CHAPTER 1

Introduction

1.1 Unsolvability Results and Lower Bounds

This book studies computation in distributed systems, specifically *unsolvability* results, which show that certain problems cannot be solved, and *lower bound* results, which show that certain problems cannot be computed when insufficient resources are available. In general, such impossibility results depend on assumptions about the system, for example, how processes communicate with one another or what kinds of failures can occur. They also depend on the types of algorithm allowed, for example, whether randomization can be used.

For solvable problems, we study their complexity under a number of different measures, most notably, time, number of messages and their size, number of shared variables, their type and size, and contention. Our goal is to find lower bounds on one or more of these resources or tradeoffs among them.

Note that, unlike the situation when we study algorithms, bigger lower bounds are better. Impossibility results for weaker problems are better because they automatically imply the same results for stronger problems. Similarly, it is better to prove impossibility results for stronger models, for example, with more powerful primitives, more synchrony, a source of randomness, or less faulty behaviour.

1.2 Why Study Impossibility Results?

Lower bounds and unsolvability results help us understand the nature of distributed computing:

- What makes certain problems hard? What parts of the problem requirements cause the difficulty? How do two different problems relate to one another?
- What makes certain systems powerful? What are the crucial limitations of real systems? How do two different systems (or, more precisely, formal models of those systems) relate to one another?

Lower bounds also tell us when to stop looking for better solutions, namely, when they match an existing upper bound. (Unfortunately, this does not happen very often.)

Impossibility results that assume restricted types of algorithms, for example, deterministic algorithms or algorithms that do not distinguish between different processes, indicate which approaches will not work.

If a problem needs to be solved despite an unsolvability result, the proof may indicate how to adjust either the problem statement (making it weaker) or the system model (making it stronger). In this manner, lower bounds have influenced real systems, by showing the system must satisfy certain assumptions, if important problems must be solved. For example, the unsolvability of consensus in asynchronous shared memory systems where processes communicate through registers has led to manufacturers including more powerful primitives such as compare&swap into their architectures.

Trying to prove a lower bound can suggest new and different algorithmic approaches for solving a problem. It can be fruitful to alternately work on getting a lower bound and getting a better algorithm, using the difficulties encountered in one to provide insight for the other.

Finally, lower bounds are fun to prove!

1.3 Structure of the Book

This book considers a variety of problems and models for distributed systems, emphasizing techniques, rather than results. After explaining a technique, we present several applications of it, going from simpler ones to more complicated ones. We have not always chosen the most important or most complicated results. Instead, we prefer proofs that expose the really important aspects of the techniques.

For each result, we begin by carefully specifying the model and the problem. We assume the reader is familiar with standard models of distributed computing, for example, as defined in Hagit Attiya and Jennifer Welch's book *Distributed Computing, second edition*, Wiley, 2004. Throughout the book, we use $p_0, \ldots, p_{n-1}$ to denote the processes of the model in which we are working. To distinguish operations we are trying to implement from those that we are using for an implementation, we denote the former with upper case letters, for example, WRITE, we denote the latter using bold lower case letters, for example, write, and we call the latter *primitives*.

CHAPTER 2

Indistinguishability

Most impossibility results in distributed computing follow from a lack of knowledge or uncertainty about the system. At any point in time, the state of a process, including the value of its input variables, is the knowledge the process has about the system. To solve many distributed computing problems, processes need to learn information about the states of other processes. Proofs of unsolvability results show that this knowledge cannot be obtained; proofs of lower bounds show that this knowledge cannot be obtained with limited resources.

Lack of knowledge stems from uncertainty about many aspects of the system, including the inputs of other processes (since different processes get different inputs), asynchrony (how many steps other processes have taken, what messages have been sent and received), and failures (crashes, omissions, and malicious processes). How do we say that a process doesn't know something? If its local knowledge is compatible with two different executions, then it doesn't know which of the two executions has occurred. A key method for capturing lack of knowledge is by arguing about the indistinguishability of certain executions or configurations.

A *configuration* describes the system at some point in time. It consists of the states of all processes and the state of the environment (for example, the values of all shared variables, or the contents of all message channels). Two configurations, C and C', are *indistinguishable* to a process p_i, if it has the same state in both configurations. In other words, p_i does not know whether it is in C or C'. This is denoted $C \overset{p_i}{\sim} C'$. If P is a set of processes and $C \overset{p_i}{\sim} C'$ for all $p_i \in P$, we write $C \overset{P}{\sim} C'$. Note that, in some of the distributed computing literature, the definition of $C \overset{p_i}{\sim} C'$ also requires that the state of the environment is the same in C and C'. We prefer to address the state of the environment separately, because we often consider configurations that differ in parts of the environment that will not affect p_i.

At any configuration, there is a fixed set of events that can occur, each of which affects one process. Some examples of events are message m is delivered to process p_i on channel c, process p_i writes value v to register r, and process p_i reads value v from register r. An *execution* is a sequence of alternating configurations and events, starting with a configuration, such that each event can occur at the configuration which precedes it and which results in the configuration that follows it. If an execution is finite, it ends with a configuration. A *solo execution* is an execution in which every event is by the same process.

The *history* associated with an execution is its sequence of events. A sequence of events σ *can occur starting at* a configuration C if there is an execution that begins with C whose history is σ. If σ is finite, we use $C\sigma$ to denote the last configuration in this execution. We often partition a history into a set of n *local histories*, one for each process, consisting of the events which affect that process.

Whether σ can occur starting at C depends on the state of the environment in C. Specifically, in shared memory systems, it depends on the values of the shared objects accessed by σ and, in message passing systems, it depends on the contents of the message channels on which messages are delivered in σ.

Two executions, α and α', starting from configurations C and C', respectively, are *indistinguishable* to a set of processes P if $C \overset{P}{\sim} C'$ and each process $p_i \in P$ has the same local history in both executions. This is denoted $\alpha \overset{P}{\sim} \alpha'$.

If two configurations, C and C', are indistinguishable to a set of processes and the same finite sequence of events, σ, can occur starting at each, then the two resulting configurations, $C\sigma$ and $C'\sigma$ will also be indistinguishable to these processes. The following lemma and its corollary are useful for identifying such situations. They are straightforward to prove by induction.

Lemma 2.1. *Let σ be a sequence of events by some set of processes P that can occur starting from configuration C. If $C \overset{P}{\sim} C'$ and the part of the environment accessed by σ has the same value in C and C', then σ can occur starting from C' and, if σ is finite, $C\sigma \overset{P}{\sim} C'\sigma$.*

Corollary 2.2. *Let α be an execution starting from some configuration C, all of whose events are by processes in some set P. If $C \overset{P}{\sim} C'$ and the part of the environment accessed by α has the same value in C and C', then there is an execution α' starting from C' such that $\alpha \overset{P}{\sim} \alpha'$.*

If some process cannot distinguish between two executions of an algorithm, but it must produce different outputs in each, then the algorithm is incorrect. It is often useful to think of these bad executions as being produced by an *adversary*, which controls, depending on the circumstances, the inputs processes receive, the order in which they take steps, when to deliver messages and the way failures occur (i.e., what kind of failures, by what processes, and when). The adversary tries to limit the amount of knowledge processes have, either forever, or for as long as possible, by keeping the execution indistinguishable from other executions in which the results should be different.

An algorithm is *wait-free* if each process that doesn't fail completes its task after taking some finite number of steps, no matter how the adversary does its scheduling. A weaker condition is *solo termination*, also known as *obstruction freedom*, in which a process completes its task, provided it is given sufficiently many consecutive steps by the adversary. Thus, lower bounds assuming solo termination also apply to wait-free algorithms.

In the remainder of this chapter, we present three fairly simple proofs of impossibility results that rely on indistinguishability, followed by two which are more involved.

Section 2.1 gives a lower bound on the tradeoff between the worst case time to perform a read and the worst case time to perform a write in any implementation of a register in a message passing system. In Section 2.2, we present a lower bound on the size of shared memory necessary for first-come first-served mutual exclusion. Section 2.3 contains a lower bound on the worst-case step complexity of approximate agreement. A lower bound on the number of rounds to solve consensus, as a function of the number of process failures that might occur, is presented in Section 2.4. Finally, in Section 2.5, we prove that wait-free set consensus is impossible in an asynchronous system using only single-writer registers.

2.1 A Tradeoff Between Read and Write Times in the Implementation of a Register

Consider the problem of implementing a register in a message passing system. A solution to this problem allows one to convert algorithms designed for a shared memory system to be used in a message passing system. Understanding the complexity of this problem tells us how much overhead is incurred in doing so. It also allows us to transfer lower bounds proved in a message passing system to shared memory.

A solution consists of two algorithms, READ and WRITE(v), for each process p_i. The input parameter v may have any value that can be stored in the register. A very weak requirement for such an implementation is the following: In any execution in which no WRITE overlaps any other operation, each READ must return the last value written before it began.

We consider a message passing model in which processes communicate by sending messages directly to one another through a complete network. We assume that the system is semisynchronous: each step by a process can take up to 1 unit of time to be executed and messages can take up to d units of time to be delivered. We can assume that no processes fail and communication is reliable.

For any implementation, let R denote the worst case time to perform a READ and let W denote the worst case time to perform a WRITE. We use an indistinguishability argument to prove the following tradeoff.

Theorem 2.3. *$R + W \geq d$.*

Proof. Suppose not. Consider an execution α_1 in which process p_1 performs WRITE(1) starting at time 0, process p_0 performs READ starting at time W, and all messages sent have delay d. Then, by time $W + R < d$, process p_0 has returned its response, but it has not received any messages.

Let α_2 be the execution that is the same as α_1, except that p_1 performs WRITE(2) instead of WRITE(1). These two executions are indistinguishable to p_0 during the first $W+R$ units of time, so it must return the same result for its READ in both executions. Thus, in at least one of these two executions, it returns an incorrect response. □

This result is from a 1988 Princeton University technical report entitled *PRAM: A scalable shared memory* by Richard Lipton and J. Sandberg.

2.2 A Space Lower Bound for First-Come First-Served Mutual Exclusion

In the mutual exclusion problem, processes may need temporary exclusive access to a shared resource. A process which has this access is said to be in the *critical section*. To get the resource, a process performs an *entry protocol*. When a process has finished with the resource, it performs an *exit protocol*. A process that does not currently care about the resource is said to be in the *remainder section*.

A mutual exclusion algorithm consists of code for the entry and exit protocols for each process. It must satisfy the following properties.

- *Mutual Exclusion*: two or more processes are never simultaneously in the critical section.
- *Deadlock Freedom*: starting from any configuration in which some process is performing the entry protocol and no process is in the critical section, some process eventually enters the critical section.
- *Unobstructed Exit*: a process can always perform the exit protocol using only a finite number of its own steps.

A mutual exclusion algorithm is *first-come first-served* if each entry protocol begins with a section of code, called a *doorway*, and processes enter the critical section in the order that they perform the doorway. More precisely, for any two processes $p_i \neq p_j$, if p_i completes the doorway of some instance of the entry protocol before p_j begins some other instance of the entry protocol, then p_i completes its instance of the entry protocol and enters the critical section before p_j does. The doorway has the property that it can always be performed by a process using only a finite number of its own steps.

In this section, we consider asynchronous shared memory models which support arbitrary objects. There are no process failures. Moreover, when a process is in the critical section, it eventually finishes using the resource (and performs the exit protocol). We use an indistinguishability argument to prove a lower bound on the space needed to solve this problem.

Theorem 2.4. *Any first-come first-served mutual exclusion algorithm has at least n possible values for its shared memory.*

Proof. Consider any mutual exclusion algorithm in which the shared memory has less than n possible values. We show that an adversarial scheduler can construct an execution starting from the initial configuration in which the first-come first-served property is violated.

Let C_0 be an initial configuration in which all processes are in the remainder section. For $i = 0, \ldots, n-1$, starting from configuration C_i, consider the finite history in which process p_i takes steps until it completes its doorway. Let C_{i+1} be the resulting configuration and let v_{i+1} be the value of the shared memory in this configuration.

By the pigeon hole principle, there exist i and j, $1 \leq i < j \leq n$ such that $v_i = v_j$. Let $P = \{p_0, \ldots, p_{i-1}\}$. Starting from C_i, consider a scheduler that repeatedly schedules the processes in P in round robin order. By deadlock freedom, eventually some process in P enters the critical section. When a process enters the critical section, it begins the exit protocol at its next turn. By unobstructed exit, it eventually completes the exit protocol. After entering the remainder section, it performs the entry protocol again, beginning at its next turn. This happens repeatedly. Eventually some process $p_k \in P$ enters the critical section a second time. Let σ denote the finite sequence of events performed starting from C_i until this occurs.

Since $C_i \overset{P}{\sim} C_j$ and $v_i = v_j$, it follows from Lemma 2.1 that σ can be performed starting from C_j. Consider the execution from C_0 to C_j followed by the sequence of events σ. In this execution, process p_{j-1} completes its doorway before process p_k begins its doorway for the second time. However, process p_k enters the critical section twice, whereas process p_{j-1} does not enter the critical section at all. This violates the first-come first-served property. □

This lower bound is due to Burns, Jackson, Lynch, Fischer, and Peterson, in their paper, *Data Requirements for Implementation of N-Process Mutual Exclusion Using a Single Shared Variable*, which appeared in JACM in 1982.

2.3 A Lower Bound on the Step Complexity of Approximate Agreement

In the *approximate agreement* problem, processes have to output values that are close to one another. Formally, each process p_i has a private input value x_i and, if it doesn't fail, has to output a value y_i. The processes all know an accuracy parameter $\epsilon > 0$. The output values must satisfy the following two properties.

- *ϵ-Agreement*: All output values are within ϵ of each other.
- *Validity*: All output values are between the smallest and largest input values, i.e., $\min\{x_0, \ldots, x_{n-1}\} \leq y_i \leq \max\{x_0, \ldots, x_{n-1}\}$ for all $i \in \{0, \ldots, n-1\}$.

In particular, if all the input values are the same, all the output values must equal this input value. One place this problem arises is in clock synchronization when processes attempt to maintain clock values that are close to one another.

We consider an asynchronous shared memory model with no process failures and only single-writer registers.

Theorem 2.5. *For $x_0, \ldots, x_{n-1} \in \{0, 1\}$ and $\epsilon < 1$, any algorithm for approximate agreement that satisfies solo termination has worst case step complexity at least $n - 1$.*

Proof. The proof is by contradiction. Consider any approximate agreement algorithm and let α be the solo execution by a process p_i starting from an initial configuration C_0 in which the input values of all processes are 0. Then, by solo termination and validity, it must output value 0. Suppose this execution takes fewer than $n - 1$ steps. Then process p_i doesn't read the single-writer register r_j of some process $p_j \neq p_i$.

Next, consider the solo execution β by process p_j starting from an initial configuration, C_1, in which the input values of all processes are 1. By solo termination and validity, it must output value 1.

Now, consider the solo execution β' by process p_j starting from an initial configuration C in which its input value is 1, but all other input values are 0. Note that $C \overset{p_j}{\sim} C_1$ and each of the single-writer registers has the same value (i.e., its initial value) in both these configurations, so $\beta' \overset{p_j}{\sim} \beta$. Hence, process p_j outputs 1 in execution β'.

Finally, let C' be the configuration at the end of β' and let α' denote the solo execution α' by process p_i starting from C'. Since every single-writer register, except r_j, has the same value in C' and C_0 and process p_i does not read from r_j during α, it follows from Corollary 2.2 that $\alpha' \overset{p_j}{\sim} \alpha$. Hence, process p_i outputs 0 in execution α'.

But this means that, in execution $\beta'\alpha'$, process p_j outputs 1 and process p_i outputs 0. This violates ϵ-agreement. □

A wait-free approximate agreement algorithm using multi-writer registers with $O(\log(1/\epsilon))$ step complexity is presented in the paper *Faster Approximate Agreement with Multi-Writer Registers* by Erik Schenk, Proceedings of FOCS, 1995, pages 714–723. When $x_0, \ldots, x_{n-1} \in \{0, 1\}$ and $\epsilon = \frac{1}{2}$, it has $O(1)$ step complexity. Together with Theorem 2.5, this implies that single-writer registers are less powerful than multi-writer registers.

Theorem 2.6. *Any implementation of a multi-writer register shared by n processes using only single-writer registers has $\Omega(n)$ step complexity in the worst case.*

Thus, lower bounds on a particular problem can be used to prove that one model is more powerful than another model. Theorem 2.6 can also be proved directly, using an argument similar to that in the proof of Theorem 2.5.

In *Atomic Shared Register Access by Asynchronous Hardware*, by Paul Vitányi and Baruch Awerbuch, Proceedings of FOCS, 1986, pages 233–243, there is a wait-free implementation of a multi-writer register with $O(n)$ step complexity. By Theorem 2.6, this is optimal.

2.4 Chain Arguments for Consensus

In a *chain argument*, the idea is to construct a chain or sequence of executions such that each pair of consecutive executions in the chain is indistinguishable to at least one process. If, in each execution, all processes must have the same result, it follows that the processes have the same result in all executions in the chain. This leads to a contradiction if the result at one end of the chain must differ from the result at the other end./

For any two executions, α and α', we write $\alpha \sim \alpha'$ if there is a process p_i such that $\alpha \overset{p_i}{\sim} \alpha'$. Let $\approx$ denote the transitive closure of $\sim$. In other words, $\alpha \approx \alpha'$ if and only if there is a chain of executions $\alpha = \alpha_0, \alpha_1, \ldots, \alpha_k = \alpha'$ such that α_{i-1} and α_i are indistinguishable to at least one process, for $i = 1, \ldots, k$, i.e., for each i, there exists a process p_j such that $\alpha_{i-1} \overset{p_j}{\sim} \alpha_i$.

Consensus is one of the most widely studied problems in the theory of distributed computing and is used as a building block in many algorithms. The *consensus* problem requires all processes that do not fail to output the same value. A trivial solution is to have each process simply output the value 0. The problem becomes more interesting if each process has a private input value and is required to output this value when every other process has the same input value.

Formally, each process p_i has a private input value x_i and, if it doesn't fail, it has to output a value y_i. The output values must satisfy the following two properties:

- *Agreement*: All output values are the same.
- *Validity*: If all input values are the same, then no other value is output.

Binary consensus is a restricted version of the consensus problem, where all input values are in $\{0, 1\}$.

We say that an execution of a consensus algorithm *decides* a value v if some process outputs v during the execution. If α and α are executions that decide v and v', respectively, and $\alpha \approx \alpha'$, it follows that $v = v'$.

We begin with an important observation about binary consensus algorithms, which is proved by a simple chain argument. It applies to both synchronous and asynchronous models in which processes can fail. We will use this observation in this section and, again, in Chapter 7.

Lemma 2.7. *Any binary consensus algorithm has an initial configuration from which there are two executions that decide different values. In one of these executions, no processes fail. In the other, one process crashes before taking any steps, but no other processes fail.*

Proof. For $i = 0, \ldots, n$, let C_i denote the initial configuration in which the first i processes, $p_0, \ldots, p_{i-1}$, have input 1 and the rest have input 0, i.e.,

$$x_j = \begin{cases} 1 & \text{for } j < i \\ 0 & \text{for } j \geq i. \end{cases}$$

Let v_i be the value decided by some execution α_i, starting from C_i, in which no processes fail. In configuration C_0, all processes have input 0, so by validity, $v_0 = 0$. Similarly, in configuration C_n, all processes have input 1, so $v_n = 1$.

Since $v_0 \neq v_n$, there exists $j \in \{0, \ldots, n-1\}$ such that $v_j \neq v_{j+1}$. Let α be an execution starting from C_j in which process p_j crashes before taking any steps and no other process fails. If α does not decide v_j, then the claim is true for executions α_j and α, which both start from C_j. So, suppose that α decides v_j.

Configurations C_j and C_{j+1} are the same, except for the state of process p_j. By Corollary 2.2, there is an execution α' starting from C_{j+1} such that $\alpha \overset{p}{\sim} \alpha'$ for all $p \neq p_j$. Hence α' decides v_j, so the claim is true for executions α_{j+1} and α', which both start from C_{j+1}. □

For the rest of this section, we consider a synchronous message passing model in which processes can only fail by crashing. In each round, every process that has not terminated and does not crash sends a message to every other process and then receives all the messages that were sent to it in that round, ordered by the identifiers of the processes that sent them. In a round in which a process crashes, it sends messages to an arbitrary prefix (chosen by an adversary) of the sequence of other processes, ordered by their identifiers. A process that crashes sends no messages in any subsequent round. Furthermore, we assume that at most f failures occur during each execution.

Now, we will prove a lower bound on the number of rounds needed to solve consensus in this model. The key to the proof is the following technical lemma, which uses a more complicated chain argument. The chain of executions that it constructs is very long.

Lemma 2.8. *Consider any f-round execution α of a consensus algorithm for $n \geq f+2$ processes in a synchronous message passing model, with at most one crash in each round. Let γ be the f-round execution that is the same as α during its first r rounds and has no crashes after round r, for some $0 \leq r \leq f$. Then $\alpha \approx \gamma$.*

Proof. By backwards induction on r. If $r = f$, then $\alpha = \gamma$, so $\alpha \approx \gamma$. Suppose $0 \leq r < f$ and assume the claim is true for $r+1$.

Let β be the f-round execution that is the same as α during its first $r+1$ rounds and has no crashes after round $r+1$. By the induction hypothesis, $\alpha \approx \beta$. Thus, it suffices to show $\beta \approx \gamma$. This is illustrated in Figure 2.1, where a round that may contain a crash is indicated by a shaded box.

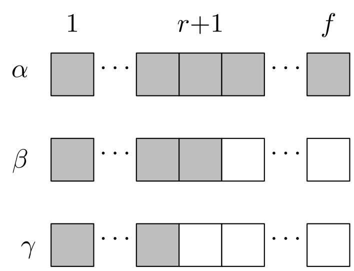

Figure 2.1. Some of the executions considered in the proof of Lemma 2.8.

If no process crashes during round $r+1$ of execution β, then $\beta = \gamma$ and, hence, $\beta \approx \gamma$. So, suppose there is a process p_i that crashes during round $r{+}1$. By assumption, no other process crashes during round $r+1$.

Let P denote the set of processes that do not fail during β. From the model, we know that $|P| \geq n - f \geq 2$. Let Q be the subset of processes in P to which p_i does *not* send a message during round $r+1$. These are the processes that can distinguish β from γ at the end of round $r+1$. If $Q = \phi$, let $t = 0$. Otherwise, let $t = |Q|$ and let $q_1, \ldots, q_t$ be the processes in Q in increasing order by identifier.

We construct a chain of executions between β and γ. Let $\beta_0 = \beta$ and, for $1 \leq k \leq t$, let β_k be the f-round execution that has no crashes after round $r+1$ and is the same as β during its first $r{+}1$ rounds, except that p_i also sends messages to $q_1, \ldots, q_k$ during round $r+1$.

First suppose that $f = r+1$. For $1 \leq k \leq t$, the only difference between β_{k-1} and β_k is whether p_i sends a message to q_k in round $r+1$. Therefore $\beta_{k-1} \overset{p}{\sim} \beta_k$ for all processes $p \in P - \{q_k\}$. Since $|P| \geq 2$, there is at least one process in this set, so $\beta_{k-1} \approx \beta_k$. Hence $\beta \approx \beta_t$. In β_t, process p_i crashes in the last round, after sending messages to all processes in P, so no process in P can learn whether p_i crashed. Note that γ is the same as β_t, except that p_i does not crash, so $\beta_t \overset{P}{\sim} \gamma$. Since $P \neq \phi$, $\beta_t \approx \gamma$ and, thus, $\beta \approx \gamma$.

Now suppose that $f > r + 1$. This case is more complicated because processes in P can communicate with one another in rounds $r+2, \ldots, f$. We inductively construct a chain of executions between β_{k-1} and β_k, for $1 \leq k \leq t$. Let γ_k be the f-round execution that is the same as β_k for its first $r + 1$ rounds, but, at the beginning of round $r+2$, process q_k crashes without sending messages to any other process and has no crashes after round $r + 2$. Similarly, let γ'_k be the f-round execution that is the same as β_{k-1} for its first $r+1$ rounds, but, at the beginning of round $r+2$, process q_k crashes without sending messages to any other process and has no crashes after round $r + 2$. It follows from the induction hypothesis that $\beta_k \approx \gamma_k$ and $\beta_{k-1} \approx \gamma'_k$.

Note that, up to the end of round $r + 1$, γ_k and γ'_k are indistinguishable to all processes in $P - \{q_k\}$. Since process q_k sends no messages in either execution after round $r + 1$, Corollary 2.2 implies that $\gamma_k \overset{p}{\sim} \gamma'_k$ for all $p \in P - \{q_k\}$. Since $|P| \geq n - f \geq 2$, there is at least one process in this set, so $\gamma_k \approx \gamma'_k$. Thus $\beta_{k-1} \approx \beta_k$ and, hence, $\beta \approx \beta_t$.

In execution β_t, process p_i crashes at the end of round $r+1$, after sending messages to all other processes, and no processes crash in subsequent rounds. Let β' be the execution that is the same as β_t, except that p_i crashes at the beginning of round $r + 2$, before sending messages to any other other processes. Then $\beta_t \overset{P}{\sim} \beta'$. Hence $\beta_t \approx \beta'$. Since β' has no crashes during round $r+1$, the first $r+1$ rounds of β' and γ are the same. By the induction hypothesis, $\beta' \approx \gamma$, so $\beta \approx \gamma$.

Therefore the claim is true for round r and, thus, by induction, for $0 \leq r \leq f$. □

Theorem 2.9. *Any consensus algorithm with $n \geq f + 2$ processes that tolerates f crashes requires more than f rounds, even if at most one process crashes in each round.*

Proof. Suppose there is a consensus algorithm with $n \geq f+2$ processes that tolerates f crashes and uses at most f rounds. By Lemma 2.7, there is an initial configuration from which there are two executions α and γ that decide different values and in which no processes crashes, except for one process that crashes at the beginning of the first round of α. Lemma 2.8 implies that $\alpha \approx \gamma$. Hence these executions decide the same value. This is a contradiction. □

The proof of Lemma 2.7 is due to Fischer, Lynch, and Paterson, from their paper, *Impossibility of Distributed Consensus with One Faulty Processor*, JACM 32, 1985, pages 374–382. Theorem 2.9 appeared in Dwork and Moses, *Knowledge and Common Knowledge in a Byzantine Environment: Crash Failures*, Information and Computation, 88, 1990, pages 156–186.

2.5 Impossibility of Set Consensus

The *k-set consensus* problem is an extension of the consensus problem in which nonfaulty processes decide on at most k different values. Formally, each process p_i has a private input value x_i and, if it doesn't fail, it has to output a value y_i. The output values must satisfy the following two properties:

- *k-Agreement:* There are at most k different output values.
- *Validity:* Every output value is one of the input values.

The *consensus* problem is the special case with $k = 1$.

We consider an asynchronous shared memory system in which processes communicate using single-writer registers of unbounded size. Any number of process crash failures are allowed.

It is trivial to solve n-set consensus for n processes: Each process can simply output its input value. However, if the number of different output values must be smaller than the number of processes, the problem become impossible to solve.

Theorem 2.10. *There is no wait-free algorithm for n processes that solves $(n-1)$-set consensus.*

The proof is by contradiction. Suppose there is such an algorithm for n processes that solves $(n-1)$-set consensus. It suffices that each process has one single-writer register, because the single-writer registers have unbounded size. We may also assume that when a process writes to its register, it writes its entire history. An algorithm that does this is called a *full information algorithm*.

Since we are not concerned with the step complexity of the algorithm, there is no loss of generality in assuming that each process starts with a write to its register and alternates between writing to its register and reading the registers of all $n-1$ other processes, in order of their process identifiers. For our proof, it suffices to restrict attention to special executions, which are *induced by* finite sequences of nonempty sets of processes, as follows: Every process p_i starts with its identifier i as input. Given a sequence of nonempty sets of processes, $B_1, B_2, \ldots, B_r$, the execution proceeds in r rounds. In the ℓ'th round, each process in B_ℓ takes n steps. First, each process in B_ℓ, in increasing order of identifier, writes to its register. Then, each process in B_ℓ, in increasing order of identifier, reads the registers of all $n-1$ other processes. For example, the three round execution β induced by $\{p_1, p_2\}, \{p_2, p_3\}, \{p_4\}$ is

$$
\begin{array}{lcccccccccc}
p_1: & w & & R & & & & & & & \\
p_2: & & w & & R & w & & R & & & \\
p_3: & & & & & & w & & R & & \\
p_4: & & & & & & & & & w & R
\end{array}
$$

Here w denotes a write by a process to its register and R denotes a read by a process of each of the other $n-1$ registers.

If a process p_i reads the single-writer register of another process p_j in the ℓ'th round of an execution, it learns the number of sets among $B_1, \ldots, B_\ell$ to which p_j belongs, which is how many times p_j participated during the first ℓ rounds. It also learns the state of process p_j immediately prior to the last round in which p_j participated. For example, every process in B_1 learns which other processes belong to B_1 and every process in $B_2 - B_1$ learns which other processes belong to $B_1 \cap B_2$, $(B_1 - B_2) \cup (B_2 - B_1)$, and $\overline{B_1} \cap \overline{B_2}$. If B_1 and B_2 are disjoint, then a process in B_2 cannot determine whether another process is in $B_2 - B_1$ or $B_1 - B_2$. Hence, the executions induced by $B_1 \cup B_2$ and B_1, B_2 are indistinguishable to the processes in B_2. However, the processes in B_1 can distinguish between these two executions. More generally, because processes take steps in a fixed order within each round, the following result can be proved inductively.

Lemma 2.11. *If β and β' are both executions induced by finite sequences of sets and they are indistinguishable to all processes, then those sequences are the same and $\beta = \beta'$.*

We are particularly interested in pairs of executions that are distinguishable by exactly one process. For example, let β_1 be the execution induced by $\{p_1\}, \{p_2\}, \{p_2, p_3\}, \{p_4\}$:

$$\begin{array}{lcccccccccc}
p_1\colon & w & R & & & & & & & & \\
p_2\colon & & & w & R & w & & R & & & \\
p_3\colon & & & & & & w & & R & & \\
p_4\colon & & & & & & & & & w & R
\end{array}$$

and let β_2 be the execution induced by $\{p_1, p_2\}, \{p_2\}, \{p_3\}, \{p_4\}$:

$$\begin{array}{lcccccccccc}
p_1\colon & w & & R & & & & & & & \\
p_2\colon & & w & & R & w & R & & & & \\
p_3\colon & & & & & & & w & R & & \\
p_4\colon & & & & & & & & & w & R.
\end{array}$$

Then β is indistinguishable from β_1 to all processes except p_1 and β is indistinguishable from β_2 to all processes except p_2, that is, $\beta_1 \stackrel{P-\{p_1\}}{\sim} \beta$, $\beta_1 \stackrel{p_1}{\not\sim} \beta$, $\beta_2 \stackrel{P-\{p_2\}}{\sim} \beta$, and $\beta_2 \stackrel{p_2}{\not\sim} \beta$.

Observation 2.12. *For every process p_i, if β and β' are the executions induced by the sequences $B_1, \ldots, B_r$ and $B_1, \ldots, B_r, \{p_i\}$, respectively, then $\beta \stackrel{P-\{p_i\}}{\sim} \beta'$ and $\beta \stackrel{p_i}{\not\sim} \beta'$.*

The next lemma gives a different situation in which two sequences of sets induce executions that are indistinguishable to all processes except p_i.

Lemma 2.13. *If p_i participates in the execution β induced by $B_1, B_2, \ldots, B_r$ and $B_r \neq \{p_i\}$, then there is a unique sequence of sets $B'_1, B'_2, \ldots, B'_{r'}$ such that $B'_{r'} \neq \{p_i\}$, $\beta \stackrel{P-\{p_i\}}{\sim} \beta'$, and $\beta \stackrel{p_i}{\not\sim} \beta'$, where β' is the execution induced by $B'_1, B'_2, \ldots, B'_{r'}$.*

Proof. Let ℓ be the latest round of β in which p_i participates. If $B_\ell \neq \{p_i\}$, split B_ℓ into two nonempty sets, the first of which contains only p_i and the second of which contains the rest of B_ℓ. Then $r' = r + 1$ and

$$B'_h = \begin{cases} B_h \text{ if } 1 \leq h < \ell \\ \{p_i\} \text{ if } h = \ell \\ B_\ell - \{p_i\} \text{ if } h = \ell + 1 \\ B_{h-1} \text{ if } \ell + 1 < h \leq r'. \end{cases}$$

Note that, if $\ell = r$, then $B'_{r'} = B_\ell - \{p_i\} \neq \{p_i\}$ and, if $\ell < r$, then $B'_{r'} = B_r \neq \{p_i\}$.

If $B_\ell = \{p_i\}$, then $\ell < r$ and $p_i \notin B_{\ell+1}$ (since ℓ is the latest round in which p_i participates). In this case, merge B_ℓ with $B_{\ell+1}$, so $r' = r - 1$ and

$$B'_h = \begin{cases} B_h \text{ if } 1 \leq h < \ell \\ B_\ell \cup B_{\ell+1} \text{ if } h = \ell \\ B_{h+1} \text{ if } \ell + 1 \leq h \leq r'. \end{cases}$$

Since $B'_{r'} \supseteq B_r \neq \{p_i\}$ and $B_r \neq \phi$, it follows that $B'_{r'} \neq \{p_i\}$.

In both cases, $\beta' \overset{p_i}{\not\sim} \beta$. However, the induced executions β and β' are the same prior to round ℓ and they become distinguishable to process p_i only after it last writes to its single-writer register in round ℓ. The processes in $P - \{p_i\}$ that participate in round ℓ of the shorter of these two executions (i.e., with r rounds) cannot tell the difference between it and round $\ell + 1$ of the longer execution. Since p_i takes no steps after round ℓ in either execution and the last $r - \ell$ rounds of these executions are the same, these executions remain indistinguishable to every other process from round ℓ onwards. Thus, $\beta' \overset{P-\{p_i\}}{\sim} \beta$.

To prove uniqueness, consider any sequence of sets $B''_1, \ldots, B''_{r''}$ with $B''_{r''} \neq \{p_i\}$ that induces an execution β'' such that $\beta'' \overset{p_i}{\not\sim} \beta$ and $\beta' \overset{P-\{p_i\}}{\sim} \beta$. Since p_i participates in round ℓ of β and $B_r \neq \{p_i\}$ it follows that the first $\ell - 1$ rounds of β and β'' are indistinguishable to all processes, $B_\ell \neq \{p_i\}$, and p_i does not participate during any later round. By Lemma 2.11, $B''_1, \ldots, B''_{\ell-1} = B_1, \ldots, B_{\ell-1}$.

The last $r - \ell$ rounds of β and β'' are indistinguishable to all processes except p_i, which does not participate. It follows by Lemma 2.11 that the last $r - \ell$ rounds of β'' and β are the same and $B_{\ell+1}, \ldots, B_r = B''_{r''-r+\ell+1}, \ldots, B''_{r''}$.

Finally, between this prefix and suffix, if a process writes a different number of times or sees a different number of writes by another process, it will have a different state. Thus, the only writes in this part of the execution must be by processes in B_ℓ, all of them must write exactly once, and all the processes in $B_\ell - \{p_i\}$ must write in the same round. Therefore the sequence that induced β'' must be either $B_1, B_2, \ldots, B_r$ or $B'_1, B'_2, \ldots, B'_{r'}$. But $\beta'' \neq \beta$, since $\beta'' \overset{p_i}{\not\sim} \beta$. Therefore $\beta'' = \beta'$ and $B''_1, \ldots, B''_{r''} = B'_1, B'_2, \ldots, B'_{r'}$. $\square$

We say that a process p_i is *seen in the ℓ'th round* of the execution induced by $B_1, B_2, \ldots, B_r$, if $p_i \in B_\ell$ and $\cup_{h=\ell}^{r} B_h \neq \{p_i\}$, i.e. there is some other process that participates in round ℓ or later. If a process p_i participates in the execution β induced by $B_1, B_2, \ldots, B_r$, but is not seen, we say that it is *unseen* in β. This means that p_i takes all its steps after all other participating processes have stopped taking steps, i.e. there exists $\ell \in \{1, \ldots, r\}$ such that $p_i \notin B_h$ for $1 \leq h < \ell$ and $B_h = \{p_i\}$ for $\ell \leq h \leq r$. At most one process is unseen in the execution induced by a sequence of sets of processes. For example, p_4 is unseen in β, β_1, and β_2.

An *m-process normal execution* is an execution induced by a sequence of subsets of $\{p_0, \ldots, p_{m-1}\}$ such that each of these m processes p_i has input $x_i = i$ and outputs a value $y_i \in \{0, \ldots, m-1\}$ in the last round in which it participates. Let N_m denote the set of all m-process normal executions in which all m input values are output, i.e. $\{y_0, \ldots y_{m-1}\} = \{0, \ldots, m-1\}$.

Lemma 2.14. *$|N_m|$ is odd, for $1 \leq m \leq n$.*

Proof. The proof is by induction. Since the algorithm is deterministic and wait-free, there is exactly one 1-process normal execution. In this execution, p_0 outputs 0. Thus $|N_1| = 1$, which is odd.

Let $1 \leq m \leq n-1$ and assume that $|N_m|$ is odd; note that since the algorithm is deterministic and wait-free, N_{m+1} is finite. Consider the set A_{m+1} of pairs (α, p_i), where $0 \leq i \leq m$ and α is an $(m+1)$-process normal execution in which processes other than p_i output all the values $\{0, \ldots, m-1\}$. We start by showing that $|A_{m+1}|$ is odd. There are three cases:

First, suppose that p_i is seen in α. Let β be the execution obtained from α by removing all rounds from the end of α in which only p_i participates. By Observation 2.12, $\beta \overset{P-\{p_i\}}{\sim} \alpha$. By Lemma 2.13, there is a unique execution β' such that the set of participants in its last round is not $\{p_i\}$, $\beta \overset{p_i}{\not\sim} \beta'$, and $\beta \overset{P-\{p_i\}}{\sim} \beta'$. Let α' be the $(m+1)$-process normal execution obtained from β' by letting process p_i perform rounds by itself until it returns a value. Since α is an extension of β, α' is an extension of β' and p_i takes the same number of steps in β and β', it follows that $\alpha \overset{p_i}{\not\sim} \alpha'$. By Observation 2.12, $\alpha' \overset{P-\{p_i\}}{\sim} \beta'$. Hence $\alpha \overset{P-\{p_i\}}{\sim} \alpha'$. Since p_i is seen in α, it follows that p_i is seen in α'. Furthermore, $\{y'_0, \ldots, y'_m\} - \{y'_i\} = \{y_0, \ldots, y_m\} - \{y_i\} = \{0, \ldots, m-1\}$, where y'_j is the output of process p_j in execution α', for $j = 0, \ldots, m$. Thus $(\alpha', p_i) \in A_{m+1}$ and $\overset{P-\{p_i\}}{\sim}$ partitions $\{(\alpha, p_i) \in A_{m+1} \mid p_i \text{ is seen in } \alpha\}$ into equivalence classes of size at least two. In fact, each of these equivalence classes has size exactly two and hence, the cardinality of this set is even. To see why an equivalence class cannot have size greater than two, consider any three executions α_1, α_2, and α_3 in the same equivalence class. For $j = 1, 2, 3$, let β_j be the execution obtained from α_j by removing all rounds from the end of α_j in which only p_i participates. Note that p_i still participates in β_j, since p_i is seen in α_j. By Observation 2.12, $\beta_j \overset{P-\{p_i\}}{\sim} \beta_k$ for all $1 \leq j < k \leq 3$. It follows from Lemma 2.13 that $\beta_j \overset{p_i}{\sim} \beta_k$ for some $1 \leq j < k \leq 3$. Then Lemma 2.11 implies that $\beta_j = \beta_k$. This, in turn, implies that $\alpha_j = \alpha_k$, since the algorithm is deterministic and α_1, α_2, and α_3 are m-process normal executions.

Next, suppose that p_i is unseen in α and $i \in \{0, \ldots, m-1\}$. Since $\{y_0, \ldots, y_m\} - \{y_i\} = \{0, \ldots, m-1\}$, there exists $j \in \{0, \ldots, m\} - \{i\}$ such that $y_j = i$. Let α' be obtained from α by deleting all steps by p_i. By Observation 2.12, $\alpha' \overset{P-\{p_i\}}{\sim} \alpha$, so $\alpha \overset{p_j}{\sim} \alpha'$. Let β be obtained from α' by changing the value of x_i from i to another value. Since p_i takes no steps in α', it follows that $\alpha' \overset{p_j}{\sim} \beta$. Hence, process p_j also outputs i in β. However, this violates validity. Thus, there are no pairs $(\alpha, p_i) \in A_{m+1}$ with $i \neq m$ in which p_i is unseen in α.

Finally, suppose that p_m is unseen in α. Let α' be obtained from α by deleting all steps by p_m. By Observation 2.12, $\alpha' \overset{P-\{p_i\}}{\sim} \alpha$, so α' is an m-process normal execution in which all the values $0, \ldots, m-1$ are output, i.e. $\alpha' \in N_m$. Similarly, from any execution $\alpha' \in N_m$, we can construct a pair $(\alpha, p_m) \in A_{m+1}$ such that p_m is unseen in α, by letting process p_m perform rounds by itself until it returns a value, starting after processes $p_0, \ldots, p_{m-1}$ have all produced their output values. Because the algorithm is deterministic and wait-free, α is unique. Thus $\{(\alpha, p_m) \in A_{m+1} \mid p_m \text{ is unseen in } \alpha\}$ is isomorphic to N_m. By the induction hypothesis, $|N_m|$ is odd. Thus, $|A_{m+1}|$ is odd.

Consider any pair $(\alpha, p_i) \in A_{m+1}$ such p_i does not output m in α. By validity, it outputs a value $v \in \{0, \ldots, m-1\}$. Since $\{y_0, \ldots, y_m\} - \{y_i\} = \{0, \ldots, m-1\}$, there is a unique other process, p_j, that decides the same value v in α. Note that $(\alpha, p_j) \in A_{m+1}$ and p_j does not output m in α. Therefore, the set of such pairs can be partitioned into groups of size two. This implies there are an even number of pairs $(\alpha, p_i) \in A_{m+1}$ such that p_i does not output m in α.

Note that $\alpha \in N_{m+1}$ if and only if $y_i = m$ for exactly one $i \in \{0, \ldots, m\}$ and

$\{y_0, \ldots, y_m\} - \{y_i\} = \{0, \ldots, m-1\}$. In turn, this is true if and only if $(\alpha, p_i) \in A_{m+1}$ and p_i outputs m in α. Since $|A_{m+1}|$ is odd and there are an even number of pairs $(\alpha, p_i) \in A_{m+1}$ such that p_i does not output m, it follows that $|N_{m+1}|$ is odd, which proves the inductive step. $\square$

Finally, we can complete the proof of Theorem 2.10. By Lemma 2.14, $|N_n|$ is odd and, hence, nonempty. Thus, there is an n-process normal execution in which all the values $0, \ldots, n-1$ are decided. This violates $(n-1)$-agreement. Therefore, the algorithm does not solve $(n-1)$-set consensus. This is a contradiction. Hence, there is no wait-free algorithm for n processes that solves $(n-1)$-set consensus.

Theorem 2.10 and Lemma 2.13 are from Hagit Attiya and Sergio Rajsbaum's paper *The Combinatorial Structure of Wait-free Solvable Tasks*, SIAM J. Comput., volume 31, 2002, pages 1286–1313. Lemma 2.14 is from *Counting-Based Impossibility Proofs for Renaming and Set Agreement*, by Hagit Attiya and Ami Paz, which appeared at DISC 2012, pages 356–370.