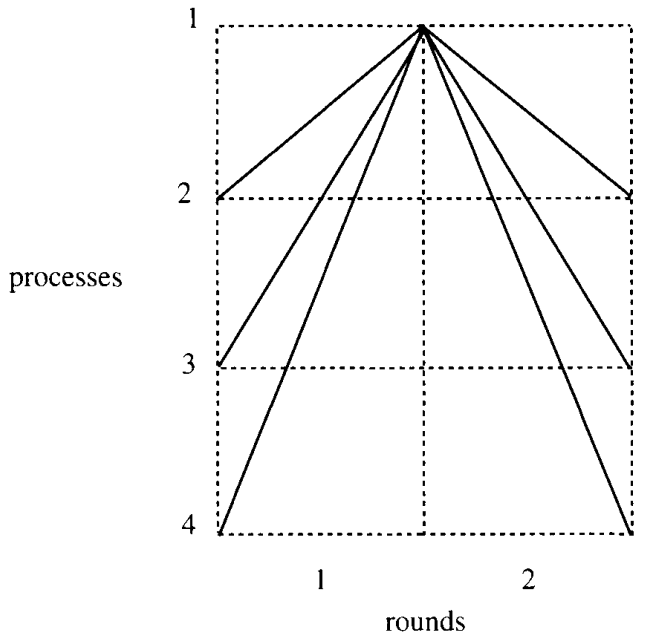

Figure 7.4: Communication pattern in *TwoPhaseCommit*.

messages. Then no other process ever learns process 1's initial value, so, because of the validity condition, no process can decide 1. On the other hand, no process can decide 0, since as far as any other process can tell, it might be that process 1 has already decided 1 just before failing, and the inconsistency would violate the agreement condition.

Complexity analysis. *TwoPhaseCommit* takes only two rounds. Recall that Theorem 6.33 gives a lower bound of $f + 1$ on the number of rounds for stopping agreement. The time bound for *TwoPhaseCommit* does not contradict this lower bound, because *TwoPhaseCommit* satisfies only the weak termination condition. The communication complexity, as measured by the worst-case number of non-*null* messages that are sent in any execution, is $2n - 2$; in particular, this number of messages is sent in a failure-free execution.

7.3.3 Three-Phase Commit

Now we describe the *ThreePhaseCommit algorithm*; this is an embellishment of the *TwoPhaseCommit* algorithm that guarantees strong termination.

The key is simply that process 1 does not decide 1 unless every process that has not yet failed is "ready" to decide 1. Making sure they are ready requires an extra round. We first describe and analyze the first three rounds

of the algorithm. The rest of the algorithm, needed to obtain the non-blocking property, is described afterward.

***ThreePhaseCommit* algorithm, first three rounds:**

Round 1: All processes except for 1 send their initial values to process 1, and any process whose initial value is 0 decides 0. Process 1 collects all these values, plus its own initial value, into a vector. If all positions in this vector are filled in with 1s, then process 1 becomes *ready* but does not yet decide. Otherwise—that is, if there is some position that contains 0 or else some position that is not filled in (because no message was received from the corresponding process)—process 1 decides 0.

Round 2: If process 1 has decided 0, then it broadcasts *decide*(0). If not, then process 1 broadcasts *ready*. Any process that receives *decide*(0) decides 0. Any process that receives *ready* becomes *ready*. Process 1 decides 1 if it has not already decided.

Round 3: If process 1 has decided 1, it broadcasts *decide*(1). Any process that receives *decide*(1) decides 1.

See Figure 7.5 for an illustration of the communication pattern used in the failure-free runs of *ThreePhaseCommit*.[4]

Before presenting the termination protocol, we analyze the situation after the first three rounds. We classify the states of each process (failed or not) into four exclusive and exhaustive categories:

1. *dec0:* Those in which the process has decided 0.
2. *dec1:* Those in which the process has decided 1.
3. *ready:* Those in which the process has not decided, but is *ready*.
4. *uncertain:* Those in which the process has not decided and is not *ready*.

The key properties of *ThreePhaseCommit* are expressed by the following lemma. It describes certain combinations of states that cannot coexist.

Lemma 7.22 *After three rounds of ThreePhaseCommit, the following are true:*

1. *If any process's state is in ready or dec1, then all processes' initial values are* 1.

[4]Again, our round designation does not correspond exactly to the usual designation of phases for the three-phase commit protocol. An extra request round is usually added at the beginning, as well as some explicit acknowledgments.

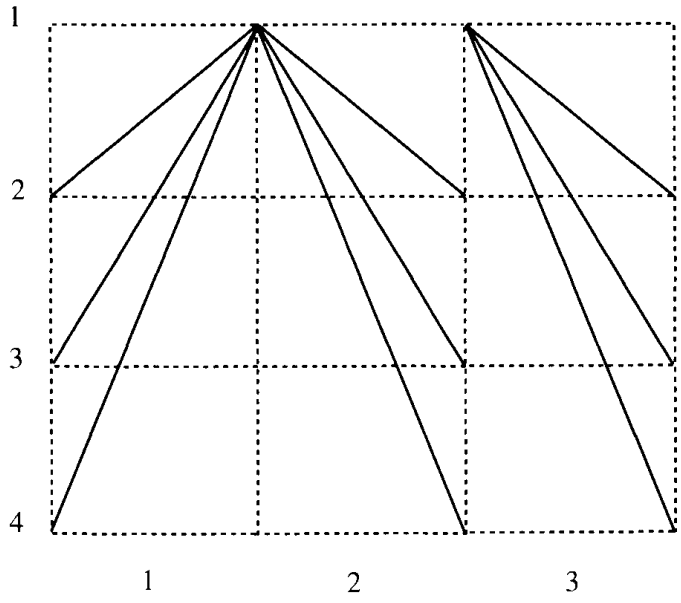

Figure 7.5: Communication pattern in *ThreePhaseCommit*.

2. *If any process's state is in dec0, then no process is in dec1, and no non-failed process is in ready.*

3. *If any process's state is in dec1, then no process is in dec0, and no non-failed process is in uncertain.*

Proof. Straightforward. The most interesting part of the proof is the proof of the third condition. For this, we note that process 1 can only decide 1 at the end of round 2, after it has already broadcast *ready* messages. This means that process 1 knows at the end of round 2 that each other process has either received and processed the *ready*, thereby entering the *ready* state, or else has failed. (The synchrony of the model is important here.) □

Now we can prove that most of the conditions of interest hold after the first three rounds.

Lemma 7.23 *After three rounds of ThreePhaseCommit, the following are true:*

1. *The agreement condition holds.*

2. *The validity condition holds.*

3. *If process* 1 *has not failed, then all non-failed processes have decided.*

Proof. The agreement condition follows from Lemma 7.22, as does half of the validity condition—the half that says that if some process starts with 0, then 0 is the only possible decision value. The other half of the validity condition can be proved by inspection.

Finally, if process 1 has not failed, then we claim that every non-failed process has decided. This is because process 1 cannot be prevented from deciding by any actions of the other processes, and once 1 decides, it immediately broadcasts its decision to the other processes, who decide in the same way. □

These three rounds alone are not enough to solve the non-blocking commit problem, however, because they do not guarantee strong termination. If process 1 does not fail, then every nonfaulty process decides, as noted in Lemma 7.23. But if process 1 fails, it is possible that the other processes might be left in an undecided state. To take care of this case, the remaining processes must execute a *termination protocol* after the first three rounds. The precise details can vary somewhat; we describe one possibility below.

ThreePhaseCommit, termination protocol:

Round 4: All (not yet failed) processes send their current status, either *dec0*, *dec1*, *ready*, or *uncertain*, to process 2. Process 2 collects all these status values, plus its own status, into a vector. Not all the positions in the vector need be filled in—process 2 just ignores those that are not. If the vector contains any *dec0* values and process 2 has not already decided, then process 2 decides 0. If the vector contains any *dec1* values and process 2 has not already decided, then process 2 decides 1. If all the filled-in positions in the vector contain the value *uncertain*, then process 2 decides 0. Otherwise—that is, if the only values in the vector are *uncertain* and *ready* and there is at least one *ready*—process 2 becomes *ready* but does not yet decide.

Round 5: In this and the next round, process 2 behaves similarly to process 1 in rounds 2 and 3. If process 2 has (ever) decided, then it broadcasts its decision, in a *decide* message. If not, then process 2 broadcasts *ready*. Any process that receives *decide*(0) or *decide*(1) and has not already decided, decides 0 or 1, as indicated. Any process that receives *ready* becomes *ready*. Process 2 decides 1 if it has not already decided.

Round 6: If process 2 has decided 1, it broadcasts *decide*(1). Any process that receives *decide*(1), and has not already decided, decides 1.

After round 6, the protocol then continues with three similar rounds coordinated by each of processes $3, \ldots, n$.

Theorem 7.24 *The complete ThreePhaseCommit algorithm, including the termination protocol, is a non-blocking commit algorithm.*

Proof Sketch. We first claim that the three properties listed in the statement of Lemma 7.22 hold after *any number* of rounds of the full *ThreePhaseCommit* algorithm, not just after three rounds as claimed. This can be shown by induction on the number of rounds.

Then, agreement and half of the validity condition—that if some process starts with 0, then 0 is the only possible decision value—follow from the extended Lemma 7.22, as before. The other half of the validity condition is true, because if there are no failures, all processes decide within the first three rounds.

We argue the strong termination property. If all processes fail, then this property is vacuously true. Otherwise, suppose that i is a nonfaulty process. Then during the time when i is the coordinator, every nonfaulty process decides. □

Complexity analysis. *ThreePhaseCommit*, in the version presented here, requires $3n$ rounds. Even if we permit all the processes to fail, this is still much higher than the bound of approximately n rounds that is generally achieved by stopping agreement algorithms of the sort studied in Chapter 6. Of course, the stopping agreement algorithms yield a different validity condition, but it is possible to modify them slightly to achieve the commit validity condition. So why are algorithms like *ThreePhaseCommit* considered better in practice?

The main reason is that the *ThreePhaseCommit* algorithm can be tailored to yield low complexity in the failure-free case. If no processes fail, then all processes decide by round 3. Then it is possible to add a simple protocol whereby processes can detect that every process has decided and can then discontinue participation in the rest of the termination protocol. With this addition, the entire algorithm requires only a small constant number of rounds and only $O(n)$ messages.

7.3.4 Lower Bound on the Number of Messages

We close this chapter (and Part I) by considering the number of messages that must be sent in order to solve the commit problem. Recall that the *TwoPhaseCommit* algorithm uses $2n - 2$ messages in the failure-free case. *ThreePhaseCommit* uses somewhat more, but still $O(n)$ if the algorithm is modified to

terminate early. In this section, we prove that it is not possible to do better than $2n-2$ in the failure-free case, even if we are satisfied with a blocking algorithm.

Theorem 7.25 *Any algorithm that solves the commit problem, even with weak termination, uses at least $2n-2$ messages in the failure-free execution in which all inputs are 1.*

For the rest of this section, we fix a particular commit algorithm A and let α_1 be the failure-free execution of A in which all inputs are 1. Our object is to show that α_1 must contain at least $2n-2$ messages.

We again use the definition of a *communication pattern* from Section 6.7. This time, we use a communication pattern to describe the set of messages that are sent in a failure-free execution. (We do not assume, as we have done in the past, that all processes send to all other processes at every round.) From any failure-free execution α of A, we extract a communication pattern $patt(\alpha)$ in the obvious way.

We also use the definition of the ordering $\leq_\gamma$ for a communication pattern γ, given in Section 5.2.2, to capture the flow of information between various processes at various times. We say that process i *affects* process j in a communication pattern γ provided that $(i,0) \leq_\gamma (j,k)$ for some k. The key idea in the lower bound is stated in the following lemma.

Lemma 7.26 *For every two processes i and j, i affects j in $patt(\alpha_1)$.*

Example 7.3.1 Lower bound for commit

Before proving Lemma 7.26, we give an example to show why it is true. Suppose that α_1 (the failure-free execution of A with all inputs equal to 1) includes exactly the messages depicted in the left-hand diagram in Figure 7.6.

By the validity and weak termination conditions, all processes must eventually decide 1 in α_1. Note that in $patt(\alpha_1)$, process 4 does not affect process 1; let us see what problems arise as a result. Consider an alternative execution α_1', which is the same as α_1 except that process 4's input is 0 and every process fails just after it first gets affected by process 4. Execution α_1' is depicted in the right-hand diagram in Figure 7.6; the failures are indicated by Xs. It is straightforward to show that $\alpha_1 \overset{1}{\sim} \alpha_1'$, which implies that process 1 also decides 1 in α_1'. But this violates the validity condition for α_1', yielding a contradiction.

The proof of Lemma 7.26 uses the same argument as in Example 7.3.1.

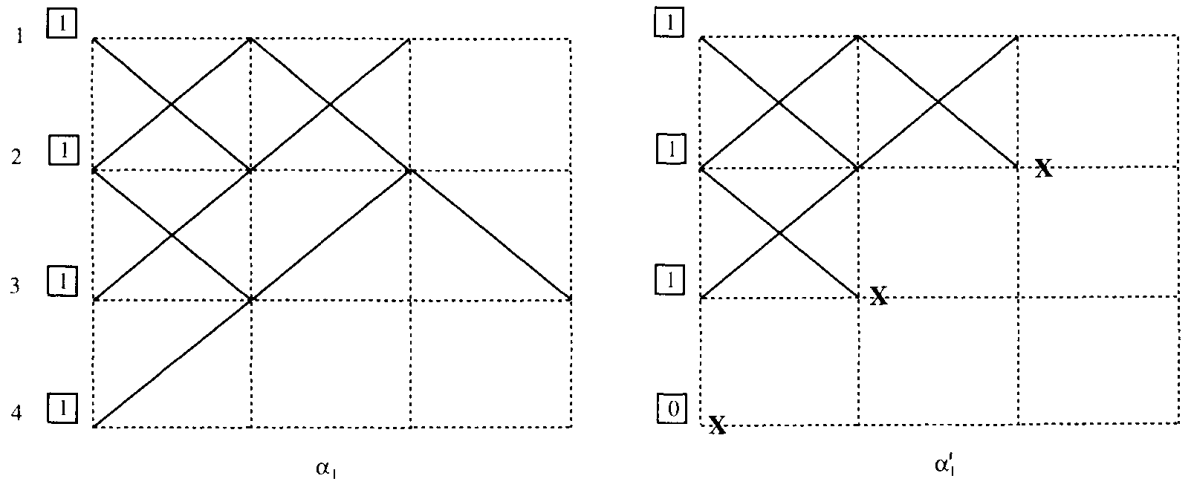

Figure 7.6: Messages sent in α_1 and α_1'.

Proof. By the validity and weak termination conditions, all processes must eventually decide 1 in α_1. Suppose that the lemma is false and fix two processes, i and j, such that i does not affect j in $patt(\alpha_1)$. By definition, it must be that $i \neq j$. Construct α_1' by changing process i's input to 0 and causing every process to fail just after it first gets affected by process i. Then $\alpha_1 \overset{j}{\sim} \alpha_1'$, so process j also decides 1 in α_1'. This violates the validity condition, yielding a contradiction. □

In order to complete the proof of Theorem 7.25, we must simply show that the requirement that every process affect every other process implies that there must be at least $2n - 2$ total messages. We use a lemma about communication patterns:

Lemma 7.27 *Let γ be any communication pattern. If in γ, each of a set of $m \geq 1$ processes affects each of the n processes in the system, then there are at least $n + m - 2$ messages (triples) in γ.*

Proof. By induction on m.

Basis: $m = 1$. Let i be the single process that we have assumed affects each of the n processes. Since i affects all n processes, γ must contain some message to each of the $n - 1$ processes other than i. This is a total of at least $n - 1$ messages, as needed.

Inductive step: We assume the lemma holds for m and show it for $m + 1$. Let I be a set of $m + 1$ processes that affect all n processes in γ. Without loss of generality, we can assume that in round 1, at least one of the processes in I sends a message to some process. For if not, then we could remove all

the initial rounds in which no process in I sends a message; in the remaining communication pattern, all processes in I would still affect all n processes. Let i be some process in I that sends a message at round 1 in γ.

Now consider the communication pattern γ', obtained from γ by removing a single round 1 message sent by i. Then all processes in $I - \{i\}$ affect all n processes in γ'. By induction, there are at least $n + m - 2$ messages in γ'. So γ contains at least $n + m - 1 = n + (m + 1) - 2$ messages, as needed. □

Now we can complete the proof of Theorem 7.25.

Proof (of Theorem 7.25). By Lemma 7.26, for every two processes i and j, i affects j in $patt(\alpha_1)$. Then Lemma 7.27 implies that there are at least $2n - 2$ messages in $patt(\alpha_1)$. □

7.4 Bibliographic Notes

The k-agreement problem has usually been called the "k-set agreement problem" in the literature. The problem was first introduced by Chaudhuri in [73] as a natural extension of the previously well-studied ordinary agreement problem. The *FloodMin* algorithm is taken from the work of Chaudhuri, Herlihy, Lynch, and Tuttle [75] and is based on an algorithm originally designed by Chaudhuri [73]. The lower bound argument for k-agreement is taken from [75, 76, 77]. Background for the algebraic topology used in the lower bound argument appears in Spanier's classical book on algebraic topology [266]. Sperner's Lemma was originally developed by Sperner [267] and is discussed in [266].

The work on approximate agreement is taken from a paper by Dolev, Lynch, Pinter, Stark, and Weihl [98]. Other work on this problem has been done by Fekete [110, 111] and by Attiya, Lynch, and Shavit [24]. The material on the commit problem, as well as the *TwoPhaseCommit* and *ThreePhaseCommit* algorithms, is taken from a book by Bernstein, Hadzilacos, and Goodman on database theory [50]. That book goes much further than this one in discussing practical implementation issues for the protocols, including how to handle recovery of failed processes. The lower bound on the number of messages for commit is taken from work by Dwork and Skeen [106].

7.5 Exercises

7.1. If the *FloodMin* algorithm for k-agreement is run for only $\lfloor \frac{f}{k} \rfloor$ rounds instead of $\lfloor \frac{f}{k} \rfloor + 1$, what is the largest number of different decisions that can be reached by nonfaulty processes?

7.2. Give a good upper bound for the length of sequence $seq(v)$, in the proof of Theorem 7.14. In order to do this, you will need to describe an explicit construction for the sequence.

7.3. Prove that the merge of a sequence of 1-runs is in fact a k-run. This involves showing that the conditions required in the definition of a run are satisfied, as well as the conditions involving the tokens.

7.4. Prove Lemma 7.4.

7.5. Prove Lemma 7.5.

7.6. Prove Lemma 7.6.

7.7. Prove Lemma 7.7.

7.8. Let $n = 5$, $k = f = 2$, and $r = 1$.

(a) Describe the Bermuda Triangle for these parameter values in detail, as well as its labelling with k-runs and process indices.

(b) Consider the trivial algorithm A that works as follows: all processes exchange values once, and each chooses the minimum value it receives. Describe the Sperner coloring C_A.

(c) Can you locate a particular tiny simplex in which three different values are decided upon, for algorithm A?

7.9. Fix any n and f, where $n > 3f$, any ϵ, any $w \in R^{\geq 0}$, and any $r \in \mathbb{N}$. Describe a particular execution of the *ConvergeApproxAgreement* algorithm with termination, for n, f, and ϵ, in which the multiset of nonfaulty processes' initial values has width at most w and in which termination takes more than r rounds.

7.10. *Research Question*: Modify *ConvergeApproxAgreement* so that the time until all processes decide is bounded above by a function of n, f, ϵ, and the width w of the multiset of nonfaulty processes' initial values.

7.11. Suppose that, instead of computing $mean(select(reduce(W)))$ in *ConvergeApproxAgreement*, the processes instead compute one of the following:

(a) $mean(select(W))$

(b) $mean(reduce(W))$

(c) $mean(W)$

Does the algorithm still solve the approximate agreement problem? Why or why not?

7.12. Prove that the approximate agreement problem can be solved in a network graph G, tolerating f Byzantine faults, if and only if both of the following hold:

(a) $n > 3f$

(b) $conn(G) > 2f$

7.13. Design an approximate agreement algorithm for the case of stopping failures.

(a) Try to minimize the number of processes needed, relative to the number of faults.

(b) Try to minimize the number of rounds required.

7.14. Formulate a variant of the approximate agreement problem that uses a fixed number r of rounds and in which ϵ is not predetermined. Each process starts with a real value, as before. After r rounds, the processes should output their final values. The validity condition is the same as before. The object is now to ensure the best possible agreement, expressed as an upper bound on the ratio of the width of the nonfaulty processes' final values to the width of the nonfaulty processes' initial values.

(a) What ratio is achieved by the *ConvergeApproxAgreement* algorithm in this setting?

(b) Prove a lower bound on the achievable ratio, in terms of n, f, and r. (*Hint:* Use chain argument ideas similar to those used in the proof of Theorem 6.33. Your upper and lower bounds probably will not match.)

7.15. Write code for the complete *ThreePhaseCommit* algorithm (including the termination protocol).

7.16. Prove carefully that Lemma 7.22 extends to any number of rounds of *ThreePhaseCommit*.

7.17. Give a careful description of a modification to the *ThreePhaseCommit* algorithm that permits processes to decide and halt quickly in the failure-free case. Your algorithm should use a small constant number of rounds and $O(n)$ messages, in the failure-free case. Prove its correctness.

7.18. Design an algorithm in the style of the stopping agreement algorithms in Chapter 6 that solves the commit problem with strong termination. Try to minimize the number of rounds.

7.19. *Research Question*: Design an algorithm that solves the commit problem with strong termination. Can you simultaneously obtain a worst-case number of rounds that is $n + k$ for some constant k, a small constant number of rounds for deciding and halting in the failure-free case, and a low communication complexity in the failure-free case?

7.20. Fill in all the details of the proof of Lemma 7.26. Where does the proof fail if we do not force any processes to fail when we construct α_1', but only change the initial value of process i from 1 to 0?

7.21. Design a non-blocking commit algorithm that uses the fewest messages you can manage, for failure-free runs. Can you prove that this number of messages is optimal?