13 Concurrency

13.5 Message Passing

While shared-memory concurrent programming is common on small-scale multicore and multiprocessor machines, most programs that run on clusters, supercomputers, or geographically distributed machines are currently based on messages. In Sections C-13.5.1 through C-13.5.3 we consider three principal issues in message-based computing: naming, sending, and receiving. In Section C-13.5.4 we look more closely at one particular combination of send and receive semantics, namely remote procedure call. Most of our examples will be drawn from the Ada, Erlang, and Go programming languages, the Java network library, and the MPI library package.

13.5.1 Naming Communication Partners

EXAMPLE 13.53
Naming processes, ports, and entries

To send or receive a message, one must generally specify where to send it to, or where to receive it from: communication partners need names for (or references to) one another. Names may refer directly to a thread or process. Alternatively, they may refer to an *entry* or *port* of a module, or to some sort of *socket* or *channel* abstraction. We illustrate these options in Figure C-13.21. ■

The first naming option—addressing messages to processes—appears in Hoare's original CSP (Communicating Sequential Processes) [Hoa78], an influential proposal for simple communication mechanisms. It also appears in Erlang and in MPI. Each MPI process has a unique `id` (an integer), and each `send` or `receive` operation specifies the `id` of the communication partner. MPI implementations are required to be reentrant; a process can safely be divided into multiple threads, each of which can send or receive messages on the process's behalf.

EXAMPLE 13.54
`entry` calls in Ada

The second naming option—addressing messages to ports—appears in Ada. An Ada *entry call* of the form `t.foo(args)` sends a message to the `entry` named `foo` in task (thread) `t` (`t` may be either a task name or the name of a variable

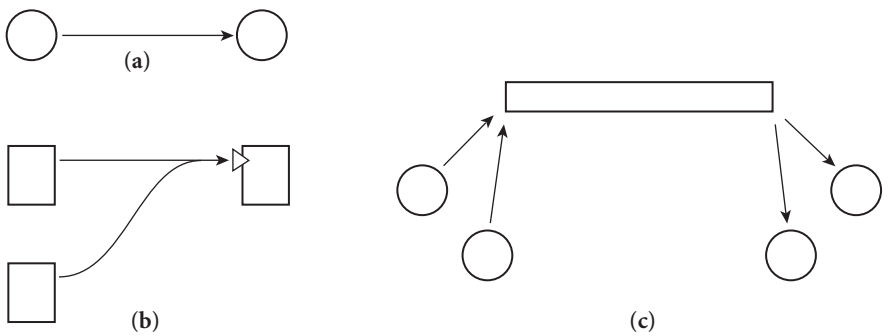

Figure 13.21 **Three common schemes to name communication partners.** In (a), processes name each other explicitly. In (b), senders name an *input port* of a receiver. The port may be called an *entry* or an *operation*. The receiver is typically a module with one or more threads inside. In (c), senders and receivers both name an independent *channel* abstraction, which may be called a *connection* or a *mailbox*.

whose value is a pointer to a task). As we saw in Section 13.2.3, an Ada task resembles a module; its entries resemble subroutine headers nested directly inside the task. A task receives a message that has been sent to one of its entries by executing an `accept` statement (to be discussed in Section C-13.5.3). Every `entry` belongs to exactly one task; all messages sent to the same `entry` must be received by that one task. ■

EXAMPLE 13.55
Channels in Go

The third naming option—addressing messages to channels—appears in Go and Occam. (Though their concurrency features are loosely based on CSP, both Go and Occam differ from Hoare's proposal in several concrete ways, including the use of channels.) Channel declarations in Go are supported with the `chan` type constructor:

```
var c1 chan int
```

This code declares `c1` to be an (initially `nil`) reference to a channel. A channel value can be created with the built-in function `make`:

```
c1 = make(chan int)
```

Typically the declaration and initialization appear together:

```
var c1 = make(chan int)
```

Here Go infers the type of `c1` from the initialization expression.

To send a message on a channel, a thread uses the binary "arrow" operator `<-` with a channel variable on the left and a message on the right:

```
c1 <- 3
```

To receive, it uses <- as a unary operator, with the channel on the right:

```
my_int = <-c1
```

To indicate that no further messages will be forthcoming, a thread can *close* a channel. A receiving thread can check for this possibility by assigning a receive expression into a pair, the second element of which is a Boolean:

```
my_int, ok = <-c1
if (ok) {
    // use my_int ...
```

EXAMPLE 13.56
Remote invocation in Go

For the common idiom in which a server thread is willing to accept requests from any of many possible client threads, each request message can include a reference to the channel on which to send a response:

```
type request struct {
    name string
    reply_to chan string
}
...
// Assume a server thread is listening on chan 'service'
...
var c = make(chan string, 1)    // create channel for response
service <- request{"Alice", c}  // send look-up request for Alice
println(<-c)                    // receive response on c
```

Internet Messaging

Java's standard `java.net` library provides two styles of message passing, corresponding to the UDP and TCP Internet protocols. UDP is the simpler of the two. It is a *datagram* protocol, meaning that each message is sent to its destination independently and unreliably. The network software will attempt to deliver it, but makes no guarantees. Moreover two messages sent to the same destination (assuming they both arrive) may arrive in either order. UDP messages use port-based naming (Figure C-13.21b): each message is sent to a specific *Internet address* and *port number*.[1] The TCP protocol also uses port-based naming, but only for the purpose of establishing *connections* (Figure C-13.21c), which it then uses for all subsequent communication. Connections deliver messages reliably and in order.

---

1 Every publicly visible machine on the Internet has its own unique address. Though a transition to 128-bit addresses has been underway for some time, as of 2008 most addresses are still 32-bit integers, usually printed as four period-separated fields (e.g., 192.5.54.209). Internet name servers translate symbolic names (e.g., `gate.cs.rochester.edu`) into numeric addresses. Port numbers are also integers, but are local to a given Internet address. Ports 1024 through 4999 are generally available for application programs; larger and smaller numbers are reserved for servers.

EXAMPLE 13.57
Datagram messages in Java

To send or receive UDP messages, a Java thread must create a *datagram socket*:

```
DatagramSocket my_socket = new DatagramSocket(port_id);
```

The parameter of the `DatagramSocket` constructor is optional; if it is not specified, the operating system will choose an available port. Typically servers specify a port and clients allow the OS to choose. To send a UDP message, a thread says

```
DatagramPacket my_msg = new DatagramPacket(buf, len, addr, port);
...  // initialize message
my_socket.send(my_msg);
```

The parameters to the `DatagramPacket` constructor specify an array of bytes `buf`, its length `len`, and the Internet address and port of the receiver. Receiving is symmetric:

```
my_socket.receive(my_msg);
...  // parse content of my_msg
```

■

EXAMPLE 13.58
Connection-based messages in Java

For TCP communication, a server typically "listens" on a port to which clients send requests to establish a connection:

```
ServerSocket my_server_socket = new ServerSocket(port_id);
Socket client_connection = my_server_socket.accept();
```

The `accept` operation blocks until the server receives a connection request from a client. Typically a server will immediately fork a new thread to communicate with the client; the parent thread loops back to wait for another connection with `accept`.

A client sends a connection request by passing the server's symbolic name and port number to the `Socket` constructor:

```
Socket server_connection = new Socket(host_name, port_id);
```

Once a connection has been created, a client and server in Java typically call methods of the `Socket` class to create input and output `streams`, which support all of the standard Java mechanisms for text I/O (Section C-8.7.3):

```
BufferedReader in = new BufferedReader(
    new InputStreamReader(client_connection.getInputStream()));
PrintStream out =
    new PrintStream(client_connection.getOutputStream());
// This is in the server; the client would make streams out
// of server_connection.
...
String s = in.readLine();
out.println("Hi, Mom\n");
```

Among all the message-passing mechanisms we have considered, datagrams are the only one that does not provide some sort of *ordering* constraint. In general, most message-passing systems guarantee that messages sent over the same "communication path" arrive in order. When naming processes explicitly, a path links a single sender to a single receiver. All messages from that sender to that receiver arrive in the order sent. When naming ports, a path links an arbitrary number of senders to a single receiver. Messages that arrive at a port in a given order will be seen by receivers in that order. Note, however, that while messages from the same sender will arrive at a port in order, messages from *different* senders may arrive in arbitrary orders.[2] When naming channels, a path links all the senders that can use the channel to all the receivers that can use it. A Java TCP connection has a single OS process at each end, but there may be many threads inside, each of which can use its process's end of the connection. The connection functions as a queue: send (enqueue) and receive (dequeue) operations are ordered, so that everything is received in the order it was sent.

13.5.2 Sending

One of the most important issues to be addressed when designing a `send` operation is the extent to which it may block the caller: once a thread has initiated a `send` operation, when is it allowed to continue execution? Blocking can serve at least three purposes:

Resource management: A sending thread should not modify outgoing data until the underlying system has copied the old values to a safe location. Most systems block the sender until a point at which it can safely modify its data, without danger of corrupting the outgoing message.

Failure semantics: Particularly when communicating over a long-distance network, message passing is more error-prone than most other aspects of computing. Many systems block a sender until they are able to guarantee that the message will be delivered without error.

Return parameters: In many cases a message constitutes a *request*, for which a *reply* is expected. Many systems block a sender until a reply has been received.

When deciding how long to block, we must consider synchronization semantics, buffering requirements, and the reporting of run-time errors.

---

2 Suppose, for example, that process *A* sends a message to port *p* of process *B*, and then sends a message to process *C*, while process *C* first receives the message from *A* and then sends its own message to port *p* of *B*. If messages are sent over a network with internal delays, and if *A* is allowed to send its message to *C* before its first message has reached port *p*, then it is possible for *B* to hear from *C* before it hears from *A*. This apparent reversal of ordering could easily happen on the Internet, for example, if the message from *A* to *B* traverses a satellite link, while the messages from *A* to *C* and from *C* to *B* use ocean-floor fiber-optic cables.

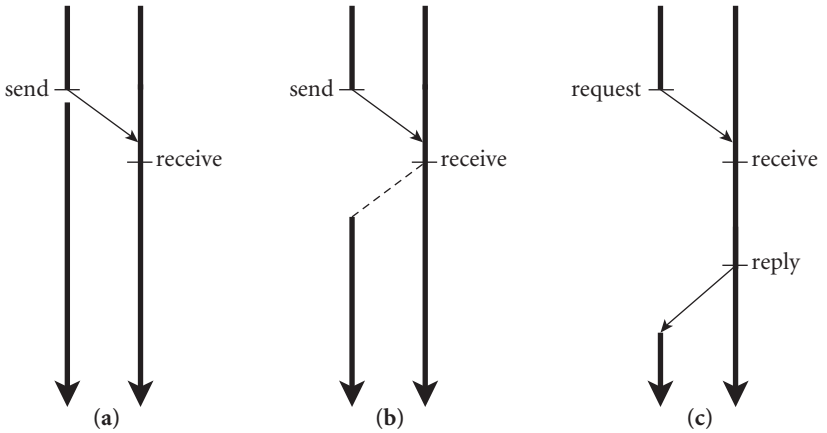

Figure 13.22 Synchronization semantics for the `send` operation: no-wait `send` (a), synchronization `send` (b), and remote-invocation `send` (c). In each diagram we have assumed that the original message arrives before the receiver executes its `receive` operation; this need not in general be the case.

Synchronization Semantics

On its way from a sender to a receiver, a message may pass through many intermediate steps, particularly if traversing the Internet. It first descends through several layers of software on the sender's machine, then through a potentially large number of intermediate machines, and finally up through several layers of software on the receiver's machine. We could imagine unblocking the sender after any of these steps, but most of the options would be indistinguishable in terms of user-level program behavior. If we assume for the moment that a message-passing system can always find buffer space to hold an outgoing message, then our three rationales for delay suggest three principal semantic options:

EXAMPLE 13.59
Three main options for `send` semantics

No-wait send: The sender does not block for more than a small, bounded period of time. The message-passing implementation copies the message to a safe location and takes responsibility for its delivery.

Synchronization send: The sender waits until its message has been received.

Remote-invocation send: The sender waits until it receives a reply.

These three alternatives are illustrated in Figure C-13.22. ■

No-wait `send` appears in Erlang and in the Java Internet library. Synchronization `send` appears in Occam and, by default, in Go. (If a Go channel is declared with an explicit *buffering capacity*, however, no-wait `send` is used.) Remote-invocation `send` appears in Ada and in Occam. MPI provides an implementation-oriented hybrid of no-wait `send` and synchronization `send`: a `send` operation blocks until the data in the outgoing message can safely be modified. In implementations that do their own internal buffering, this rule amounts to no-wait `send`. In other implementations, it amounts to synchronization `send`.

The programmer has the option, if desired, to insist on no-wait `send` or synchronization `send`; performance may suffer on some systems if the request is different from the default.

Buffering

In practice, unfortunately, no message-passing system can provide a version of `send` that never waits (unless of course it simply throws some messages away). If we imagine a thread that sits in a loop sending messages to a thread that never receives them, we quickly see that unlimited amounts of buffer space would be required. At some point, any implementation must be prepared to block an overactive sender, to keep it from overwhelming the system. Such blocking is a form of *backpressure*. Milder backpressure can also be applied by reducing a thread's scheduling priority or by increasing the (still bounded) delay before a "no-wait" `send` returns.

EXAMPLE 13.60
Buffering-dependent deadlock

For any fixed amount of buffer space, it is possible to design a program that requires a larger amount of space to run correctly. Imagine, for example, that the message-passing system is able to buffer n messages on a given communication path. Now imagine a program in which A sends $n + 1$ messages to B, followed by one message to C. C then sends one message to B, on a different communication path. Finally, B insists on receiving the message from C before receiving the messages from A. If A blocks after message n, implementation-dependent deadlock will result. The best that an implementation can do is to provide a sufficiently

DESIGN & IMPLEMENTATION

13.10 The semantic impact of implementation issues

The inability to buffer unlimited amounts of data and, likewise, to report errors synchronously to a sender that has continued execution are only the most recent of many examples we have seen in which pragmatic implementation issues may restrict the language semantics available to the programmer. Other examples include limitations on the length of source lines or variable names (Section 2.1.1); limits on the memory available for data (whether global, stack, or heap allocated) and for recursive function evaluation (Section 3.2); the lack of ranges in `case` statement labels (Section 6.4.2); `in reverse`, `downto`, and constant step sizes for `for` loops (Section 6.5.1); limits on set universe size (to accommodate bit vectors—Section 8.4); limited procedure nesting (to accommodate displays—Section 9.1); the pointer-only restriction on opaque exports in Modula-2 (Section 10.2.1); and the lack of nested threads or of unrestricted arms on a `cobegin` statement (to avoid the need for cactus stacks—Section 9.5.1). Some of these limitations are reflected in the formal semantics of the language. Others (generally those that vary most from one implementation to another) restrict the set of semantically valid programs that the system will run correctly.

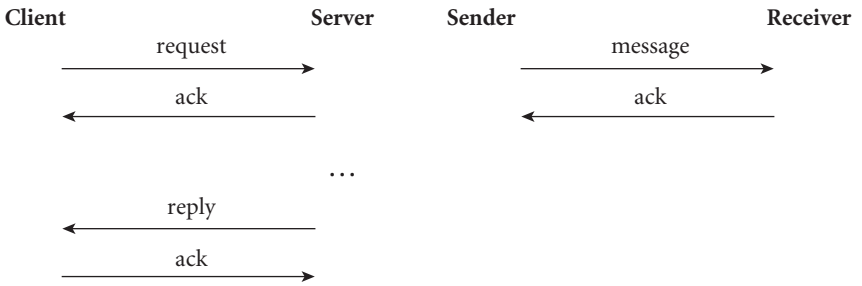

Figure 13.23 **Acknowledgment messages for error detection.** In the absence of piggybacking, remote-invocation `send` (left) may require four underlying messages; synchronization `send` (right) may require two.

large amount of space that realistic applications are unlikely to find the limit to be a problem. ■

For synchronization `send` and remote-invocation `send`, buffer space is not generally a problem: the total amount of space required for messages is bounded by the number of threads, and there are already likely to be limits on how many threads a program can create. A thread that sends a reply message can always be permitted to proceed: we know that we shall be able to reuse the buffer space quickly, because the thread that sent the request is already waiting for the reply.

Error Reporting

EXAMPLE 13.61
Acknowledgments

If the underlying message-passing system is unreliable, a language or library will typically employ *acknowledgment* messages to verify successful transmission (Figure C-13.23). If an acknowledgment is not received within a reasonable amount of time, the implementation will typically resend. If several attempts fail to elicit an acknowledgment, an error will be reported. ■

As long as the sender of a message is blocked, errors that occur in attempting to deliver a message can be reflected back as exceptions, or as status information in result parameters or global variables. Once a sender has continued, there is no obvious way in which to report any problems that arise. Like limits on message buffering, this dilemma poses semantic problems for no-wait `send`. For UDP, the solution is to state that messages are unreliable: if something goes wrong, the message is simply lost, silently. For TCP, the "solution" is to state that only "catastrophic" errors will cause a message to be lost, in which case the connection will become unusable and future calls will fail immediately. An even more drastic approach was taken in the original version of MPI: certain implementation-specific errors could be detected and handled at run time, but in general if a message could not be delivered then the program as a whole was considered to have failed. Newer versions of MPI provide a richer set of error-reporting facilities that can be used, with some effort, to build fault-tolerant programs.

Emulation of Alternatives

All three varieties of `send` can be emulated by the others. To obtain the effect of remote-invocation `send`, a thread can follow a no-wait `send` of a request with a `receive` of the reply, as we saw in Example C-13.56. Similar code will allow us to emulate remote-invocation `send` using synchronization `send`. To obtain the effect of synchronization `send`, a thread can follow a no-wait `send` with a `receive` of a high-level acknowledgment, which the receiver will send immediately upon receipt of the original message. To obtain the effect of synchronization `send` using remote-invocation `send`, a thread that receives a request can simply reply immediately, with no return parameters.

To obtain the effect of no-wait `send` using synchronization `send` or remote-invocation `send`, we must interpose a buffer process (the message-passing analogue of our shared-memory bounded buffer) that replies immediately to "senders" or "receivers" whenever possible. The space available in the buffer process makes explicit the resource limitations that are always present below the surface in implementations of no-wait `send`.

Syntax and Language Integration

In the emulation examples above, our hypothetical syntax assumed a library-based implementation of message passing. Because `send`, `receive`, `accept`, and so on are ordinary subroutines in such an implementation, they usually take a

DESIGN & IMPLEMENTATION

13.11 Emulation and efficiency

Unfortunately, user-level emulations of alternative `send` semantics are seldom as efficient as optimized implementations using the underlying primitives. Suppose for example that we wish to use remote-invocation `send` to emulate synchronization `send`. Suppose further that our implementation of remote-invocation `send` is built on top of network software that needs acknowledgments to verify message delivery. After sending a reply, the server's run-time system will wait for an acknowledgment from the client. If a server thread can work for an arbitrary amount of time before sending a reply, then the run-time system will need to send separate acknowledgments for the request and the reply. If a programmer uses this implementation of remote-invocation `send` to emulate synchronization `send`, then the underlying network may end up transmitting a total of four messages (more if there are any transmission errors). By contrast, a "native" implementation of synchronization `send` would require only two underlying messages. In some cases the run-time system for remote-invocation `send` may be able to delay transmission of the first acknowledgment long enough to "piggy-back" it on the subsequent reply if there is one; in this case an emulation of synchronization `send` may transmit three underlying messages instead of only two. We consider the efficiency of emulations further in Exercise C-13.36 and Exploration C-13.52.

fixed, static number of parameters, two of which typically specify the location and size of the message to be sent. To send a message containing values held in more than one program variable, the programmer may need to explicitly *gather*, or *marshal*, those values into the fields of a record. On the receiving end, the programmer may then need to *scatter* (*unmarshal*) the values back into program variables. By contrast, a concurrent programming language can provide message-passing operations whose "argument" lists can include an arbitrary number of values to be sent. Moreover, the compiler can arrange to perform type checking on those values, using techniques similar to those employed for subroutine linkage across compilation units (to be described in Section 15.6.2). Finally, as we shall see in Section C-13.5.3, an explicitly concurrent language can employ non-procedure-call syntax, for example to couple a remote-invocation `accept` and `reply` in such a way that the `reply` doesn't have to explicitly identify the `accept` to which it corresponds.

13.5.3 Receiving

Probably the most important dimension on which to categorize mechanisms for receiving messages is the distinction between explicit `receive` operations and the *implicit* receipt described in Section 13.2.3. Among the languages and systems we have been using as examples, none provides implicit receipt, but it appears in a variety of research languages, and in some of the RPC systems we will consider in Section C-13.5.4).

With implicit receipt, every message that arrives at a given port (or over a given channel) will create a new thread of control, subject to resource limitations (any implementation will have to stall incoming requests when the number of threads grows too large). With explicit receipt, a message will be queued until some already-existing thread indicates a willingness to receive it. At any given point in time there may be a potentially large number of messages waiting to be received. Most languages and libraries with explicit receipt allow a thread to exercise some sort of *selectivity* with respect to which messages it wants to consider.

In MPI, every message includes the `id` of the process that sent it, together with an integer *tag* specified by the sender. A `receive` operation specifies a desired sender `id` and message tag. Only matching messages will be received. In many cases receivers specify "wild cards" for the sender `id` and/or message tag, allowing any of a variety of messages to be received. Special versions of `receive` also allow a process to test (without blocking) to see if a message of a particular type is currently available (this operation is known as *polling*), or to "time out" and continue if a matching message cannot be received within a specified interval of time.

Because they are languages instead of library packages, Ada, Erlang, Go, and Occam are able to use special, non-procedure-call syntax for selective message receipt. Moreover because messages are built into the naming and typing system, these languages are able to receive selectively on the basis of port/channel names

```
task buffer is
    entry insert(d : in bdata);
    entry remove(d : out bdata);
end buffer;

task body buffer is
    SIZE : constant integer := 10;
    subtype index is integer range 1..SIZE;
    buf : array (index) of bdata;
    next_empty, next_full : index := 1;
    full_slots : integer range 0..SIZE := 0;
begin
    loop
        select
          when full_slots < SIZE =>
            accept insert(d : in bdata) do
                buf(next_empty) := d;
            end;
            next_empty := next_empty mod SIZE + 1;
            full_slots := full_slots + 1;
        or
          when full_slots > 0 =>
            accept remove(d : out bdata) do
                d := buf(next_full);
            end;
            next_full := next_full mod SIZE + 1;
            full_slots := full_slots - 1;
        end select;
    end loop;
end buffer;
```

Figure 13.24 Bounded buffer in Ada, with an explicit manager task.

and parameters, rather than the more primitive notion of tags. In all four languages, the selective `receive` construct is a special form of *guarded command*, as described in Section C-6.7.

EXAMPLE 13.62
Bounded buffer in Ada 83

Figure C-13.24 contains code for a bounded buffer in Ada 83. Here an active "manager" thread executes a `select` statement inside a loop. (Recall that it is also possible to write a bounded buffer in Ada using *protected objects*, without a manager thread, as described in Section 13.4.2.) The Ada `accept` statement receives the `in` and `in out` parameters (Section 9.3.1) of a remote invocation request. At the matching `end`, `accept` returns the `in out` and `out` parameters as a reply message. A client task would communicate with the bounded buffer using an *entry call*:

```
-- producer:                    -- consumer:
buffer.insert(3);               buffer.remove(x);
```

The `select` statement in our buffer example has two arms. The first arm may be selected when the buffer is not full and there is an available `insert` request; the second arm may be selected when the buffer is not empty and there is an available `remove` request. Selection among arms is a two-step process: first the guards (`when` expressions) are evaluated, then for any that are true the subsequent `accept` statements are considered to see if a message is available. (The guard in front of an `accept` is optional; if missing it behaves as `when true =>`.) If both of the guards in our example are true (the buffer is partly full) and both kinds of messages are available, then either arm of the statement may be executed, at the discretion of the implementation. (For a discussion of issues of *fairness* in the choice among true guards, see Sidebar C-6.11.) ■

EXAMPLE 13.63
Timeout and distributed termination

Every `select` statement must have at least one arm beginning with `accept` (and optionally `when`). In addition, it may have three other types of arms:

```
when condition => delay how_long
    other_statements
...
or when condition => terminate
...
else ...
```

A `delay` arm may be selected if no other arm becomes selectable within *how_long* seconds. (Ada implementations are required to support delays as long as 1 day or as short as 20 ms.) A `terminate` arm may be selected only if all potential communication partners have already terminated or are likewise stuck in `select` statements with `terminate` arms. Selection of the arm causes the task that was executing the `select` statement to terminate. An `else` arm, if present, will be selected when none of the guards are true or when no `accept` statement can be executed immediately. A `select` statement with an `else` arm is not permitted to have any `delay` arms. In practice, one would probably want to include a `terminate` arm in the `select` statement of a manager-style bounded buffer. ■

EXAMPLE 13.64
Bounded buffer in Go

In Go, a bounded buffer is trivial: it's just a buffered channel:

```
type bdata struct {
    n int   // or whatever
}
var buffer = make(chan bdata, 10)   // space for ten items of type bdata
...
buffer <- bdata{3}                  // insert
...
my_int = (<-buffer).n               // remove
```

To illustrate language features, we can also build a bounded buffer with an explicit thread, an array, and a pair of default (unbuffered) channels, in a manner similar to the Ada example of Figure C-13.24, but with synchronization `send` instead of remote invocation. Code for this alternative appears in Figure C-13.25.

Unlike built-in buffered channels, it could easily be augmented to support functionality like priority-based (as opposed to FIFO) queueing, or methods to clear the buffer or to query the number of messages currently queued. To use the basic `insert`/`remove` operations, we might write:

```
var b = make_buffer()
...
b.insert(bdata{3})                    // insert
...
my_int = b.remove().n                 // remove
```

As in the Ada example, requests are processed by an active manager thread (called a "goroutine" in Go), here started with the `go` command. The `select` statement in Go does not support explicit guards; we have achieved a similar effect in Figure C-13.25 by setting the `ic` and `rc` channels to `nil` when they should not be selected. Because we have used synchronization `send`—channels `insert_c` and `remove_c` have zero capacity—there is an asymmetry between the handling of `insert` and `remove` requests: the former need only send the manager data; the latter must send a channel reference and then wait for the manager to send the data back. ■

EXAMPLE 13.65
Bounded buffer in Erlang

In Erlang, which uses no-wait `send`, one might at first expect asymmetry similar to that of Figure C-13.25: a consumer would have to receive a reply from a bounded buffer, but a producer could simply send data. Such asymmetry would have a hidden flaw, however: because a process does not wait after sending, the producer could easily send more items than the buffer can hold, with the excess being buffered in the message system. If we want the buffer to truly be bounded, we must require the producer to wait for an acknowledgment. Code for the buffer appears in Figure C-13.26. Because Erlang is a functional language, we use tail recursion instead of iteration. Code for the producer and consumer looks like this:

```
-- producer:                          -- consumer:
Buffer ! {insert, X, self()},         Buffer ! {remove, self()},
receive ok -> [] end.                 receive X -> [] end.
```

The exclamation point (`!`), borrowed from CSP, is used to send a message. ■

EXAMPLE 13.66
Peeking at messages in Erlang

Several languages—Erlang among them—place the parameters of an incoming message within the scope of the guard condition, allowing a receiver to "peek inside" a message before deciding whether to receive it. In Erlang, we can say

```
receive
    {insert, D} when D rem 2 == 1 ->    % accept only odd numbers
```

The ability to peek implies that the content of incoming messages must be visible to the language run-time system. An Erlang implementation must therefore be prepared to accept (and buffer) an arbitrary number of messages; it cannot rely on the operating system or other underlying software to provide the buffering for it. Moreover the fact that buffer space can never be truly unlimited means that guards and scheduling expressions will be unable to see messages whose delivery has been delayed by backpressure. ■

```
type buffer struct {
    full_slots, next_full, next_empty int
    buf [SIZE]bdata
    insert_c chan bdata
    remove_c chan chan bdata
}
func manager(b *buffer) {
    var ic chan bdata = b.insert_c
    var rc chan chan bdata = nil
    for {           // at least one of ic and rc will always be non-nil
        select {
          case d := <-ic:       // := means "declare and initialize"
            b.buf[b.next_empty] = d
            b.next_empty = (b.next_empty + 1) % SIZE
            b.full_slots++
            rc = b.remove_c     // there is definitely data to remove
            if b.full_slots == SIZE { ic = nil }
          case c := <-rc:
            c <- b.buf[b.next_full]
            b.next_full = (b.next_full + 1) % SIZE
            b.full_slots--
            ic = b.insert_c     // there is definitely space to fill
            if b.full_slots == 0 { rc = nil }
        }
    }
}
func make_buffer() (b *buffer) {    // return value has name 'b'
    b = new(buffer)
    b.full_slots = 0
    b.next_full = 0
    b.next_empty = 0
    b.insert_c = make(chan bdata)
    b.remove_c = make(chan chan bdata)
    go manager(b)               // create active manager thread
    return
}
func (b *buffer) insert(e bdata) {
    b.insert_c <- e             // send data to manager
}
func (b *buffer) remove() bdata {
    var c = make(chan bdata)
    b.remove_c <- c             // send temporary channel to manager
    return <-c                  // receive and return response
}
```

Figure 13.25 **Bounded buffer with an explicit manager thread in Go.** The `insert` and `remove` functions serve as methods of buffer `b`. Note that in the absence of additional functionality (not shown), this code would better be replaced by trivial use of a buffered channel with capacity `SIZE`. Also, if using this version, we would probably want a way to terminate the manager thread when the buffer is no longer needed.

```
buffer(Max, Free, Q) ->
    receive
        {insert, D, Client} when Free > 0 ->
            Client ! ok,                              % send ack
            buffer(Max, Free-1, queue:in(D, Q));      % enqueue
        {remove, Client} when Free < Max ->
            {{value, D}, NewQ} = queue:out(Q),        % dequeue
            Client ! D,                               % send element
            buffer(Max, Free+1, NewQ)
    end.
```

Figure 13.26 Bounded buffer in Erlang. Variables (names that can be instantiated with a value) begin with a capital letter; constants begin with a lower-case letter. Queue operations (`in`, `out`) are provided by the standard Erlang library. Typing is dynamic. The `send` operator (`!`) is as in CSP and Occam. Each clause of the `receive` ends with a tail recursive call.

13.5.4 Remote Procedure Call

Any of the three principal forms of `send` (no-wait, synchronization, remote-invocation) can be paired with either of the principal forms of `receive` (explicit or implicit). The combination of remote-invocation `send` with explicit receipt (e.g., as in Ada) is sometimes known as *rendezvous*. The combination of remote-invocation `send` with implicit receipt is usually known as *remote procedure call*. RPC is available in several concurrent languages, and is also supported on many systems by augmenting a sequential language with a *stub compiler*. The stub compiler is independent of the language's regular compiler. It accepts as input a formal description of the subroutines that are to be called remotely. The description is roughly equivalent to the subroutine headers and declarations of the types of all parameters. Based on this input the stub compiler generates source code for *client* and *server stubs*. A client stub for a given subroutine marshals request parameters and an indication of the desired operation into a message buffer, sends the message to the server, waits for a reply message, and unmarshals that message into result parameters. A server stub takes a message buffer as parameter, unmarshals request parameters, calls the appropriate local subroutine, marshals return parameters into a reply message, and sends that message back to the appropriate client. Invocation of a client stub is relatively straightforward. Invocation of server stubs is discussed under "Implementation" below.

Semantics

A principal goal of most RPC systems is to make the remote nature of calls as *transparent* as possible; that is, to make remote calls look as much like local calls as possible [BN84]. In a stub compiler system, a client stub should have the same interface as the remote procedure for which it acts as proxy; the programmer should usually be able to call the routine without knowing or caring whether it is local or remote.

Several issues make it difficult to achieve transparency in practice:

Parameter modes: It is difficult to implement call-by-reference parameters across a network, since actual parameters will not be in the address space of the called routine. (Access to global variables is similarly difficult.)

Performance: There is no escaping the fact that remote procedures may take a long time to return. In the face of network delays, one cannot use them casually.

Failure semantics: Remote procedures are much more likely to fail than are local procedures. It is generally acceptable in the local case to assume that a called procedure will either run exactly once or else the entire program will fail. Such an assumption is overly restrictive in the remote case.

We can use value/result parameters in place of reference parameters so long as program correctness does not rely on the aliasing created by reference parameters. As noted in Section 9.3.1, Ada declares that a program is *erroneous* if it can tell the difference between pass-by-reference and pass-by-value/result implementations of `in out` parameters. If absolutely necessary, reference parameters and global variables can be implemented with message-passing thunks in a manner reminiscent of call-by-name parameters (Section C-9.3.2), but only at very high cost. As noted in Section 7.4, a few languages and systems perform deep copies of linked data structures passed to remote routines.

Performance differences between local and remote calls can be hidden only by artificially slowing down the local case. Such an option is clearly unacceptable.

DESIGN & IMPLEMENTATION

13.12 Parameters to remote procedures

Ada's comparatively high-level semantics for parameter modes allows the same set of modes to be used for both subroutines and entries (rendezvous). An Ada compiler will generally pass a large argument to a subroutine by reference whenever possible, to avoid the expense of copying. If tasks are on separate nodes of a cluster, however, the compiler will generally pass the same argument to an entry by value-result.

A few concurrent languages provide parameter modes specifically designed with remote invocation in mind. In Emerald [BHJL07], for example, every parameter is a reference to an object. References to remote objects are implemented transparently via message passing. To minimize the frequency of such references, objects passed to remote procedures often *migrate* with the call: they are packaged with the request message, sent to the remote site (where they can be accessed locally), and returned to the caller in the reply. Emerald calls this *call by move*. In Hermes [SBG+91] and Rust, parameter passing is *destructive*: arguments become uninitialized from the caller's point of view, and can therefore migrate to a remote callee without danger of inducing remote references.

Exactly-once failure semantics can be provided by aborting the caller in the event of failure or, in highly reliable systems, by delaying the caller until the operating system or language run-time system is able to rebuild the failed computation using information previously dumped to disk. (Failure recovery techniques are beyond the scope of this text.) An attractive alternative is to accept "at-most-once" semantics with notification of failure. The implementation retransmits requests for remote invocations as necessary in an attempt to recover from lost messages. It guarantees that retransmissions will never cause an invocation to happen more than once, but it admits that in the presence of communication failures the invocation may not happen at all. If the programming language provides exceptions then the implementation can use them to make communication failures look like any other kind of run-time error.

Implementation

At the level of the kernel interface, `receive` is usually an explicit operation. To make `receive` appear implicit to the application programmer, the code produced by an RPC stub compiler (or the run-time system of an RPC-based language) must bridge this explicit-to-implicit gap. The typical implementation resembles the thread-based event handling of Section 9.6.2. We describe it here in terms of stub compilers; in a concurrent language with implicit receipt the regular compiler does essentially the same work.

EXAMPLE 13.67
An RPC server system

Figure C-13.27 illustrates the layers of a typical RPC system. Code above the upper horizontal line is written by the application programmer. Code in the middle is a combination of library routines and code produced by the RPC stub compiler. To initialize the RPC system, the application makes a pair of calls into the run-time system. The first provides the system with pointers to the stub routines produced by the stub compiler; the second starts a *message dispatcher*. What happens after this second call depends on whether the server is concurrent and, if so, whether its threads are implemented on top of one OS process or several.

In the simplest case—a single-threaded server on a single OS process—the dispatcher runs a loop that calls into the kernel to receive a message. When a message arrives, the dispatcher calls the appropriate RPC stub, which unmarshals request parameters and calls the appropriate application-level procedure. When that procedure returns, the stub marshals return parameters into a reply message, calls into the kernel to send the message back to the caller, and then returns to the dispatcher. ■

This simple organization works well so long as each remote request can be handled quickly, without ever needing to block. If remote requests must sometimes wait for user-level synchronization, then the server's process must manage a ready list of threads, as described in Section 13.2.4, but with the dispatcher integrated into the usual thread scheduler. When the current thread blocks (in application code), the scheduler/dispatcher will grab a new thread from the ready list. If the ready list is empty, the scheduler/dispatcher will call into the kernel to receive a message, fork a new user-level thread to handle it, and then continue to execute

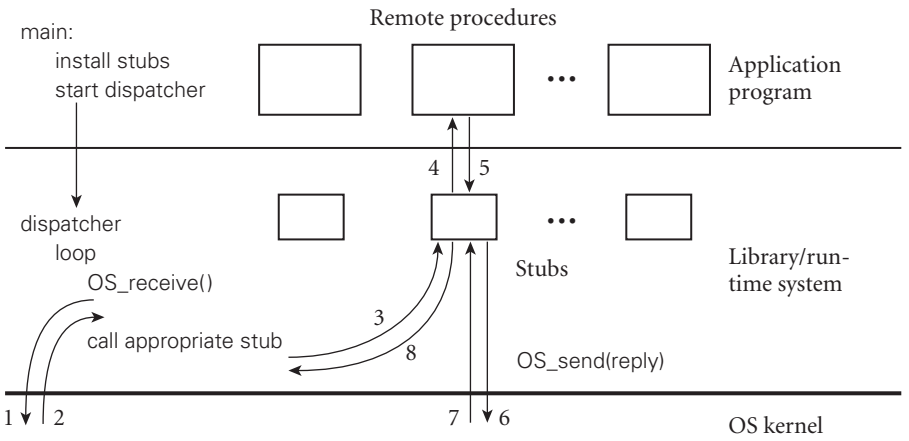

Figure 13.27 **Implementation of a remote procedure call server.** Application code initializes the RPC system by installing stubs generated by the stub compiler (not shown). It then calls into the run-time system to enable incoming calls. Depending on details of the particular system in use, the dispatcher may use the thread from the main program (in which case the call to start the dispatcher never returns), or it may create a pool of threads that handle incoming requests.

runnable threads until the list is empty again (each thread will terminate when it finishes handling its request).

In a multithreaded server, the call to start the dispatcher will generally ask the kernel to fork a "pool" of threads to service remote requests. Each of these threads will then perform the operations described in the previous paragraphs. In a language or library with a one–one correspondence between user threads and kernel threads, each will repeatedly receive a message from the kernel, call the appropriate stub, and loop back for another request. With a more general thread package, each kernel thread will run threads from the application's ready list until the list is empty, at which point it (the kernel thread) will call into the kernel for another message. So long as the number of runnable user threads is greater than or equal to the number of kernel threads, no new messages will be received. When the number of runnable user threads drops below the number of kernel threads, the extra kernel threads will call into the kernel, where they will block until requests arrive.

✓ CHECK YOUR UNDERSTANDING

50. Describe three ways in which processes or threads commonly name their communication partners.

51. What is a *datagram*?

52. Why, in general, might a `send` operation need to block?
53. What are the three principal synchronization options for the sender of a message? What are the tradeoffs among them?
54. What are *gather* and *scatter* operations in a message-passing program? What are *marshalling* and *unmarshalling*?
55. Describe the tradeoffs between *explicit* and *implicit* message receipt.
56. What is a *remote procedure call* (RPC)? What is a *stub compiler*?
57. What are the obstacles to *transparency* in an RPC system?
58. What is a *rendezvous*? How does it differ from a remote procedure call?
59. Explain the purpose of a `select` statement in Ada or Go.
60. What semantic and pragmatic challenges are introduced by the ability to "peek" inside messages before they are received?