Chapter 2

Model

Message passing models simulate networks. Because any interaction between physically separated processors requires transmitting information from one place to another, all distributed systems are, at a low enough level, message-passing systems. We start by defining a formal model of these systems.

2.1 Basic message-passing model

We have a collection of n **processes** $p_1 \ldots p_2$, each of which has a **state** consisting of a state from from state set Q_i. We think of these processes as nodes in a directed **communication graph** or **network**. The edges in this graph are a collection of point-to-point **channels** or **buffers** b_{ij}, one for each pair of adjacent processes i and j, representing messages that have been sent but that have not yet been delivered. Implicit in this definition is that messages are point-to-point, with a single sender and recipient: if you want broadcast, you have to build it yourself.

A **configuration** of the system consists of a vector of states, one for each process and channel. The configuration of the system is updated by an **event**, in which (1) zero or more messages in channels b_{ij} are delivered to process p_j, removing them from b_{ij}; (2) p_j updates its state in response; and (3) zero or more messages are added by p_j to outgoing channels b_{ji}. We generally think of these events as **delivery events** when at least one message is delivered, and as **computation events** when none are. An **execution segment** is a sequence of alternating configurations and events $C_0, \phi_1, C_1, \phi_2, \ldots$, in which each triple $C_i \phi_{i+1} C_{i+1}$ is consistent with the transition rules for the event ϕ_{i+1}, and the last element of the sequence (if any) is a configuration. If the first configuration C_0 is an **initial configuration** of the system, we have an

execution. A **schedule** is an execution with the configurations removed.

2.1.1 Formal details

Let P be the set of processes, Q the set of process states, and M the set of possible messages.

Each process p_i has a state $\mathsf{state}_i \in Q$. Each channel b_{ij} has a state $\mathsf{buffer}_{ij} \in \mathcal{P}(M)$. We assume each process has a **transition function** $\delta : Q \times \mathcal{P}(M) \to QU\,\mathcal{P}\,P \times M$ that maps tuples consisting of a state and a set of incoming messages a new state and a set of recipients and messages to be sent. An important feature of the transition function is that the process's behavior can't depend on which of its previous messages have been delivered or not. A delivery event $\mathsf{del}(i, A)$, where $A = \{(j_k, m_k)\}$) removes each message m_k from b_{ji}, updates state_i according to $\delta(\mathsf{state}_i, A)$, and adds the outgoing messages specified to $\delta(\mathsf{state}_i, A)$ to the appropriate channels. A computation event $\mathsf{comp}(i)$ does the same thing, except that it applies $\delta(\mathsf{state}_j, \emptyset)$.

Some implicit features in this definition:

- A process can't tell when its outgoing messages are delivered, because the channel states aren't available as input to δ.
- Processes are **deterministic**: The next action of each process depends only on its current state, and not on extrinsic variables like the phase of the moon, coin-flips, etc. We may wish to relax this condition later by allowing coin-flips; to do so, we will need to extend the model to incorporate probabilities.
- It is possible to determine the accessible state of a process by looking only at events that involve that process. Specifically, given a schedule S, define the **restriction** $S|i$ to be the subsequence consisting of all $\mathsf{comp}(i)$ and $\mathsf{del}(i, A)$ events (ranging over all possible A). Since these are the only events that affect the state of i, and only the state of i is needed to apply the transition function, we can compute the state of i looking only at $S|i$. In particular, this means that i will have the same accessible state after any two schedules S and S' where $S|i = S'|i$, and thus will take the same actions in both schedules. This is the basis for **indistinguishability proofs** (§8.2), a central technique in obtaining lower bounds and impossibility results.

Attiya and Welch [AW04] use a different model in which communication channels are not modeled separately from processes, but instead are baked

into processes as outbuf and inbuf variables. This leads to some oddities like having to distinguish the accessible state of a process (which excludes the outbufs) from the full state (which doesn't). Our approach is close to that of Lynch [Lyn96], in that we have separate automata representing processes and communication channels. But since the resulting model produces essentially the same executions, the exact details don't really matter.[1]

2.1.2 Network structure

It may be the case that not all processes can communicate directly; if so, we impose a network structure in the form of a directed graph, where i can send a message to j if and only if there is an edge from i to j in the graph. Typically we assume that each process knows the identity of all its neighbors.

For some problems (e.g., in peer-to-peer systems or other **overlay networks**) it may be natural to assume that there is a fully-connected underlying network but that we have a dynamic network on top of it, where processes can only send to other processes that they have obtained the addresses of in some way.

2.2 Asynchronous systems

In an **asynchronous** model, only minimal restrictions are placed on when messages are delivered and when local computation occurs. A schedule is said to be **admissible** if (a) there are infinitely many computation steps for each process, and (b) every message is eventually delivered. (These are **fairness** conditions.) The first condition (a) assumes that processes do not explicitly terminate, which is the assumption used in [AW04]; an alternative, which we will use when convenient, is to assume that every process either has infinitely many computation steps or reaches an explicit halting state.

[1] The late 1970s and early 1980s saw a lot of research on finding the "right" definition of a distributed system, and some of the disputes from that era were hard fought. But in the end, all the various proposed models turned out to be more or less equivalent, which is not surprising since the authors were ultimately trying to represent the same intuitive understanding of these systems. So most distributed computing papers now just use some phrasing like "we consider the standard model of an asynchronous message-passing system" and leave to the reader to assume that this standard model is their favorite one.

An example of this trick in action is that you will never see $\mathsf{del}(i, A)$ or $\mathsf{comp}(i)$ again after you finish reading this footnote.

2.2.1 Example: client-server computing

Almost every distributed system in practical use is based on **client-server** interactions. Here one process, the **client**, sends a **request** to a second process, the **server**, which in turn sends back a **response**. We can model this interaction using our asynchronous message-passing model by describing what the transition functions for the client and the server look like: see Algorithms 2.1 and 2.2.

```
1 initially do
2     send request to server
```

Algorithm 2.1: Client-server computation: client code

```
1 upon receiving request do
2     send response to client
```

Algorithm 2.2: Client-server computation: server code

The interpretation of Algorithm 2.1 is that the client sends request (by adding it to its outbuf) in its very first computation event (after which it does nothing). The interpretation of Algorithm 2.2 is that in any computation event where the server observes request in its inbuf, it sends response.

We want to claim that the client eventually receives response in any admissible execution. To prove this, observe that:

1. After finitely many steps, the client carries out a computation event. This computation event puts request in its outbuf.

2. After finitely many more steps, a delivery event occurs that delivers request to the server. This causes the server to send response.

3. After finitely many more steps, a delivery event delivers response to the client, causing it to process response (and do nothing, given that we haven't included any code to handle this response).

Each step of the proof is justified by the constraints on admissible executions. If we could run for infinitely many steps without a particular process doing a computation event or a particular message being delivered, we'd violate those constraints.

Most of the time we will not attempt to prove the correctness of a protocol at quite this level of tedious detail. But if you are only interested in

distributed algorithms that people actually use, you have now seen a proof of correctness for 99.9% of them, and do not need to read any further.

2.3 Synchronous systems

A **synchronous message-passing** system is exactly like an asynchronous system, except we insist that the schedule consists of alternating phases in which (a) every process executes a computation step, and (b) all messages are delivered while none are sent.[2] The combination of a computation phase and a delivery phase is called a **round**. Synchronous systems are effectively those in which all processes execute in lock-step, and there is no timing uncertainty. This makes protocols much easier to design, but makes them less resistant to real-world timing oddities. Sometimes this can be dealt with by applying a **synchronizer** (Chapter 7), which transforms synchronous protocols into asynchronous protocols at a small cost in complexity.

2.4 Drawing message-passing executions

Though formally we can describe an execution in a message-passing system as a long list of events, this doesn't help much with visualizing the underlying communication pattern. So it can sometimes be helpful to use a more visual representation of a message-passing execution that shows how information flows through the system.

A typical example is given in Figure 2.1. In this picture, time flows from left to right, and each process is represented by a horizontal line. This convention reflects the fact that processes have memory, so any information available to a process at some time t is also available at all times $t' \geq t$. Events are represented by marked points on these lines, and messages are represented by diagonal lines between events. The resulting picture looks like a collection of **world lines** as used in physics to illustrate the path taken by various objects through spacetime.

Pictures like Figure 2.1 can be helpful for illustrating the various constraints we might put on message delivery. In Figure 2.1, the system is completely asynchronous: messages can be delivered in any order, even if sent between the same processes. If we run the same protocol under stronger assumptions, we will get different communication patterns.

[2] Formally, the delivery phase consists of n separate delivery events, in any order, that between them clean out all the channels.

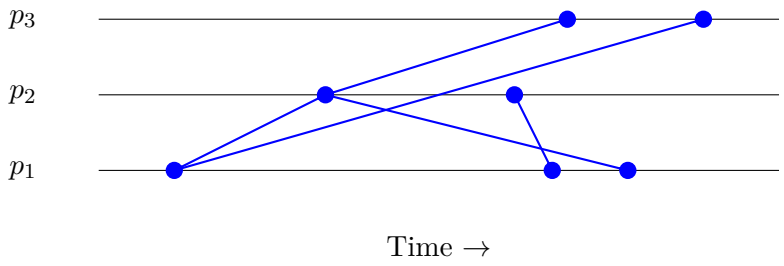

Figure 2.1: Asynchronous message-passing execution. Time flows left-to-right. Horizontal lines represent processes. Nodes represent events. Diagonal edges between events represent messages. In this execution, p_1 executes a computation event that sends messages to p_2 and p_3. When p_2 receives this message, it sends messages to p_1 and p_3. Later, p_2 executes a computation event that sends a second message to p_1. Because the system is asynchronous, there is no guarantee that messages arrive in the same order they are sent.

For example, Figure 2.2 shows an execution that is still asynchronous but that assumes FIFO (first-in first-out) channels. A **FIFO channel** from some process p to another process q guarantees that q receives messages in the same order that p sends them (this can be simulated by a non-FIFO channel by adding a **sequence number** to each message, and queuing messages at the receiver until all previous messages have been processed).

If we go as far as to assume synchrony, we get the execution in Figure 2.3. Now all messages take exactly one time unit to arrive, and computation events follow each other in lockstep.

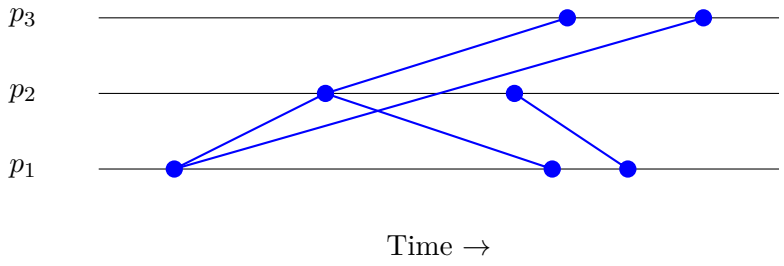

Figure 2.2: Asynchronous message-passing execution with FIFO channels. Multiple messages from one process to another are now guaranteed to be delivered in the order they are sent.

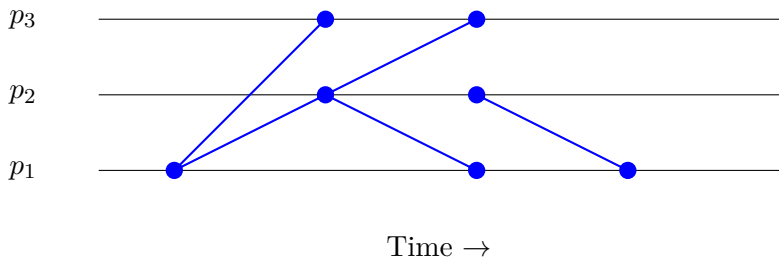

Figure 2.3: Synchronous message-passing execution. All messages are now delivered in exactly one time unit, and computation events occur exactly one time unit after the previous event.

2.5 Complexity measures

There is no explicit notion of time in the asynchronous model, but we can define a time measure by adopting the rule that every message is delivered and processed at most 1 time unit after it is sent. Formally, we assign time 0 to the first event, and assign the largest time we can to each subsequent event, subject to the constraints that (a) no event is assigned a larger time than any later event; (b) if a message m from i to j is created by an event at time t, then the time for the delivery of m from i to j is no greater than $j+1$, and (c) any computation step is assigned a time no later than the previous event at the same process (or 0 if the process has no previous events). This is consistent with an assumption that message propagation takes at most 1 time unit and that local computation takes 0 time units.

Another way to look at this is that it is a definition of a time unit in terms of maximum message delay together with an assumption that message delays dominate the cost of the computation. This last assumption is pretty much always true for real-world networks with any non-trivial physical separation between components, thanks to speed of light limitations.

An example of an execution annotated with times in this way is given in Figure 2.4.

The **time complexity** of a protocol (that terminates) is the time of the last event at any process.

Note that looking at **step complexity**, the number of computation events involving either a particular process (**individual step complexity**) or all processes (**total step complexity**) is not useful in the asynchronous model, because a process may be scheduled to carry out arbitrarily many computation steps without any of its incoming or outgoing messages being

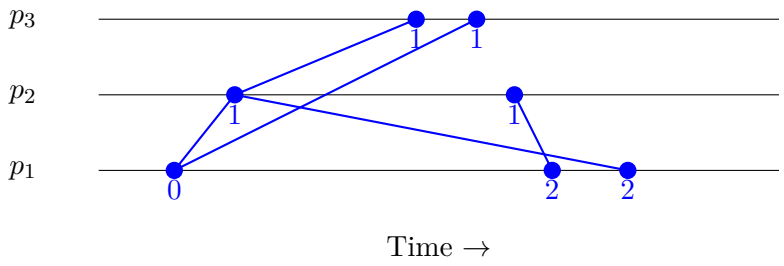

Figure 2.4: Asynchronous message-passing execution with times.

delivered, which probably means that it won't be making any progress. These complexity measures will be more useful when we look at shared-memory models (Part II).

For a protocol that terminates, the **message complexity** is the total number of messages sent. We can also look at message length in bits, total bits sent, etc., if these are useful for distinguishing our new improved protocol from last year's model.

For synchronous systems, time complexity becomes just the number of rounds until a protocol finishes. Message complexity is still only loosely connected to time complexity; for example, there are synchronous **leader election** (Chapter 5) algorithms that, by virtue of grossly abusing the synchrony assumption, have unbounded time complexity but very low message complexity.