## Contents

1. **Introduction**  
   - 1.1 Decision Tasks ........................................ 8  
   - 1.2 Communication ......................................... 9  
   - 1.3 Failures .................................................. 9  
   - 1.4 Timing .................................................... 10  
     - 1.4.1 Processes and Protocols .......................... 10  
   - 1.5 Chapter Notes ............................................ 12  

2. **Elements of Combinatorial Topology**  
   - 2.1 Basic Concepts ......................................... 13  
   - 2.2 Distributed Computing ................................ 15  
   - 2.3 Chapter Notes ............................................ 16  
   - 2.4 Exercises ................................................ 16  

3. **Manifolds, Impossibility, and Separation**  
   - 3.1 Manifold Complexes .................................... 17  
   - 3.2 Immediate Snapshots ................................... 18  
   - 3.3 Manifold Protocols ..................................... 23  
   - 3.4 Set Agreement ............................................ 25  
   - 3.5 Anonymous Protocols .................................. 28  
   - 3.6 Weak Symmetry-Breaking ............................... 29  
   - 3.7 Anonymous Set Agreement versus Weak Symmetry Breaking ...... 30  
   - 3.8 Chapter Notes ............................................ 35  
   - 3.9 Exercises ................................................ 35  

4. **Connectivity**  
   - 4.1 Consensus and Path-Connectivity ................... 37  
   - 4.2 Consensus in Asynchronous Read-Write Memory ....... 39  
   - 4.3 Set Agreement and Connectivity in Higher Dimensions .. 43
CONTENTS

4.4 Set Agreement and Read-Write memory .................................. 49
  4.4.1 Critical States ................................................. 53
4.5 Chapter Notes .................................................................. 54
4.6 Exercises ....................................................................... 54

5 Colorless Tasks .................................................................. 57
  5.1 Pseudospheres ............................................................. 58
  5.2 Wait-Free Read-Write Memory ......................................... 61
    5.2.1 Read-Write Protocols and Pseudospheres ....................... 61
    5.2.2 Necessary and Sufficient Conditions ............................... 62
  5.3 Read-Write Memory with $k$-Set Agreement ......................... 64
  5.4 Decidability .................................................................... 66
    5.4.1 Loop Agreement ...................................................... 66
    5.4.2 Read-Write Memory .................................................. 68
    5.4.3 Read-Write Memory .................................................. 68
  5.5 Chapter Notes .................................................................. 69
  5.6 Exercises ....................................................................... 70
Part I: Undergraduate Course
Chapter 1

Introduction

The problem of coordinating concurrent processes remains one of the central problems of distributed computing. Coordination problems arise at all scales in distributed and concurrent systems, ranging from synchronizing data access in tightly-coupled multiprocessors, to allocating data paths in networks. Coordination is difficult because modern concurrent and distributed systems are inherently subject to failures and delays: processes may be delayed without warning for a variety of reasons, including interrupts, pre-emption, cache misses, communication delays, or processor crashes. Delays can vary enormously in scale: a cache miss might delay a process for fewer than ten instructions, a page fault for a few million instructions, and operating system pre-emption for hundreds of millions of instructions. At the limit, delays may be indistinguishable from crashes.

In this book, we use techniques adapted from modern Combinatorial Topology to investigate the circumstances under which various coordination tasks can be solved. We show that both the coordination problem to be solved, and any concurrent algorithm that might solve the problem, can both be modeled as combinatorial structures called simplicial complexes. Simplifying somewhat, a particular concurrent algorithm solves a particular coordination problem if and only if there exists a map from one simplicial complex to the other satisfying certain regularity properties.

The appeal of this approach is that it reduces the problem of reasoning about computations that unfold in time to the more familiar problem of reasoning about static combinatorial structures. Equally important, we can call upon a vast literature of results in combinatorial and algebraic topology.

This approach is particularly well-suited for impossibility results. Classical combinatorial topology excels at using topological invariants to prove
that two spaces cannot be homeomorphic, that is, cannot be mapped continuously each to the other. The same techniques can be adapted to show that no concurrent algorithm, in a particular model of computation, can solve a particular coordination problem.

1.1 Decision Tasks

To distill the notion of a distributed computation to its simplest interesting form, we focus on a simple but important class of problems called decision tasks. We are given a set of \( n + 1 \) sequential processes \( P_0, \ldots, P_n \). Each process starts out with a private input value, typically subject to task-specific constraints. The processes communicate for a while, then each process chooses a private output value, also subject to task-specific constraints, and then halts.

Decision tasks are intended to model reactive systems such as databases, file systems, or flight control systems. An input value represents information entering the system from the outside world, such as a character typed at a keyboard, a message from another computer, or a signal from a sensor. An output value models an effect on the outside world, such as an irrevocable decision to commit a transaction, to dispense cash, or to launch a missile.

Perhaps the simplest example of a decision task is consensus. Each process starts with an input value and chooses an output value. All output values must agree, and each output value must have been some process’s input value. If the input values are Boolean, the task is called binary consensus. The consensus task was originally studied as an idealization of the transaction commitment problem, in which a number of database sites must agree on whether to commit or abort a distributed transaction. For short, we call the consensus task for \( n \) processes \( n \)-consensus.

A natural generalization of consensus is \( k \)-set agreement. Like consensus, each process’s output value must be some process’s input value. Unlike consensus, which requires that all processes agree, \( k \)-set agreement requires that no more than \( k \) distinct output values be chosen. Consensus is 1-set agreement.

In the renaming task, processes are issued unique input names from a large name space, and must choose unique output names taken from a smaller name space. To rule out trivial solutions, protocols must be anonymous, meaning that the value any process chooses depends only on its input value and how its steps are interleaved with the steps of the other processes.

In the weak symmetry-breaking task, processes are required to sort them-
selves into two groups, A and B. If all \( n+1 \) processes participate, then each group must have at least one member. If fewer participate, then any distribution is correct. Like renaming, weak symmetry-breaking is required to be anonymous.

### 1.2 Communication

Perhaps the oldest communication model is *message-passing*. Each processor sends messages to other processes, receives messages sent to it by the other processes in that round, performs some internal computation, and changes state. We assume that processes are following a full-information protocol, which means that each processor sends its entire local state to every processor in every round.

In shared-memory models, processes communicate by applying operations to objects in shared memory. The simplest kind of shared-memory object is *read-write* memory, where the processes share an array of memory locations. There are many models for read-write memory. Memory variables may encompass a single bit, or an arbitrary number of bits, and variables can be single-writer or multi-writer. Fortunately, all such models are equivalent in the sense that any one can be implemented in a wait-free manner from any other. From these variables, in turn, one can implement an *atomic snapshot memory*: an array where each process writes its own variables and can atomically reads ("snapshot") the entire memory.

In models that more accurately reflect today’s multiprocessors, we can augment read-write memory with shared objects such as stacks, queues, test-and-set variables, or objects of arbitrary abstract type. In particular, it is instructive to augment read-write memory with synchronization primitives that cannot be implemented in read-write memory itself.

### 1.3 Failures

The theory of concurrent computing is largely the theory of what can be accomplished in the presence of timing uncertainty and failures. In the most basic model, the goal is to provide *wait-free* algorithms that solve particular tasks when any number of processes may fail. In some timing models, such failures can eventually be detected, while in other models, a failed process is indistinguishable from a slow process.

The wait-free failure model is very demanding, and sometimes we are willing to settle for less. A *t-resilient* algorithm is one that works correctly
when the number of faulty processes does not exceed a value \( t \). A wait-free algorithm for \( n + 1 \) processes is \( n \)-resilient.

A limitation of these classical models is that they implicitly assume that processes fail independently. In a distributed system, however, failures may be correlated for processes running on the same node, in the same network partition, or managed by the same provider. In a multiprocessor, failures may be correlated for processes running on the same core, the same processor, or the same card. To model these situations, it is natural to introduce the notion of an adversary scheduler that can cause certain subsets of processes to fail.

There are several ways to characterize adversaries. The most straightforward is to enumerate the faulty sets: all sets of processes that can fail in some execution. We find it more convenient to use the dual notions of cores and survivor sets, as proposed by Junqueira and Marzullo [13, 14]. A core is a minimal set of processes that will not all fail in any execution, while a survivor set is a minimal set of processes that might not fail in some execution. For the wait-free adversary, the entire set of processes is the only core, and for the \( t \)-faulty adversary, any set of \( t + 1 \) processes is a core. For the adversary given by faulty sets \( \{\emptyset, P, QR\} \), the cores are \( PQ \) and \( PR \), and the survivor sets are \( P \) and \( QR \).

### 1.4 Timing

In synchronous timing models, all non-faulty processes take steps at the same time. In synchronous models, it is usually possible to detect process failures. In asynchronous models, there is no bound on process step time. A failed process cannot be distinguished from a slow process. In semi-synchronous models, there is an upper bound on how long it takes for a non-faulty process to communicate with another. In this model, a failed process can be detected following a (usually length) timeout.

#### 1.4.1 Processes and Protocols

A system is a collection of state machines called processes, together with an environment such as shared read-write memory, other shared objects, or message queues. Each process executes a finite protocol. It starts in an initial state, and takes steps until it either fails, meaning it halts and takes no additional steps, or it halts, usually because it has completed the protocol. Processes that have failed are said to be faulty. Each step typically involves communicating with other process’s either through shared objects or
message-passing. Processes are deterministic: each transition is determined by the process’s current state and the state of the environment.

Steps of different processes may be interleaved. This interleaving is typically non-deterministic, although the timing properties of the model can restrict the set of possible interleavings.

The protocol state is given the set of states of non-faulty processes and the state of the environment. An execution is a sequence of process state transitions. An execution $e$ carries the system from one state $s$ to another state $s'$. Two executions are equivalent if (1) they leave the system in the same final system state, and (2) every non-faulty process executes the same steps in both. Equivalent executions do not need to start from the same state, nor do faulty processes need to execute the same steps before they fail.

Here is a very high-level description of a generic protocol. We omit details for now because we want a description that applies to many different models of computation. We consider full-information protocols, where a process’s local state consists of its input value, and a record of all communications with other processes. Each process repeatedly (1) communicates its current local state to the others, (2) collects local states from the others, and (3) updates its local state to reflect the information collect in Step (2). The protocol terminates in a final state, where it chooses an output value based on the current local state.

For example, consider a one-round, two-process protocol, in which processes share a two-element memory $m$. Process $A$ writes $m[0]$ and reads $m[1]$, while $B$ writes $m[1]$ and reads $m[0]$. In a one-round protocol (Fig. 1.4.1), each process writes its input to its memory element, then reads the other’s, and appends that value to its state.
1.5 Chapter Notes

The foundation paper in this area is by Fischer, Lynch, and Paterson [cite], who showed there is a simple coordination problem that cannot be solved in a message-passing system if even one process may fail, either by halting, or by being arbitrary slow.

A first step toward a systematic characterization of asynchronous computability was taken in 1988 by Biran, Moran, and Zaks [2] who gave a graph-theoretic characterization of the tasks that could be solved by a message-passing system in the presence of a single failure.
Bibliography


