Chapter 11
Message Passing and Distributed Programming

Absence of shared memory is the defining characteristic of a distributed system. Processes in such a system coordinate by reading and writing shared objects called communications channels. Thus, concurrent programs for distributed systems, or distributed programs, are fundamentally no different from other concurrent programs: all use shared objects of one sort or another for synchronization and communication.

However, the shared objects (communications channels) in a distributed system can be read and written only in restricted ways. These restrictions make designing distributed programs more difficult than designing shared-memory concurrent programs. Overcoming this difficulty is a subject of this chapter.

Two common forms of message passing are described: asynchronous message-passing, which is frequently supported by an operating system, and synchronous message-passing, which more often appears in concurrent programming notations. Other message-passing primitives are likely to be derivatives of these. We then turn to techniques for deriving programs that use message-passing. We illustrate the techniques by applying them to two classical distributed programming problems.

11.1 Asynchronous Message-Passing

With asynchronous message-passing, a send statement

\[(11.1) \text{ send } expr \text{ on } C\]

is an unconditional atomic action. To execute (11.1), \(expr\) is evaluated and a message with that value is sent on channel \(C\).
A receive statement

(11.2) receive \( m \) from \( C \)

blocks until some message is eligible for receipt on channel \( C \); an eligible message is then \textit{received} by assigning it to \( m \). The \textit{eligible} messages on a channel are those that have been sent on the channel but not yet received.

Various guarantees are typically made about whether messages sent must become eligible and the order in which eligible messages may be received. These guarantees are usually associated with the channel.

- With a \textit{reliable channel}, every message sent eventually becomes eligible for receipt.
- With an \textit{unreliable channel}, some messages may never become eligible for receipt—the lost messages might be victims of hardware failures, noise bursts, or buffer overruns.
- With a \textit{virtual circuit}, messages are received in the order sent.
- With a \textit{datagram service}, no order is imposed on the receipt of messages.

\textit{Axiomatization of Asynchronous Message-Passing}

To axiomatize send statement (11.1) and receive statement (11.2), we translate them into atomic actions about which we already know how to reason. We concentrate here on reliable virtual circuits, since they are so popular.\(^1\) A channel \( C \) that implements a reliable virtual circuit is modeled by an implicit shared variable \( C \) that contains the sequence of eligible messages in the order they have been sent. Initially, shared variable \( C \) equals \( \varepsilon \).

(11.3) \textbf{Virtual Circuit Initialization Axiom:} For a program \( S \) with a reliable virtual circuit \( C \):

\[
Init_S \Rightarrow C = \varepsilon
\]

Send statement (11.1) has the same effect as assignment statement\(^2\)

(11.4) \( C := C \ \textit{expr} \)

and receive statement (11.2) is equivalent to atomic statement

(11.5) \( \langle \text{if } C \neq \varepsilon \rightarrow m, \ C := C[0], \ C[1..] \ \text{fi} \rangle \).

---

\(^1\)See exercise 11.6 for an axiomatization of unreliable virtual circuits and exercise 11.7 for an axiomatization of an unreliable datagram service.

\(^2\)Recall, expression \( C \ \textit{expr} \) evaluates to the result of appending the value of \( \textit{expr} \) to \( C \).