Chapter 1

Introduction

Distributed systems are characterized by their structure: a typical distributed system will consist of some large number of interacting devices that each run their own programs but that are affected by receiving messages, or observing shared-memory updates or the states of other devices. Examples of distributed systems range from simple systems in which a single client talks to a single server to huge amorphous networks like the Internet as a whole.

As distributed systems get larger, it becomes harder and harder to predict or even understand their behavior. Part of the reason for this is that we as programmers have not yet developed a standardized set of tools for managing complexity (like subroutines or objects with narrow interfaces, or even simple structured programming mechanisms like loops or if/then statements) as are found in sequential programming. Part of the reason is that large distributed systems bring with them large amounts of inherent **nondeterminism**—unpredictable events like delays in message arrivals, the sudden failure of components, or in extreme cases the nefarious actions of faulty or malicious machines opposed to the goals of the system as a whole. Because of the unpredictability and scale of large distributed systems, it can often be difficult to test or simulate them adequately. Thus there is a need for theoretical tools that allow us to prove properties of these systems that will let us use them with confidence.

The first task of any theory of distributed systems is modeling: defining a mathematical structure that abstracts out all relevant properties of a large distributed system. There are many foundational models for distributed systems, but for this class we will follow [AW04] and use simple automaton-based models.

What this means is that we model each process in the system as an automaton that has some sort of local **state**, and model local computation as a transition rule that tells us how to update this state in response to various **events**. Depending on what kinds of system we are modeling, these events might correspond to local computation, to delivery of a message by a network, carrying out some operation on a shared memory, or even something like a chemical reaction between two molecules. The transition rule for a system specifies how the states of all processes involved in the event are updated, based on their previous states. We can think of the transition rule as an arbitrary mathematical function (or relation if the processes are nondeterministic); this corresponds in programming terms to implementing local computation by processes as a gigantic table lookup.

Obviously this is not how we program systems in practice. But what this approach does is allow us to abstract away completely from how individual processes work, and emphasize how all of the processes interact with each other. This can lead to odd results: for example, it's perfectly consistent with this model for some process to be able to solve the halting problem, or carry out arbitrarily complex calculations between receiving a message and sending its response. A partial justification for this assumption is that in practice, the multi-millisecond latencies in even reasonably fast networks are eons in terms of local computation. And as with any assumption, we can always modify it if it gets us into trouble.

1.1 Models

The global state consisting of all process states is called a **configuration**, and we think of the system as a whole as passing from one global state or **configuration** to another in response to each event. When this occurs the processes participating in the event update their states, and the other processes do nothing. This does not model concurrency directly; instead, we interleave potentially concurrent events in some arbitrary way. The advantage of this interleaving approach is that it gives us essentially the same behavior as we would get if we modeled simultaneous events explicitly, but still allows us to consider only one event at a time and use induction to prove various properties of the sequence of configurations we might reach.

We will often use lowercase Greek letters for individual events or sequences of events. Configurations are typically written as capital Latin letters (often C). An **execution** of a schedule is an alternating sequence of configurations and events $C_0\sigma_0C_1\sigma_1C_2\ldots$, where C_{i+1} is the configuration that results from

applying event σ_i to configuration C. A **schedule** is the sequence of events $\sigma_0\sigma_1\ldots$ from some execution. We say that an event σ is **enabled** in C if this event can be carried out in C; an example would be that the event that we deliver a particular message in a message-passing system is enabled only if that message has been sent and not yet delivered. When σ is enabled in C, it is sometime convenient to write $C\sigma$ for the configuration that results from applying σ to C.

What events are available, and what effects they have, will depend on what kind of model we are considering. We may also have additional constraints on what kinds of schedules are **admissible**, which restricts the schedules under considerations to those that have certain desirable properties (say, every message that is sent is eventually delivered). There are many models in the distributed computing literature, which can be divided into a handful of broad categories:

- **Message passing** models (which we will cover in Part I) correspond to systems where processes communicate by sending messages through a network. In **synchronous message-passing**, every process sends out messages at time t that are delivered at time $t+1$, at which point more messages are sent out that are delivered at time $t+2$, and so on: the whole system runs in lockstep, marching forward in perfect synchrony.[1] Such systems are difficult to build when the components become too numerous or too widely dispersed, but they are often easier to analyze than **asynchronous** systems, where messages are only delivered eventually after some unknown delay. Variants on these models include **semi-synchronous** systems, where message delays are unpredictable but bounded, and various sorts of timed systems. Further variations come from restricting which processes can communicate with which others, by allowing various sorts of failures: **crash failures** that stop a process dead, **Byzantine failures** that turn a process evil, or **omission failures** that drop messages in transit. Or—on the helpful side—we may supply additional tools like **failure detectors** (Chapter 13) or **randomization** (Chapter 23).

- **Shared-memory** models (Part II) correspond to systems where processes communicate by executing operations on shared objects

[1] In an interleaving model, these apparently simultaneous events are still recorded one at a time. What makes the system synchronous is that we demand that, in any admissible schedule, all n events for time t occur as a sequential block, followed by all n events for time $t+1$, and so on.

In the simplest case, the objects are simple memory cells supporting read and write operations; these are called (**atomic registers**. But in general, the objects could be more complex hardware primitives like **compare-and-swap** (§18.1.3), **load-linked/store-conditional** (§18.1.3), **atomic queues**, or even more exotic objects from the seldom-visited theoretical depths. Practical shared-memory systems may be implemented as **distributed shared-memory** (Chapter 16) on top of a message-passing system in various ways.

Like message-passing systems, shared-memory systems must also deal with issues of asynchrony and failures, both in the processes and in the shared objects.

Realistic shared-memory systems have additional complications, in that modern CPUs allow out-of-order execution in the absence of special (and expensive) operations called **fences** or **memory barriers**.[AG95] We will effectively be assuming that our shared-memory code is liberally sprinkled with these operations so that nothing surprising happens, but this is not always true of real production code, and indeed there is work in the theory of distributed computing literature on algorithms that don't require unlimited use of memory barriers.

- A third family of models has no communication mechanism independent of the processes. Instead, the processes may directly observe the states of other processes. These models are used in analyzing **self-stabilization**, for some **biologically inspired systems**, and for computation by **population protocols** or **chemical reaction networks**. We will discuss some of this work in Part III.

- Other specialized models emphasize particular details of distributed systems, such as the labeled-graph models used for analyzing routing or the topological models used to represent some specialized agreement problems (see Chapter 28.

We'll see many of these at some point in this course, and examine which of them can simulate each other under various conditions.

1.2 Properties

Properties we might want to prove about a system include:

- **Safety** properties, of the form "nothing bad ever happens" or more precisely "there are no bad reachable states of the system." These

include things like "at most one of the traffic lights at the intersection of Busy and Main is ever green." Such properties are typically proved using **invariants**, properties of the state of the system that are true initially and that are preserved by all transitions; this is essentially a disguised induction proof.

- **Liveness** properties, of the form "something good eventually happens." An example might be "my email is eventually either delivered or returned to me." These are not properties of particular states (I might unhappily await the eventual delivery of my email for decades without violating the liveness property just described), but of executions, where the property must hold starting at some finite time. Liveness properties are generally proved either from other liveness properties (e.g., "all messages in this message-passing system are eventually delivered") or from a combination of such properties and some sort of timer argument where some progress metric improves with every transition and guarantees the desirable state when it reaches some bound (also a disguised induction proof).

- **Fairness** properties are a strong kind of liveness property of the form "something good eventually happens to everybody." Such properties exclude **starvation**, a situation where most of the kids are happily chowing down at the orphanage ("some kid eventually eats something" is a liveness property) but poor Oliver Twist is dying for lack of gruel in the corner.

- **Simulations** show how to build one kind of system from another, such as a reliable message-passing system built on top of an unreliable system (TCP [Pos81]), a shared-memory system built on top of a message-passing system (distributed shared memory—see Chapter 16), or a synchronous system build on top of an asynchronous system (**synchronizers**—see Chapter 7).

- **Impossibility results** describe things we can't do. For example, the classic **Two Generals** impossibility result (Chapter 8) says that it's impossible to guarantee agreement between two processes across an unreliable message-passing channel if even a single message can be lost. Other results characterize what problems can be solved if various fractions of the processes are unreliable, or if asynchrony makes timing assumptions impossible. These results, and similar lower bounds that describe things we can't do quickly, include some of the most technically

sophisticated results in distributed computing. They stand in contrast to the situation with sequential computing, where the reliability and predictability of the underlying hardware makes proving lower bounds extremely difficult.

There are some basic proof techniques that we will see over and over again in distributed computing.

For **lower bound** and **impossibility** proofs, the main tool is the **indistinguishability** argument. Here we construct two (or more) executions in which some process has the same input and thus behaves the same way, regardless of what algorithm it is running. This exploitation of process's ignorance is what makes impossibility results possible in distributed computing despite being notoriously difficult in most areas of computer science.[2]

For **safety properties**, statements that some bad outcome never occurs, the main proof technique is to construct an **invariant**. An invariant is essentially an induction hypothesis on reachable configurations of the system; an invariant proof shows that the invariant holds in all initial configurations, and that if it holds in some configuration, it holds in any configuration that is reachable in one step.

Induction is also useful for proving **termination** and **liveness** properties, statements that some good outcome occurs after a bounded amount of time. Here we typically structure the induction hypothesis as a **progress measure**, showing that some sort of partial progress holds by a particular time, with the full guarantee implied after the time bound is reached.

[2] An exception might be lower bounds for data structures, which also rely on a process's ignorance.