

6. Message-Passing in Distributed Programs

A distributed program is a collection of concurrent processes that run on a network of computers:

- Typically, each process is a concurrent program that executes on a single computer.
- A process (or program) on one computer communicates with processes on other computers by passing messages across the network.
- The threads in a single process all execute on the same computer, so they can use message passing and/or shared variables to communicate.

Outline:

- Mechanisms for message passing in distributed programs.
- Java message passing classes
- Distributed solutions to several classical synchronization problems.\
- Tracing, testing, and replaying distributed programs.

6.1 TCP Sockets

Communication channels are formed across a communications network with help from the operating system.

- Each thread creates an endpoint object that represents its end of the network channel.
- The operating system manages the hardware and software that is used to transport messages between the endpoints, i.e., “across the channel”.

The endpoint objects are called *sockets* (Fig. 6.1):

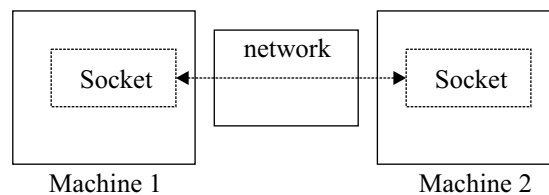


Figure 6.1 Sockets are the endpoints of channels across a network.

- The client thread’s socket specifies a local IO port to be used for sending messages (or the IO port can be chosen by the operating system).
- The client’s socket also specifies the address of the destination machine and the port number that is expected to be bound to the server thread’s socket.
- The server’s socket specifies a local IO port for receiving messages. Messages can be received from any client that knows both the server’s machine address and the port number bound to the server’s socket.
- The client issues a request to the server to form a connection between the two sockets. Once the server accepts the connection request, messages can be passed in either direction across the channel.

6.1.1 Channel Reliability

Messages that travel across a network may be lost or corrupted, or they may arrive out of order. Some applications may be able to tolerate this, e.g., streaming music.

An application can choose how reliably its messages are transmitted by selecting a transport protocol:

- Transmission Control Protocol (TCP) ensures the reliable transmission of messages. TCP guarantees that messages are not lost or corrupted, and that messages are delivered in the correct order. However, this adds overhead to message transport.
- User Data Protocol (UDP) is a fast but unreliable method for transporting messages. Messages sent using UDP will not be corrupted, but they may be lost or duplicated, and they may arrive in an order different from the order in which they were sent.

Both TCP and UDP utilize the Internet Protocol (IP) to carry packets of data. The IP standard defines the packet format, which includes formats for specifying source and destination addresses and IO ports.

6.1.2 TCP Sockets in Java

The `java.net` class library provides classes *Socket* and *ServerSocket* for TCP-based message passing.

6.1.2.1 Class Socket.

A client's first step is to create a TCP socket and try to connect to the server:

```
InetAddress host;                // server's machine address
int serverPort = 2020;           // port number bound to server's socket
Socket socket;                  // client's socket
try { host = InetAddress.getByName("www.cs.gmu.edu"); }
catch (UnknownHostException) { ... }
try { socket = new Socket(host,serverPort); } // create a socket and request a
catch (IOException e) { ... }           // connection to host
```

The client assumes that the server is listening for TCP connection requests on port *serverPort*. The *Socket* constructor throws an *IOException* if it cannot make a connection.

Client creates an input stream to receive data from its socket and an output stream to send data to the socket at the server's end of the channel. (Looks like file I/O!)

```
PrintWriter toServer = new PrintWriter(socket.getOutputStream(),true);
BufferedReader fromServer = new BufferedReader(new
    inputStreamReader(socket.getInputStream()));
toServer.println("Hello");           // send a message to the server
String line = fromServer.readLine(); // receive the server's reply
System.out.println ("Client received: " + line + " from Server");
toServer.close(); fromServer.close(); socket.close();
```

A read operation on the *InputStream* associated with a *Socket* normally blocks. To set a one second timeout:

```
socket.setSoTimeout(1000);          // set a one second timeout
```

When the timeout expires, a *java.net.SocketTimeoutException* is raised and the *Socket* is still valid.

6.1.2.2 Class ServerSocket. The server begins by creating a *ServerSocket*:

```
int serverPort = 2020;
ServerSocket listen;
try { listen = new ServerSocket(serverPort); }
catch (IOException e) { ... }
```

The server then calls the *accept()* method of the *ServerSocket*:

- method *accept()* waits until a client requests a connection,
- it then returns a *Socket* that connects the client to the server.
- server gets input and output streams and communicates with the client.
- client, server, or both, close the connection, and the server waits for a connection request from another client.

```
try {
    listen = new ServerSocket(serverPort);
    while (true) {
        Socket socket = listen.accept(); // wait for a client request
        toClient = new PrintWriter(socket.getOutputStream(),true);
        fromClient = new BufferedReader(new
            InputStreamReader(socket.getInputStream()));
        String line = fromClient.readLine();// receive a message from the client
        System.out.println("Server received " + line);
        toClient.println("Good-bye"); // send a reply to the client
    }
}
catch (IOException e) { ... }
finally { if (listen != null) try { listen.close();}
        catch (IOException e) { e.printStackTrace(); } }
```

The server can create a separate thread to handle the client's requests.

```
Socket socket = listen.accept();
clientHandler c = new clientHandler(socket); // clientHandler extends Thread
c.start();
```

The *run()* method of class *clientHandler* uses input and output streams obtained from the *socket* to communicate with the client, exactly as shown above.

Listing 6.2 shows a client and server program using TCP sockets:

- *Message* objects are serialized and passed between the client and the server.
- If the client program is executed using “java Client 2”, then the client will send the value 2 to the server, receive the value 4 from the server, and display the message “2 x 2 = 4”.

```
import java.net.*; import java.io.*;
public final class Client {
    public static void main (String args[]) {
        int serverPort = 2020; Socket socket = null;
        ObjectOutputStream toServer = null; ObjectInputStream fromServer=null;
        try {
            if (args.length != 1) {
                System.out.println("need 1 argument"); System.exit(1);
            }
            int number = Integer.parseInt(args[0]);
            // client and server run on the same machine, known as the Local Host
            InetAddress serverHost = InetAddress.getLocalHost();
            socket = new Socket(serverHost,serverPort);
            // send a value to the server
            toServer = new ObjectOutputStream(new BufferedOutputStream
                (socket.getOutputStream()));
            Message msgToSend = new Message(number);
            toServer.writeObject(msgToSend); toServer.flush();
            // This will block until the corresponding ObjectOutputStream in the
            // server has written an object and flushed the header.
            fromServer = new ObjectInputStream(new
                BufferedInputStream(socket.getInputStream()));
            Message msgFromReply = (Message) fromServer.readObject();
            System.out.println (number + " x " + number + " = " +
                msgFromReply.number);
        }
        catch (IOException e) { e.printStackTrace(); System.exit(1); }
        catch (ClassNotFoundException e) {e.printStackTrace(); System.exit(1); }
        finally { if (socket != null) try { socket.close();} catch (IOException e) {
            e.printStackTrace(); }}
    }
}
```

```

public final class Server {
    public static void main (String args[]) {
        int serverPort = 2020; ServerSocket listen=null;
        ObjectOutputStream toClient; ObjectInputStream fromClient;
        try {
            listen = new ServerSocket(serverPort);
            while (true) {
                Socket socket = listen.accept();
                toClient = new ObjectOutputStream(new BufferedOutputStream
                    (socket.getOutputStream()));
                // This will block until the corresponding ObjectOutputStream in the
                // client has written an object and flushed the header.
                fromClient = new ObjectInputStream(new BufferedInputStream
                    (socket.getInputStream()));
                Message msgRequest = (Message) fromClient.readObject();
                // compute a reply and send it back to the client
                int number = msgRequest.number;
                toClient.writeObject(new Message(number*number)); toClient.flush();
            } }
        catch (IOException e) {e.printStackTrace(); System.exit(1); }
        catch (ClassNotFoundException e) {e.printStackTrace(); System.exit(1); }
        finally { if (listen != null) try { listen.close();} catch (IOException e) {
            e.printStackTrace(); }}
    }
}

public final class Message implements Serializable {
    public int number;
    Message(int number) {this.number = number; }
}

```

Listing 6.2 Client and server using TCP sockets.

6.2 Java TCP Channel Classes

The custom TCP channel classes:

- provide asynchronous and synchronous message passing
- can be used with the selective wait statement presented in Chapter 5
- hide the complexity of using TCP
- support tracing, testing, and replay in a distributed environment.
- based on the notion of a mailbox that holds messages deposited by senders and withdrawn by receivers. (They are not based on the client and server paradigm.)

6.2.1 Classes *TCP*Sender and *TCP*Mailbox

Classes *TCP*Sender and *TCP*Mailbox provide asynchronous message passing using a buffer-blocking *send()* method and a blocking *receive()* method, respectively.

Listing 6.3 illustrates the use of these classes in a distributed solution to the bounded buffer problem.

- Program *Producer* creates a *TCP*Sender object named *deposit* for sending items to program *Buffer*.
- Program *Buffer* creates a *TCP*Mailbox object named *deposit* for receiving items from the *Producer*, and a *TCP*Sender object named *withdraw* for sending items to the *Consumer*.
- Program *Buffer* acts as a one-slot bounded buffer, receiving items from the *Producer* and forwarding them to the *Consumer*.
- Program *Consumer* has a *TCP*Mailbox object named *withdraw* to receive the messages sent by *Buffer*.

Method *connect()*:

- must be called before a *TCPSender* can be used to send messages.
- opens a TCP connection using the host and port number specified when the *TCPSender* object is constructed.

When there are no more messages to send, method *close()* is called and the TCP connection is closed.

Methods *send()* and *receive()* both operate on *messageParts* objects.

- A *messageParts* object packages the message to be sent with the return address of the sender. (The return address is optional.)
- The return address information includes the sender's host address and a port number that the sender will use to wait for a reply.

A thread that calls *receive()* receives a *messageParts* object.

- The return address in this object can be saved and used later to reply to the sender.
- If no return address is needed, a *messageParts* object can be constructed without a return address.

```

import java.net.*;
import java.io.*;
public final class Producer {
    public static void main (String args[]) {
        final int bufferPort = 2020; String bufferHost = null;
        try {
            bufferHost = InetAddress.getLocalHost().getHostName();
            TCPSender deposit = new TCPSender(bufferHost,bufferPort);
            deposit.connect();
            for (int i=0; i<3; i++) {
                System.out.println("Producing" + i);
                messageParts msg = new messageParts(new Message(i));
                deposit.send(msg);
            }
            deposit.close();
        }
        catch (UnknownHostException e) {e.printStackTrace();}
        catch (TCPChannelException e) {e.printStackTrace();}
    }
}

public final class Buffer {
    public static void main (String args[]) {
        final int bufferPort = 2020; final int consumerPort = 2022;
        try {
            String consumerHost = InetAddress.getLocalHost().getHostName();
            TCPMailbox deposit = new TCPMailbox(bufferPort,"deposit");
            TCPSender withdraw = new TCPSender(consumerHost,consumerPort);
            withdraw.connect();
            for (int i=0; i<3; i++) {
                messageParts m = (messageParts) deposit.receive();
                withdraw.send(m);
            }
            withdraw.close(); deposit.close();
        }
        catch (UnknownHostException e) {e.printStackTrace();}
        catch (TCPChannelException e) {e.printStackTrace();}
    }
}

```

```

public final class Consumer {
    public static void main (String args[]) {
        final int consumerPort = 2022;
        try {
            TCPMailbox withdraw = new TCPMailbox(consumerPort,"withdraw");
            for (int i=0; i<3; i++) {
                messageParts m = (messageParts) withdraw.receive();
                Message msg = (Message) m.obj;
                System.out.println("Consumed " + msg.number);
            }
            withdraw.close();
        }
        catch (TCPChannelException e) {e.printStackTrace();}
    }
}

public final class messageParts implements Serializable {
    public Object obj; // message to be sent
    public String host; // host address of the sender
    public int port; // port where sender will wait for a reply, if any
    messageParts(Object obj, String host, int port) {
        this.obj = obj; this.host = host; this.port = port;
    }
    // no return address
    messageParts(Object obj) {this.obj = obj; this.host = ""; this.port = 0;}
}

```

Listing 6.3 Distributed bounded buffer using *TCPSEnder* and *TCPMailbox*.

Listing 6.4 shows classes *TCPSender* and *TCPMailbox*.

Class *TCPSender*:

- method *send()* is used to send messages to a particular *TCPMailbox*. The host address and port number of the destination *TCPMailbox* are specified when the *TCPSender* object is constructed.
- Method *connect()* is used to connect the *TCPSender* to the *TCPMailbox*. Once the connection is made, each call to *send()* uses the same connection.
- The connection is closed with a call to method *close ()*.

A single *TCPMailbox* object may receive connection requests from any number of *TCPSenders*:

- A *TCPMailbox* objects begins listening for connection requests when the *TCPMailbox* object is constructed.
- If multiple *TCPSenders* connect to the same *TCPMailbox*, the connections are handled concurrently.
- When *close()* is called on a *TCPMailbox* object, it stops listening for new connection requests.

Method *receive()* of class *TCPMailbox* returns a *messageParts* object:

- The *messageParts* object returned by *receive()* is withdrawn from a *messageBuffer* called *buffer*.
- A *messageBuffer* is simply a bounded buffer of *messagePart* objects implemented as an SU monitor

A *TCPMailbox* object is an “active object” -- during construction it automatically starts an internal thread to receive messages:

```
public class TCPMailbox implements Runnable {
    ...
    public TCPMailbox( int port, String channelName ) throws
        TCPChannelException {
        this.port = port;
        this.channelName = channelName;
        try {listen = new ServerSocket( port ); }
        catch (IOException e) { e.printStackTrace();
            throw new TCPChannelException(e.getMessage());
        }
        buffer = new messageBuffer(100);
        Thread internal = new Thread(this);
        internal.start(); // internal thread executes method run()
    }
    ...
}
```

The *run()* method of *TCPMailbox* accepts connection requests from *TCPSender* objects and starts a *connectionHandler* thread to handle the connection:

```
public void run() {
    while (true) {
        Socket socket = listen.accept(); // listen for senders
        (new connectionHandler(socket)).start();
    }
}
```

The *connectionHandler* thread obtains an input stream from the *socket*:

```
connectionHandler (Socket socket) throws IOException {
    this.socket = socket;
    from = new ObjectInputStream(socket.getInputStream());
}
```

and then uses the input stream to receive *messageParts* objects. The *messagePart* objects are deposited into the *messageBuffer*:

```
while (true) { // read objects until get EOF
    messageParts msg = null;
    try {
        msg = (messageParts) from.readObject(); // receive messageParts object
        buffer.deposit(msg); // deposit messageParts object into buffer
    }
    catch (EOFException e) { break;}
}
```

If *buffer* becomes full, method *deposit()* will block. This will prevent any more *messagePart* objects from being received until a *messagePart* object is withdrawn using method *receive()*.

Note: The preferred way to use a *TCPSEnder* object S is to issue an *S.close()* operation only after all the messages have been sent:

- An *S.connect()* operation appears at the beginning of the program and an *S.close()* operation appears at the end (refer back to Listing 6.3.).
- All the messages sent over S will use a single connection.
- If multiple *TCPSEnder* objects connect to the same *TCPMailbox*, the *TCPMailbox* will handle the connections concurrently.

6.2.2 Classes *TCPsynchronousSender* and *TCPsynchronousMailbox*

Classes *TCPsynchronousSender* and *TCPsynchronousMailbox* implement synchronous channels:

- method *send()* of *TCPsynchronousSender* waits for an acknowledgment that the sent message has been received by the destination thread.
- The *receive()* method of *TCPsynchronousMailbox* sends an acknowledgement when the message is withdrawn from the *messageBuffer*, indicating that the destination thread has received the message.

6.2.3 Class *TCPSelectableSynchronousMailbox*

A *TCPSelectableSynchronousMailbox* object is used just like a *selectableEntry* or *selectablePort* object in Chapter 5.

Listing 6.6 shows bounded buffer program *Buffer*, which uses a *selectiveWait* object and *TCPSelectableSynchronousMailbox* objects *deposit* and *withdraw*.

Notice that *Buffer* selects *deposit* and *withdraw* alternatives in an infinite loop. One way to terminate this loop is to add a *delay* alternative to the selective wait, which would give *Buffer* a chance to timeout and terminate after a period of inactivity.

Detecting the point at which a distributed computation has terminated is not trivial since no process has complete knowledge of the global state of the computation, and neither global time nor common memory exists in a distributed system.

```

public final class Buffer {
    public static void main (String args[]) {
        final int depositPort = 2020; final int withdrawPort = 2021;
        final int withdrawReplyPort = 2022; int fullSlots=0; int capacity = 2;
        Object[] buffer = new Object[capacity]; int in = 0, out = 0;
        try {
            TCPSelectableSynchronousMailbox deposit = new
                TCPSelectableSynchronousMailbox(depositPort);
            TCPSelectableSynchronousMailbox withdraw = new
                TCPSelectableSynchronousMailbox (withdrawPort);
            String consumerHost = InetAddress.getLocalHost().getHostName();
            TCPSender withdrawReply = new
                TCPSender(consumerHost,withdrawReplyPort);
            selectiveWait select = new selectiveWait();
            select.add(deposit);           // alternative 1
            select.add(withdraw);         // alternative 2
            while(true) {
                withdraw.guard(fullSlots>0);
                deposit.guard (fullSlots<capacity);
                switch (select.choose()) {
                    case 1:Object o = deposit.receive(); // item from Producer
                        buffer[in] = o; in = (in + 1) % capacity; ++fullSlots;
                        break;
                    case 2:messageParts withdrawRequest = withdraw.receive();
                        messageParts m = (messageParts) buffer[out];
                        try {// send an item back to the Consumer
                            withdrawReply.send(m);
                        } catch (TCPChannelException e)
                            {e.printStackTrace();}
                        out = (out + 1) % capacity; --fullSlots;
                        break;
                }
            }
        } catch (InterruptedException e) {e.printStackTrace();System.exit(1);}
        catch (TCPChannelException e) {e.printStackTrace();System.exit(1);}
        catch (UnknownHostException e) {e.printStackTrace();}
    }
}

```

Listing 6.6 Using a *selectiveWait* statement in a distributed program.

6.3 Timestamps and Event Ordering

In a distributed environment, it is difficult to determine the execution order of events.

Distributed mutual exclusion: Distributed processes that need access to a shared resource must send each other requests to obtain exclusive access to the resource. Processes can access the shared resource in the order of their requests, but the request order is not easy to determine.

Event ordering is also a critical problem during testing and debugging.

- the event order observed during tracing must be consistent with an event order that actually occurred.
- reachability testing depends on accurate event ordering to identify concurrent events and generate race variants.

6.3.1 Event Ordering Problems

Consider the following program, which uses asynchronous communication:

<u>Thread1</u>	<u>Thread2</u>	<u>Thread3</u>
(a) send A to Thread2;	(b) receive X;	(d) send B to Thread2;
	(c) receive Y;	

The possible executions of this program are represented by diagrams (D1) and (D2) in Fig. 6.7.

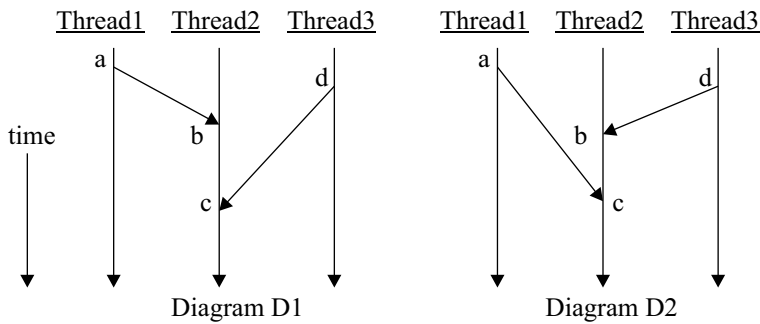


Figure 6.7 Diagrams D1 and D2.

Assume that the threads in diagram D1 send asynchronous trace messages to the controller whenever they execute a message passing event.

Fig. 6.8 illustrates two “observability problems” that can occur when the controller relies on the arrival order of the trace messages to determine the order of events.

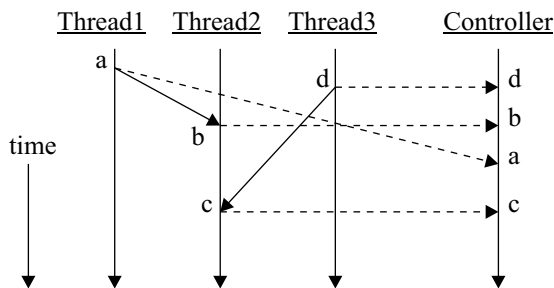


Figure 6.8 Observability Problems for Diagram D1

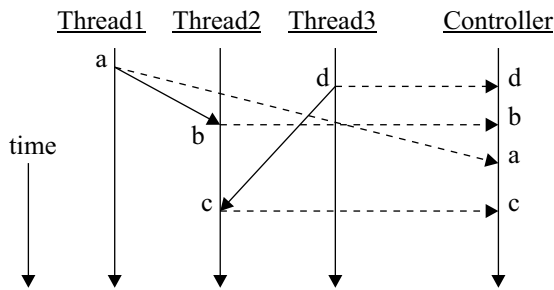


Figure 6.8 Observability Problems for Diagram D1

Incorrect orderings: The controller observes event *b* occur before event *a*, which is not what happened.

Arbitrary orderings:

- The controller observes event *d* occur before event *b*. Since *d* and *b* are concurrent events they can occur in either order, and the order the controller will observe is non-deterministic.
- The controller cannot distinguish non-deterministic orderings from orderings that are enforced by the program.
 - the programmer may mistakenly conclude that *d must precede b*.
 - the programmer may feel the need to create a test case where the order of events *d* and *b* is reversed, even though this change is not significant.

The controller can use timestamps to accurately order events. In Table 6.1 An 'X' indicates that a timestamp mechanism has a particular observability problem.

Observability Problems	Timestamp Mechanisms				
	Arrival order	Local real-time clocks	Global real-time clock	Totally-ordered logical clocks	Partially-ordered logical clocks
Incorrect orderings	X	X			
Arbitrary orderings	X	X	X	X	

Table 6.1 Effectiveness of timestamping mechanisms [Fidge 1996]. An 'X' indicates that a problem exists.

6.3.2 Local Real-Time Clocks

The real-time clock available on each processor can be used as the source of the timestamp.

Since the real-time clocks on different processors are not synchronized, incorrect orderings may be seen, and concurrent events are arbitrarily ordered.

Fig. 6.9 shows two ways in which the events in diagram D1 of Fig. 6.7 can be timestamped.

- On the left, the clock of Thread 1's processor is ahead of the clock of Thread 2's processor so event *b* erroneously appears to occur before event *a*.
- The ordering of events *d* and *b* is arbitrary, and depends on the relative speeds of the threads and the amount by which the processor's real-time clocks differ.

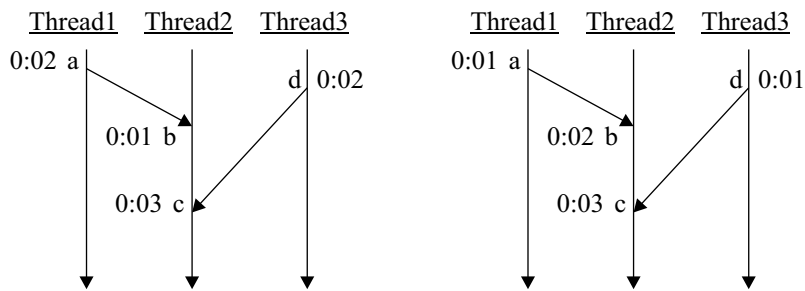


Figure 6.9 Timestamps using unsynchronized local real-time clocks.

6.3.3 Global Real-Time Clocks

If local real-time clocks are synchronized, there is a global reference for real time.

- avoids incorrect orderings, but as shown in Fig. 6.10, arbitrary orderings are still imposed on concurrent events *d* and *b*.
- Sufficiently accurate clock synchronization is difficult and sometimes impossible to achieve.

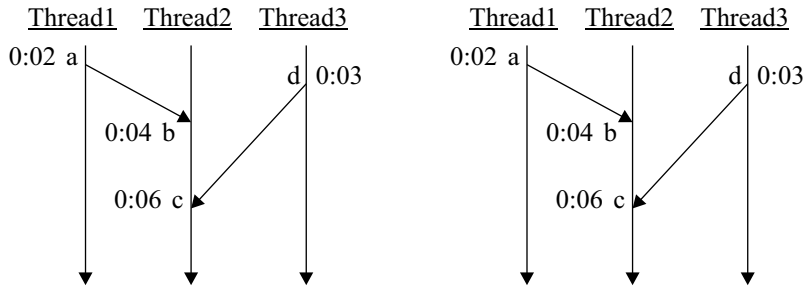


Figure 6.10 Timestamps using a real-time global clock.

6.3.4 Causality

The remaining two schemes use logical clocks instead of real-time clocks.

Logical clock schemes rely on the semantics of program operations to determine whether one event occurred before another event:

- if events A and B are local events in the same thread and A is executed before B, then A's logical timestamp will indicate that A happened before B.
- if S is an asynchronous send event in one thread and R is the corresponding receive event in another thread, then S's logical timestamp will indicate that S occurred before R.

(More accurately, S occurred before the *completion* of R, since the send operation S might have occurred long after the receive operation R started waiting for a message to arrive.)

It is important that any event ordering is consistent with the cause and effect relationships between the events.

The causality or "happened before" relation " \Rightarrow " for an execution of a message-passing program is defined as follows:

(C1) If events e and f are events in the same thread and e occurs before f , then $e \Rightarrow f$.

(C2) If there is a message $e \rightarrow f$ (i.e., e is a non-blocking send and f is the corresponding receive), then $e \Rightarrow f$.

(C3) If there is a message $e \leftrightarrow f$ or $f \leftrightarrow e$ (i.e., one of e or f is a blocking send and the other is the corresponding blocking receive), then for event g such that $e \Rightarrow g$, we have $f \Rightarrow g$, and for event h such that $h \Rightarrow f$, we have $h \Rightarrow e$.

(C4) If $e \Rightarrow f$ and $f \Rightarrow g$, then $e \Rightarrow g$. (Thus, " \Rightarrow " is transitive.)

It is easy to visually examine a space-time diagram and determine the causal relations:

For two events e and f in a space-time diagram, $e \Rightarrow f$ if and only if there is no message $e \leftrightarrow f$ or $f \leftrightarrow e$ and there exists a path from e to f that follows the vertical lines and arrows in the diagram.

(A double headed arrow allows a path to cross in either direction.).

For events e and f of an execution, if neither $e \Rightarrow f$ nor $f \Rightarrow e$, then e and f are said to be concurrent, denoted as $e \parallel f$:

- if there is a message $e \leftrightarrow f$ or $f \leftrightarrow e$, then e and f are concurrent events.
- Since $e \parallel f$ and $f \parallel e$ are equivalent, the " \parallel " relation is symmetric.
- the " \parallel " relation is not transitive.

In diagram (D1), $a \rightarrow b$, $b \rightarrow c$, $d \rightarrow c$, $a \rightarrow c$, $a \parallel d$, and $d \parallel b$, but a and b are not concurrent events.

In diagram (D2), $a \rightarrow c$, $b \rightarrow c$, $d \rightarrow c$, $d \rightarrow c$, $b \parallel a$, and $a \parallel d$, but b and d are not concurrent events.

6.3.5 Integer Timestamps

Each event receives a logical (integer) timestamp and these timestamps are used to order the events.

Consider a message-passing program that uses asynchronous communication and contains threads Thread_1 , Thread_2 , ..., and Thread_n . Thread_i , $1 \leq i \leq n$, contains a logical clock C_i , which is simply an integer variable initialized to 0.

During execution, logical time flows as follows:

- (IT1) Thread_i increments C_i by one immediately before each event it executes.
- (IT2) When Thread_i sends a message, it also sends the value of C_i as the timestamp for the send event.
- (IT3) When Thread_i receives a message with *ts* as its timestamp, if $ts \geq C_i$, then Thread_i sets C_i to *ts*+1 and assigns *ts*+1 as the timestamp for the receive event. Hence, $C_i = \max(C_i, ts+1)$.

Diagram (D3) in Fig. 6.11 represents an execution of three threads that use asynchronous communication.

Notice that the integer timestamp for event *v* is less than the integer timestamp for event *b*, but $v \Rightarrow b$ does not hold. (There is no path from *v* to *b* in diagram D3.)

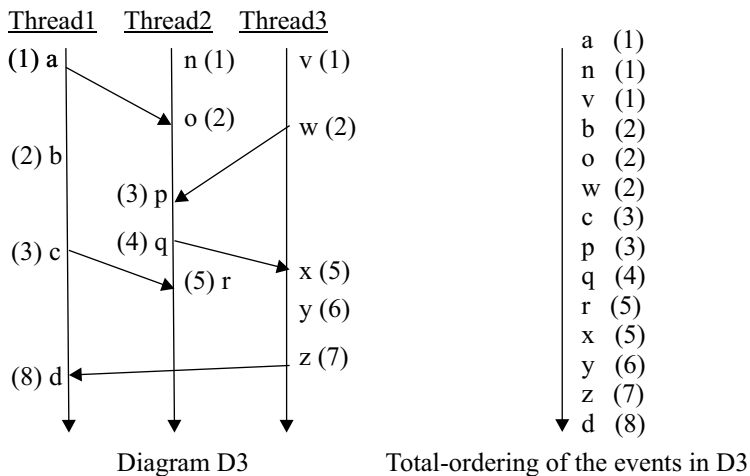


Figure 6.11 Diagram D3 and a total-ordering for D3.

Denote the integer timestamp recorded for event e as $IT(e)$ and let s and t be two events of an execution.

- If $s \Rightarrow t$, then $IT(s)$ will definitely be less than $IT(t)$.
- The fact that $IT(s)$ is less than $IT(t)$ does not imply that $s \Rightarrow t$.
- If s and t are concurrent, then their timestamps will be consistent with one of their two possible causal orderings.

\Rightarrow we cannot determine whether or not $s \Rightarrow t$ by using the integer timestamps recorded for s and t .

Although integer timestamps cannot tell us the causality relationships that hold between the events, we can use integer timestamps to produce one or more total orders that preserve the causal order:

- Order the events in ascending order of their integer timestamps. For the events that have the same integer timestamp, break the tie in some consistent way.
- A method for tie breaking: order events with the same integer timestamps in increasing order of their thread identifiers (refer again to Fig. 6.11).

From Table 6.1:

- integer timestamps avoid incorrect orderings
- but independent events are arbitrary ordered.

6.3.6 Vector Timestamps

Integer timestamps cannot be used to determine that two events are *not* causally related.

To do this, each thread maintains a vector clock, which is a vector of integer clock values. The vector clock for Thread_{*i*}, $1 \leq i \leq n$, is denoted as VC_i , where $VC_i[j]$, $1 \leq j \leq n$, refers to the *j*th element of vector clock VC_i .

During execution, vector time is maintained as follows:

- (VT1) Thread_{*i*} increments $VC_i[i]$ by one before each event of Thread_{*i*}.
- (VT2) When Thread_{*i*} executes a non-blocking send, it sends the value of its vector clock VC_i as the timestamp for the send operation.
- (VT3) When Thread_{*i*} receives a message with timestamp VT_m from a non-blocking send of another thread, Thread_{*i*} sets VC_i to the maximum of VC_i and VT_m and assigns VC_i as the timestamp for the receive event. Hence, $VC_i = \max(VC_i, VT_m)$. That is: for ($k = 1$; $k \leq n$; $k++$) $VC_i[k] = \max(VC_i[k], VT_m[k])$;
- (VT4) When one of Thread_{*i*} or Thread_{*j*} executes a blocking send that is received by the other, Thread_{*i*} and Thread_{*j*} exchange their vector clock values, set their vector clocks to the maximum of the two vector clock values, and assign their new vector clocks (which now have the same value) as the timestamps for the send and receive events. Hence, Thread_{*i*} performs the following operations:
 - Thread_{*i*} sends VC_i to Thread_{*j*} and receives VC_j from Thread_{*j*};
 - Thread_{*i*} sets $VC_i = \max(VC_i, VC_j)$;
 - Thread_{*i*} assigns VC_i as the timestamp for the send or receive event that Thread_{*i*} performed.

Thread_{*j*} performs similar operations.

The value $VC_i[j]$, where $j \neq i$, denotes the best estimate $Thread_i$ is able to make about $Thread_j$'s current logical clock value $VC_j[j]$, i.e., the number of events in $Thread_j$ that $Thread_i$ "knows about" through direct communication with $Thread_j$ or through communication with other threads that have communicated with $Thread_j$ and $Thread_i$.

Examples:

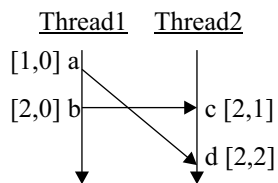


Figure 6.12 Diagram D4.

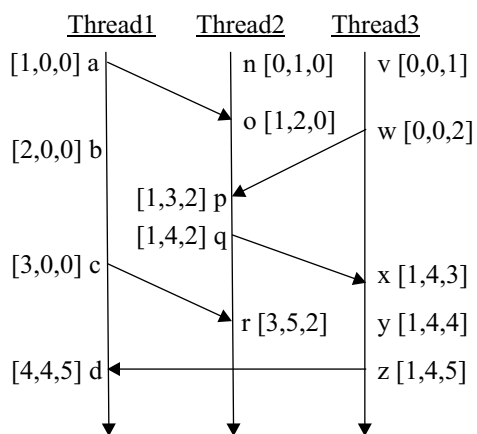


Figure 6.13 Diagram D5.

Rule HB1:

- Denote the vector timestamp recorded for event e as $VT(e)$.
- For a given execution, let e_i be an event in $Thread_i$ and e_j an event in (possibly the same thread) $Thread_j$.
- Threads are permitted to use asynchronous or synchronous communication, or a mix of both.
- $e_i \Rightarrow e_j$ if and only if there exists a path from e_i to e_j in the space-time diagram of the execution and there is no message $e_i \leftrightarrow e_j$ or $e_j \leftrightarrow e_i$. Thus:

(HB1) $e_i \Rightarrow e_j$ if and only if (for $1 \leq k \leq n$, $VT(e_i)[k] \leq VT(e_j)[k]$) and $(VT(e_i) \neq VT(e_j))$.

Note that if there is a message $e_i \leftrightarrow e_j$ or $e_j \leftrightarrow e_i$ then $VT(e_i) = VT(e_j)$ and (HB1) cannot be true.

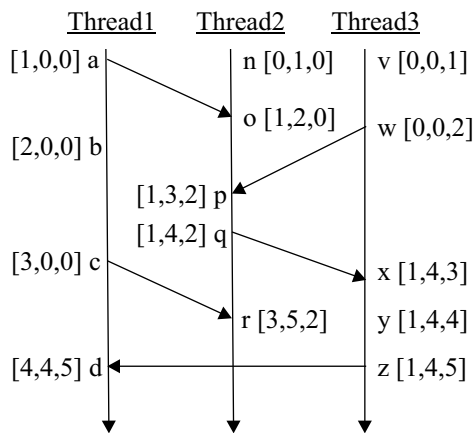


Figure 6.13 Diagram D5.

Rule HB2: Actually, we only need to compare two pairs of values, as the following rule shows:

$$(HB2) \ e_i \Rightarrow e_j \text{ if and only if } (VT(e_i)[i] \leq VT(e_j)[i]) \text{ and } (VT(e_i)[j] < VT(e_j)[j]).$$

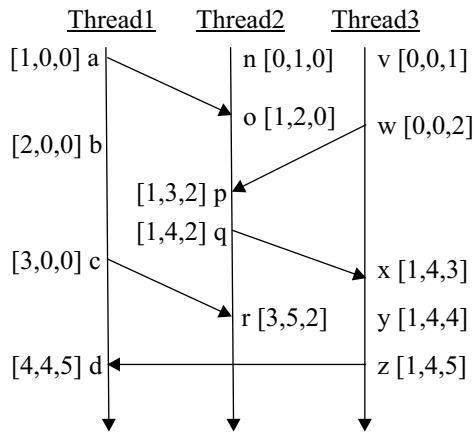


Figure 6.13 Diagram D5.

If there is a message $e_i \leftrightarrow e_j$ or $e_j \leftrightarrow e_i$ then $e_i \Rightarrow e_j$ is not true and $(VT(e_i)[j] < VT(e_j)[j])$ cannot be true (since the timestamps of e_i and e_j will be the same). This is also true if e_i and e_j are the same event.

In Fig. 6.13, events v and p are in Thread3 and Thread2, respectively, where $VT(v) = [0,0,1]$ and $VT(p) = [1,3,2]$:

- Since $(VT(v)[3] \leq VT(p)[3])$ and $(VT(v)[2] < VT(p)[2])$ we conclude $v \Rightarrow p$.
- For event w in Thread3 we have $VT(w) = [0,0,2]$. Since there is a message $w \leftrightarrow p$, the timestamps for w and p are the same, which means that $VT(v)[3] \leq VT(w)[3]$ must be true and that $VT(v)[2] < VT(w)[2]$ cannot be true.

Hence, $w \Rightarrow p$ is not true, as expected.

In general, suppose that the value of $VT(e_i)[j]$ is x and the value of $VT(e_j)[j]$ is y .

- the only way for $VT(e_i)[j] < VT(e_j)[j]$ to be false is if $Thread_i$ knows (through communication with $Thread_j$ or with other threads that have communicated with $Thread_j$) that $Thread_j$ already performed its x^{th} event, which was either e_j or an event that happened after e_j (as $x \geq y$).
- In either case, $e_i \Rightarrow e_j$ can't be true (otherwise, we would have $e_i \Rightarrow e_j \Rightarrow e_i$, which is impossible).

Rule HB3: If events e_i and e_j are in different threads and only asynchronous communication is used, then the rule can be further simplified to:

(HB3) $e_i \Rightarrow e_j$ if and only if $VT(e_i)[i] \leq VT(e_j)[i]$.

Example:

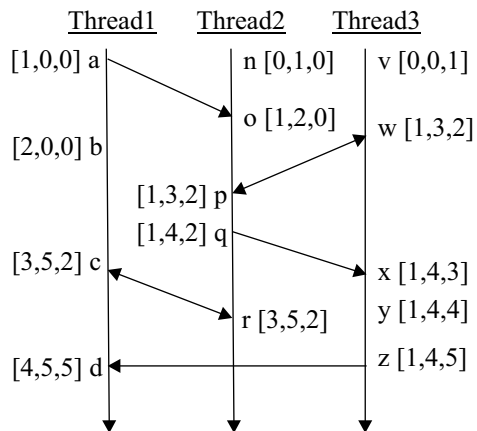


Figure 6.14 Diagram D6.

Referring to Table 6.1, using vector timestamps avoids incorrect orderings and independent events are not arbitrarily ordered.

6.3.7 Timestamps for Programs using Messages and Shared Variables

In programs that use both message passing and shared variable communication, vector timestamps must be assigned to send and receive events and also to events involving shared variables, such as read and write events or entering a monitor.

For two read/write events e and f on a shared variable V , let $e \xrightarrow{V} f$ denote that event e occurs before f on V .

(C1–C4) Same as rules C1 – C4 above for send and receive events.

(C5) For two different events e and f on shared variable V such that at least one of them is a write event, if $e \xrightarrow{V} f$ then $e \Rightarrow f$.

Timestamps for send and receive events are assigned as described earlier.

For each shared variable V , we maintain two vector timestamps:

- $VT_LastWrite(V)$: contains the vector timestamp of the last write event on V , initially all zeros.
- $VT_Current(V)$: is the current vector clock of V , initially all zeros.

When Thread_{*i*} performs an event *e*:

(VT1–VT4) If *e* is a send or receive event, same as (VT1–VT4) in Section 6.3.6

(VT5) If *e* is a write operation on shared variable *V*, Thread_{*i*} performs the following operations after performing write operation *e*:

$$(VT5.1) \quad VC_i = \max(VC_i, VT_Current(V))$$

$$(VT5.2) \quad VT_LastWrite(V) = VC_i$$

$$(VT5.3) \quad VT_Current(V) = VC_i$$

(VT6) If *e* is a read operation on shared variable *V*, Thread_{*i*} performs the following operations after performing read operation *e*:

$$(VT6.1) \quad VC_i = \max(VC_i, VT_LastWrite(V))$$

$$(VT6.2) \quad VT_Current(V) = \max(VC_i, VT_Current(V))$$

For write event *e* in rule (VT5), VC_i is set to $\max(VC_i, VT_Current(V))$.

For read event *e* in rule (VT6), VC_i is set to $\max(VC_i, VT_LastWrite(V))$.

The reason for the difference is the following.

- A write event on *V* causally precedes all the read and write events on *V* that follow it.
- A read event on *V* is concurrent with other read events on *V* that happen after the most recent write event on *V* and happen before the next write event, provided that there are no causal relations between these read events due to messages or accesses to other shared variables.

Example:

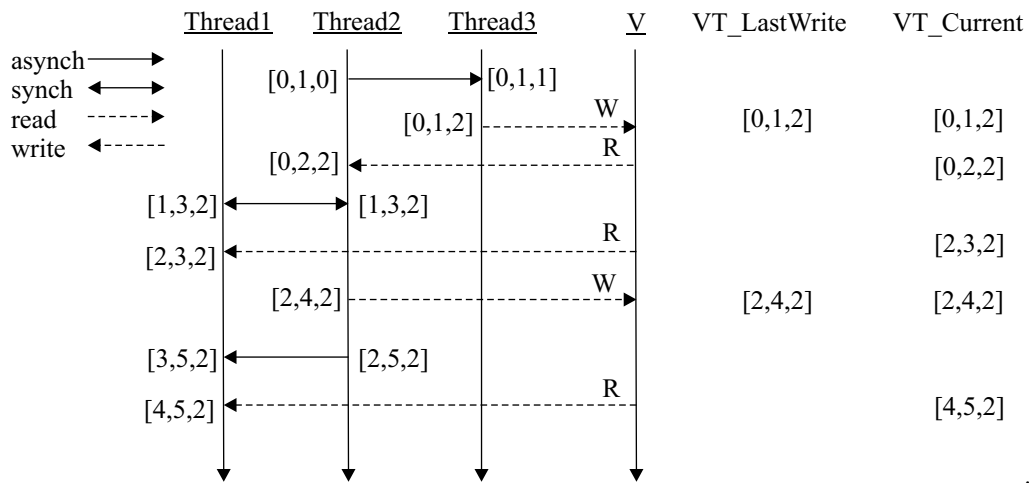


Figure 6.15 Timestamps for a program that uses message passing and shared variables.

6.4 Message-Based Solutions to Distributed Programming Problems

Distributed mutual exclusion and distributed readers and writers: Both problems involve events that must be totally-ordered, such as requests to read or write, or requests to enter a critical section. Requests are ordered using integer timestamps (Section 6.3.5).

Alternating bit protocol (ABP). The ABP is used to ensure the reliable transfer of data over faulty communication lines. Protocols similar to ABP are used by TCP.

6.4.1 Distributed Mutual Exclusion

When a process wishes to enter its critical section, it requests and waits for permission from all the other processes. When a process receives a request:

- if the process is not interested in entering its critical section, the process gives its permission by sending a reply as soon as it receives the request.
- if the process does want to enter its critical section, then it may defer its reply, depending on the relative order of its request among requests by other processes.

Requests are ordered based on a sequence number that is associated with each request and a sequence number maintained locally by each process:

- Sequence numbers are essentially integer timestamps.
- If a process receives a request having a sequence number that is the same as the local sequence number of the process, then the tie is resolved in favor of the process with the lowest ID.
- Since requests can be totally ordered using sequence numbers and IDs, there is always just one process that can enter its critical section next.

In program *distributedMutualExclusion* in Listing 6.16, each *distributedProcess* is started with a user assigned ID:

- assume there are three *distributedMutualExclusion* programs running at the same time, on the same computer.
- *numberOfProcesses* is three, and the IDs of the processes are 0, 1, and 2.

Each *distributedProcess* uses an array of three *TCPSender* objects for sending requests, and an array of three *TCPReceiver* objects for sending replies:

- When a *TCPSender* object is constructed, it is associated with the host address and port number of a *TCPMailbox* object owned by one of the distributed processes.
- All messages sent through the *TCPSender* object are addressed to the associated *TCPMailbox*.

Each *distributedProcess* uses *TCPMailbox* objects named *receiveRequests* and *receiveReplies* to receive request and reply messages from the other processes.

Connections between all the *TCPSender* and *TCPMailbox* objects are made by calling the *connect()* method of each *TCPSender* object at the start of the *run()* method for each *distributedProcess*.

The port numbers used for *TCPMailbox* objects *receiveRequests* and *receiveReplies* are as follows:

- *distributedProcess0* uses 2020 and 2021 for its two *TCPMailbox* objects
- *distributedProcess1* uses 2022 and 2023 for its two *TCPMailbox* objects
- *distributedProcess2* uses 2024 and 2025 for its two *TCPMailbox* objects.

Example:

- *distributedProcess0* sends requests ports 2022 and 2024
- Requests from the other two processes to *distributedProcess0* are addressed to port 2020

- Replies from the other processes to *distributedProcess0* are addressed to 2021
- Replies from *distributedProcess0* to *distributedProcess1* and *distributedProcess2* are addressed to ports 2023 and 2025, respectively.

When a *distributedProcess* wants to enter its critical section in method *run()* it:

- computes a sequence number
- sets flag *requestingOrExecuting* to true
- sends a request to each of the other processes
- waits for each of the processes to reply.

Each *distributedProcess* has a *Helper* thread that handles requests received from the other processes:

- If the *Helper* for *distributedProcess i* receives a *requestMessage* from *distributedProcess j*, the *Helper* replies immediately if
 - the sequence number in *j*'s request < the sequence number stored at *i*, or
 - if *distributedProcess i* is not trying to enter its critical section.
- The *Helper* defers the reply if *distributedProcess i* is in its critical section, or if *distributedProcess i* wants to enter its critical section and the *requestMessage* from *distributedProcess j* has a higher sequence number.
- If the sequence numbers are the same, then the tie is broken by comparing process identifiers. (Each request message contains a sequence number and the identifier of the sending process.)

When a *distributedProcess* sends its request, it computes a sequence number by adding one to the highest sequence number it has received in requests from other processes.

Class *Coordinator* is a monitor that synchronizes a *distributedProcess* thread and its *Helper* thread.

```

import java.io.*; import java.net.*;
class distributedMutualExclusion {
    public static void main(String[] args) {
        if (args.length != 1) {
            System.out.println("need 1 argument: process ID (IDs start with 0)");
            System.exit(1);
        }
        int ID = Integer.parseInt(args[0]); new distributedProcess(ID).start();
    }
}
class distributedProcess extends TDThreadD {
    private int ID; // process ID
    private int number; // the sequence number sent in request messages
    private int replyCount; // number of replies received so far
    final private int numberOfProcesses = 4;
    final private int basePort = 2020; String processHost = null;
    // requests sent to other processes
    private TCPSender[] sendRequests = new TCPSender[numberOfProcesses];
    private TCPSender[] sendReplies = new TCPSender[numberOfProcesses];
    private TCPMailbox receiveRequests = null;
    private TCPMailbox receiveReplies = null;
    private boolean[] deferred = null; // true means reply was deferred
    private Coordinator C; // monitor C coordinates distributedProcess and helper
    private Helper helper; // manages incoming requests for distributedProcess
    distributedProcess(int ID) {
        this.ID = ID;
        try {processHost = InetAddress.getLocalHost().getHostName(); }
        catch (UnknownHostException e) {e.printStackTrace();System.exit(1);}
        for (int i = 0; i < numberOfProcesses; i++) {
            sendRequests[i] = new TCPSender(processHost,basePort+(2*i));
            sendReplies[i] = new TCPSender(processHost,basePort+(2*i)+1);
        }
        receiveRequests = new TCPMailbox(basePort+(2*ID),"RequestsFor"+ID);
        receiveReplies = new TCPMailbox(basePort+(2*ID)+1,"RepliesFor"+ID);
        C = new Coordinator(); deferred = new boolean[numberOfProcesses];
        for (int i=0; i<numberOfProcesses; i++) deferred[i] = false;
        helper = new Helper(); helper.setDaemon(true); // start helper in run() method
    }
    class Helper extends TDThreadD {
        // manages requests from other distributed processes
        public void run() { // handle requests from other distributedProcesses
            while (true) {
                messageParts msg = (messageParts) receiveRequests.receive();
                requestMessage m = (requestMessage) msg.obj;

```

```

        if (!(C.deferrMessage(m))) { // if no deferral, then send a reply
            messageParts reply = new messageParts(new Integer(ID),
                processHost,basePort+(2*ID));
            sendReplies[m.ID].send(reply);
        }
    }
}
}
class Coordinator extends monitorSC {
// Synchronizes the distributed process and its helper
// requestingOrExecuting true if in or trying to enter CS
private boolean requestingOrExecuting = false;
private int highNumber; // highest sequence number seen so far
public boolean deferrMessage(requestMessage m) {
    enterMonitor("decideAboutDeferral");
    highNumber = Math.max(highNumber,m.number);
    boolean deferMessage = requestingOrExecuting &&
        ((number < m.number) ||| (number == m.number && ID < m.ID));
    if (deferMessage)
        deferred[m.ID] = true; // remember that the reply was deferred
    exitMonitor();
    return deferMessage;
}

public int chooseNumberAndSetRequesting() {
// choose sequence number and indicate process has entered or is
// requesting to enter critical section
    enterMonitor("chooseNumberAndSetRequesting");
    requestingOrExecuting = true;
    number = highNumber + 1; // get the next sequence number
    exitMonitor();
    return number;
}

public void resetRequesting() {
    enterMonitor("resetRequesting");
    requestingOrExecuting = false;
    exitMonitor();
}
}
}

```

```

public void run() {
    int count = 0;
    try {Thread.sleep(2000);} // give other processes time to start
    catch (InterruptedException e) {e.printStackTrace();System.exit(1);}
    System.out.println("Process " + ID + " starting");
    helper.start();
    for (int i = 0; i < numberOfProcesses; i++) {
    // connect to the mailboxes of the other processes
        if (i != ID) {sendRequests[i].connect(); sendReplies[i].connect();}
    }
    while (count++<3) {
        System.out.println(ID + " Before Critical Section"); System.out.flush();
        int number = C.chooseNumberAndSetRequesting();
        sendRequests(); waitforReplies();
        System.out.println(ID + " Leaving Critical Section-"+count); System.out.flush();
        try {Thread.sleep(500);} catch (InterruptedException e) {}
        C.resetRequesting();
        replytoDeferredProcesses();
    }
    try {Thread.sleep(10000);} // let other processes finish
    catch (InterruptedException e) {e.printStackTrace();System.exit(1);}
    for (int i = 0; i < numberOfProcesses; i++) // close connections
        if (i != ID) {sendRequests[i].close(); sendReplies[i].close();}
    }
public void sendRequests() {
    replyCount = 0;
    for (int i = 0; i < numberOfProcesses; i++) {
        if (i != ID) {
            messageParts msg = new messageParts(new requestMessage(number,ID));
            sendRequests[i].send(msg); // send sequence number and process ID
            System.out.println(ID + " sent request to Thread " + i);
            try {Thread.sleep(1000);} catch (InterruptedException e) {}
        }
    }
}
public void replytoDeferredProcesses() {
    System.out.println("replying to deferred processes");
    for (int i=0; i < numberOfProcesses; i++)
        if (deferred[i]) {
            deferred[i] = false; // ID sent as a convenience for identifying sender
            messageParts msg = new messageParts(new Integer(ID));
            sendReplies[i].send(msg);
        }
}
}

```

```

public void waitForReplies() { // wait for all the other processes to reply
    while (true) {
        messageParts m = (messageParts) receiveReplies.receive();
        // ID of replying thread is available but not needed
        int receivedID = ((Integer)m.obj).intValue();
        replyCount++;
        if (replyCount == numberOfProcesses-1)
            break; // all replies have been received
    }
}
}
}
class requestMessage implements Serializable {
    public int ID; // process ID
    public int number; // sequence number (integer timestamp)
    public requestMessage(int number, int ID) {
        this.ID = ID; this.number = number;
    }
}

```

Listing 6.16 Permission-based algorithm for distributed mutual exclusion.

6.4.2 Distributed Readers and Writers

We implement strategy R=W.2, which allows concurrent reading and gives readers and writers equal priority.

Mutual exclusion is provided using the permission-based distributed mutual exclusion algorithm described previously.

When a process wants to perform its read or write operation:

- It sends a request to each of the other processes and waits for replies.
- A request consists of the same pair of values (sequence number and ID) used in the mutual exclusion algorithm, along with a flag that indicates the type of operation (read or write) being requested.
- When process i receives a request from process j , process i sends j an immediate reply if:
 - process i is not executing or requesting to execute its read or write operation
 - process i is executing or requesting to execute a “compatible” operation. Two read operations are compatible, but two write operations, or a read and a write operation, are not compatible.
 - process i is also requesting to execute a non-compatible operation but process j 's request has priority over process i 's request.

The differences between *distributedReadersAndWriters* and program *distributedMutualExclusion* in Listing 6.16:

1. Each *distributedProcess* is either a reader or a writer.

```
java distributedReadersAndWriters 0 Reader // Reader process with ID 0
```

2. In method *decideAboutDeferral()* (Listing 6.17)

- The flag *requestingOrExecuting* is true when a *distributedProcess* is requesting to execute or is executing its read or write operation.
- The *requestMessage m* contains the type of *operation* being requested, and the sequence *number* and *ID* of the requesting process.
- If the receiving process is executing its operation or is requesting to execute its operation, method *compatible()* is used to determine whether or not the two operations are compatible.

Note: if process *i* is executing read or write operation OP when it receives a request *m* from process *j*, then the condition (*number* < *m.number*) in method *decideAboutDeferral()* must be true. That is:

- process *j* must have already received process *i*'s request for operation OP, sent its permission to *i*, and updated its sequence number to be higher than the number in process *i*'s request.
- In this case, if process *j*'s operation is not compatible, it will definitely be deferred.

```

public void decideAboutDeferral(requestMessage m) {
    enterMonitor();
    highNumber = Math.max(highNumber,m.number);
    boolean deferMessage = requestingOrExecuting &&
        !compatible(m.operation) && ((number < m.number) ||
            (number == m.number && ID < m.ID));
    if (deferMessage) {
        deferred[m.ID] = true;
    }
    else {
        messageParts msg = new messageParts(new Integer(ID));
        sendReplies[m.ID].send(msg);
    }
    exitMonitor();
}

private boolean compatible(int requestedOperation) {
    if (readerOrWriter == READER && requestedOperation == READER)
        return true; // only READER operations are compatible
    else // READER/WRITER and WRITER/WRITER operations incompatible
        return false;
}

```

Listing 6.17 Methods *decideAboutDeferral()* and *compatible()* in program *distributedReadersAndWriters*.

6.4.3 Alternating Bit Protocol

The Alternating Bit Protocol (ABP) is designed to ensure the reliable transfer of data over an unreliable communication medium.

The name of the protocol refers to the method used - messages are sent tagged with the bits 1 and 0 alternately, and these bits are also sent as acknowledgments.

Listing 6.18 shows classes *ABPSender* and *ABPReceiver* and two client threads.

- Thread *client1* is a source of messages for an *ABPSender* thread called *sender*.
- The *sender* receives messages from *client1* and sends the messages to an *ABPReceiver* thread called *receiver*.
- The *receiver* thread passes each message it receives to thread *client2*, which displays the message.

We assume that messages sent between an *ABPSender* and an *ABPReceiver* will not be corrupted, duplicated, or reordered, but they may be lost. The ABP will handle the detection and retransmission of lost messages.

An *ABPSender S* works as follows.

- After accepting a message from its client, *S* sends the message and sets a timer.
- *S* appends a one-bit sequence number (initially 1) to each message it sends out.
- There are then three possibilities:
 - *S* receives an acknowledgement from *ABPReceiver R* with the same sequence number. If this happens, the sequence number is incremented (modulo 2), and *S* is ready to accept the next message from its client.
 - *S* receives an acknowledgment with the wrong sequence number. In this case *S* resends the message (with the original sequence number), sets a timer, and waits for another acknowledgement from *R*.
 - *S* gets a timeout from the timer while waiting for an acknowledgement. In this case, *S* resends the message (with the original sequence number), sets a timer, and waits for an acknowledgement from *R*.

An *ABPReceiver R* receives a message and checks that the message has the expected sequence number (initially 1). There are two possibilities:

- *R* receives a message with a sequence number that matches the sequence number that *R* expects. If this happens,
 - *R* delivers the message to its client and sends an acknowledgement to *S*. The acknowledgment contains the same sequence number that *R* received.
 - *R* then increments the expected sequence number (modulo 2) and waits for the next message.
- *R* receives a message but the sequence number does not match the sequence number that *R* expects. In this case,
 - *R* sends *S* an acknowledgement that contains the sequence number that *R* received (i.e, the unexpected number),
 - *R* waits for *S* to resend the message.

Note that in both cases, the acknowledgement sent by *R* contains the sequence number that *R* received.

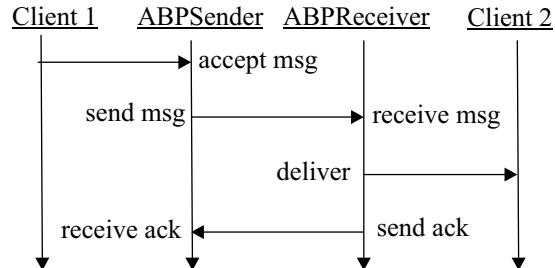
Communication between the sender and its client, and the receiver and its client, is through shared *link* (Chapter 5) channels.

The sender and receiver threads communicate using *TCPSender* and *TCPMailbox* objects, and a selectable asynchronous mailbox called *TCPSelectableMailbox*.

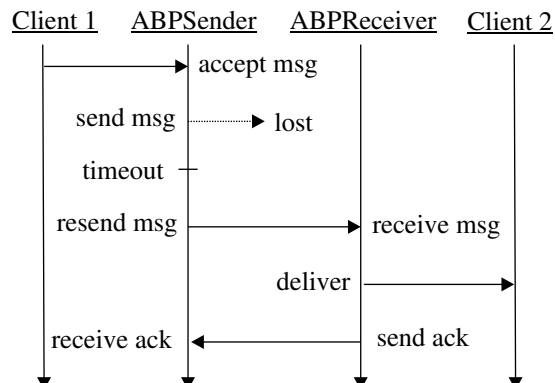
The *delayAlternative* in the *sender's* selective wait statement allows the *sender* to timeout while it is waiting to receive an acknowledgement from the *receiver*.

TCPUnreliableMailbox and *TCPUnreliableSelectableMailbox* randomly discard messages and acknowledgements sent between the *sender* and *receiver*.

Fig. 6.19 shows one possible flow of messages through the ABP program. In this scenario, no message or acknowledgement is lost.



In Fig. 6.20, the *ABPSender's* message is lost. In this case, the *ABPSender* receives a timeout from its timer and resends the message.



In Fig. 6.21, the *ABPSender's* message is not lost, but the *ABPSender* still receives a timeout before it receives an acknowledgement. In this case,

- the *ABPReceiver* delivers and acknowledges the first message but only acknowledges the second.
- The *ABPSender* receives two acknowledgements, but it ignores the second one.

