6.5 Testing and Debugging Distributed Programs

Outline:

- SYN-sequences for distributed Java programs that use classes *TCPSender* and *TCPMailbox*
- tracing, replay and feasibility

Let DP be a distributed program:

- DP consists of multiple Java programs,
- There are one or more Java programs running on each node in the system.
- Each Java program contains one or more threads.
- These threads use *TCPSender* and *TCPMailbox* objects to communicate with threads in other programs and possibly on other nodes.
- Threads in the same Java program can communicate and synchronize using shared variables and the channel, semaphore, or monitor objects presented in previous chapters and are traced, tested, and replayed using the techniques described previously.

6.5.1 Object-Based Sequences

There is one SYN-sequence for each synchronization object in the program.
The synchronization objects in program DP are its *TCPMailbox* objects.

The threads in DP execute CARC (*C*onect*A*rrival*R*eceive*C*lose) synchronization events of the following four types:

- *connection*: A connection is created between a *TCPSender* object and its associated *TCPMailbox* object by calling *connect()* on the *TCPSender*. (The host address and port number of a *TCPMailbox* is associated with a *TCPSender* object when the *TCPSender* object is constructed.)

- *arrival*: A message M is sent by a thread that calls operation *send(M)* on a *TCPSender* object. The arrival of message M at the corresponding *TCPMailbox* occurs some time after M it is sent. When message M arrives, it is queued in the message buffer of the *TCPMailbox* object.

- *receive*: a message is received by a thread that calls operation *receive()* on a *TCPMailbox* object. A *receive()* operation withdraws and returns a message from the message buffer of the *TCPMailbox* object.

- *close*: a connection between a *TCPSender* object and its associated *TCPMailbox* object is closed by calling *close()* on the *TCPSender* object.

Notice that there are *arrival* events but no *send* events. This is because it is the order in which messages arrive that determines the result of an execution, not the order in which messages are sent.

Note: several TCP message passing events occur during the execution of *connect, arrival, receive,* and *close events;* but these low-level events are hidden during tracing, testing, and reply.

The distributed bounded buffer program in Listing 6.22 is the program in Listing 6.3 with two minor modifications for tracing, testing, and replay:

- The *Producer*, *Buffer*, and *Consumer* programs use *TCPSender* and *TCPMailbox* objects named “deposit” and “withdraw” to pass messages.
- Classes *Producer*, *Buffer*, and *Consumer* inherit from class *TDThreadD,* which is class *TDThread* modified for distributed programs.

```
import java.net.*;
public final class Producer extends TDThreadD {
   public static void main (String args[]) {(new Producer()).start();}
   public void run() {
      final int bufferPort = 2020;
      String bufferHost = null;
      try {
         bufferHost = InetAddress.getLocalHost().getHostName();
         TCPSender deposit = new TCPSender(bufferHost,bufferPort,"deposit");
         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 extends TDThreadD {
   public static void main (String args[]) {(new Buffer()).start();}
   public void run() {
      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");
         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 extends TDThreadD {
   public static void main (String args[]) {  (new Consumer()).start();}
   public void run() {
      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();}
   }
}
```

Listing 6.22 The distributed bounded buffer program in Listing 6.4 modified for tracing, testing, and replay.

Fig. 6.23 shows a diagram of the threads in Listing 6.22. Fig. 6.24 shows part of a feasible sequence of *connect*, *arrival*, *receive*, and *close* events for this program.

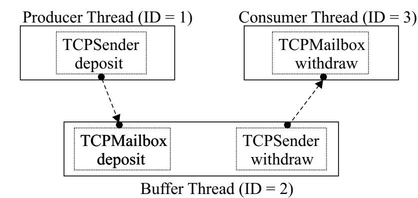

Figure 6.23 Threads and mailboxes in the distributed bounded buffer program.

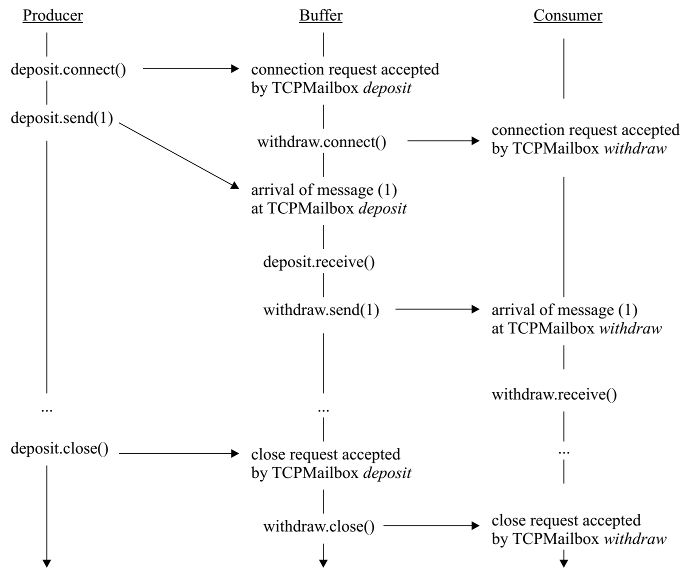

Figure 6.24 Sequence of *connect*, *arrival*, *receive*, and *close* events for the program in Listing 6.22.

An object-based CARC-event for a *TCPMailbox* M is denoted by

(threadID, orderNumber, eventType)

where:

- *threadID* is the ID of the thread that executed this event. This means that this thread sent a message that arrived at mailbox *M*, or opened or closed a connection with *M*, or received a message from *M*.
- *orderNumber* is the order number of the thread that executed this event, which gives the relative order of this event among all of the thread's CARC-events. The order number of an *arrival* event is the order number of the send operation that generated it.
- *eventType* is the type of this event, which is either *connection*, *arrival*, *receive*, or *close*.

Technically, there is no sending thread involved in a *receive()* operation:

- a *receive()* operation only withdraws a message from the message buffer of a *TCPMailbox* object, as opposed to actually receiving a message over the network.
- The message withdrawn by a *receive()* operation is one that arrived earlier.
- Thus, it was a previous *arrival* event that actually involved receiving the message from the sender and depositing the message into the message buffer.

The sending thread's ID and the vector timestamp generated for each event are helpful for understanding an execution and can be included in the trace information.

A CARC-sequence of *TCPMailbox M* is denoted by

M: (($threadID_1$, $orderNumber_1$, $eventType_1$),

($threadID_2$, $orderNumber_2$, $eventType_2$), ...)

where ($threadID_i$ $orderNumber_i$, $eventType_i$) denotes the ith, i>0, CARC-event in the CARC-sequence of M. An object-based CARC-sequence of a program contains a CARC-sequence for each *TCPMailbox* in the program.

For a *TCPSynchronousMailbox* object M, there are *connection* and *close* events as defined above, and a third event type:

- SRsynchronization: a synchronization between a *send()* and *receive()* operation involving M.

No *arrival* events are recorded for a *TCPSynchronousMailbox* object since the order of arrivals is implied by the order of *SRsynchronization* events.

An object-based *connection*, *SRsynchronization*, or *close* event (CSC-event) for *TCPSynchronousMailbox* M is denoted by

(callingThreadID, calledThreadID, callerOrderNumber, calledOrderNumber, eventType)

where:

- *callingThreadID* is the ID of the thread that opened or closed a connection with *M*, or the ID of the sending thread in an *SRsynchronization* at M.
- *calledThreadID* is the ID of the receiving thread in an *SRsynchronization* at *M*. There is no *calledThread* for *connection* or *close* events involving *M*.
- *callingOrderNumber* is the order number of the *callingThread*, which gives the relative order of this event among all of the *callingThread's* events.
- *calledOrderNumber* is the order number of the *calledThread*, which gives the relative order of this event among all of the *calledThread's* events. There is no *calledOrderNumber* for *connection* and *close* events involving *M*.
- *eventType* is the type of this event, which is either *connection*, *SRsynchronization*, or *close*.

We will use the value -1 for the *calledThread* and *calledOrderNumber* in *connection* and *close* events to indicate that the information about the called thread is not applicable.

6.5.1.1 Thread-Based Sequences.

Format 1: A thread-based CARC- or CSC-event for thread *T* is denoted by:

(mailbox, mailboxOrderNumber, eventType)

where:

- *mailbox* is the *TCPMailbox* name.
- *mailboxOrderNumber* is the order number of *mailbox*.
- *eventType* is the type of the event.

The order number of *M* gives the relative order of this event among all the events involving *M*. The list of event types is the same as that above for object-based events.

In Format 2, each *TCPMailbox* and *TCPSynchronousMailbox* has one receiving thread that is considered to be the "owner" of the mailbox:

- Since a mailbox can have multiple receiving threads, one of the receivers must be arbitrarily chosen to be the owner.
- If a thread *T* contains a selective wait, the selective wait will choose among one or more *TCPSelectableSynchronousMailboxes*, all of which we require to be owned by thread *T*.

We add a new event type for selective waits:

- *elseDelay*: selection of an *else* or *delay* alternative

Since thread T is associated with all of the mailbox and *elseDelay* events generated by its selective waits, we record these events in the sequence for thread *T*.

Format 2: A thread-based CARC- or CSC-event for thread T, where T is the owner of the mailbox associated with the event, is denoted by:

(sendingThreadID, sendingThreadOrderNumber, mailbox, eventType)

where:

- *sendingThreadID* is the ID of the sending thread (and T is always the receiver).
- *sendingThreadOrderNumber* is the sending thread's order number.
- *mailbox* is the name of the mailbox.
- *eventType* is the type of this event.

An *SRsynchronization* event representing a synchronization between send and receive operations on a synchronous mailbox appears in the CSC-sequence for the owning thread (i.e. the receiving thread). There is no corresponding send event for the sending thread.

In Format 2, no order number is required for mailbox *M* since the CARC- or CSC-sequence of M's owning thread *T* contains all of the events for mailbox *M* and thus the order numbers for *M* are given implicitly (i.e., the *i*th event for *M* implicitly has order number *i*.)

Given the two formats for thread-based CARC- and CSC-events, we can now define thread-based sequences for threads and for programs. For Format 1, a CARC- or CSC-sequence of thread *T* is denoted by

T:((mailbox$_1$, mailboxOrderNumber$_1$, eventType$_1$),
(mailbox$_2$, mailboxOrderNumber$_2$, eventType$_2$), ...).

For Format 2, a CARC- or CSC-sequence of thread T is denoted by

T: ((sendingThreadID$_1$, sendingThreadOrderNumber$_1$, mailbox$_1$, eventType$_1$),
(sendingThreadID$_2$, sendingThreadOrderNumber$_2$, mailbox$_2$, eventType$_2$), ...).

A thread-based CARC- or CSC-sequence of a program contains a thread-based CARC- or CSC-sequence for each thread in the program.

6.5.1.2 Totally-Ordered Sequences.

Tthere is only one sequence for the program and this sequence contains all of the events for all of the mailboxes and threads.

A CARC-event in a totally-ordered sequence is denoted by

(threadID, mailbox, eventType)

where:

- *mailbox* is the name of the *mailbox*. Each mailbox is associated with a name that is generated when it is constructed. (We assume users supply a name for each mailbox, or the name will be generated automatically as we have done for other objects.)
- *threadID* is the ID of the thread that executed this event. This means that this thread sent a message to the *mailbox*, or opened or closed a connection with the *mailbox*, or executed a *receive()* operation on the *mailbox*.
- *eventType* is the type of this event (either *connection*, *arrival*, *receive*, or *close*).

A CSC-event in a totally-ordered sequence is denoted by

(callingThreadID, calledThreadID, mailbox, eventType)

which requires both the calling and the called thread to be specified for *SRsynchronization* events.

A totally-ordered CARC- or CSC-sequence contains no order numbers for mailboxes or threads, since this information is specified implicitly by the ordering of events in the sequence.

A totally-ordered CARC-sequence of program DP is denoted as

DP: ((threadID$_1$, mailbox$_1$, eventType$_1$), (threadID$_1$, mailbox$_2$, eventType$_2$), ...)

A totally-ordered CSC-sequence of program DP is denoted as

DP: ((callingThreadID$_1$, calledThreadID$_1$, mailbox$_1$, eventType$_1$),
(callingThreadID$_2$, calledThreadID$_2$, mailbox$_2$, eventType$_2$), ...)

6.5.1.3 Node-Based Sequences.

To simplify the tracing and replay process, we use one sequence for each node in the system. There may be several programs running on a single node.

A totally-ordered CARC-sequence of node N is denoted by

N: ($eventType_1$, $threadID_1$, $mailbox_1$, $orderNumber_1$),

($eventType_2$, $threadID_2$, $mailbox_2$, $orderNumber_2$) …

where

- $threadID_i$ is the ID of the thread that executed the ith event
- $orderNumber_i$ is the order number of the thread that executed the ith event
- $mailbox_i$ is the name of the *TCPMailbox* involved in this event.

Mailbox $mailbox_i$ must be a mailbox created in a program on node N.

- only events in a CARC-sequence of node N are events that involve some *TCPMailbox* created on node N.
- If a thread running on node N is a calling thread for a *connection*, *arrival*, or *close* event, then that event appears in the CARC-sequence of the receiving node, not the CARC-sequence of N.
- Since the sender's order number is recorded in *connection*, *arrival*, and *close* events, the total order of the sending thread's events can be determined by analyzing the CARC-sequence(s) for all the nodes.

A totally-ordered CSC-sequence of node N is denoted by

N: (($eventType_1$,$callingThreadID_1$,$calledThreadID_1$,$callerOrderNumber_1$,$calledOrderNumber_1$,$mailbox_1$),

($eventType_2$,$callingThreadID_2$,$calledThreadID_2$,$callerOrderNumber_2$,$calledOrderNumber_2$,$mailbox_2$),

…)

A CARC- or CSC-sequence of a distributed program DP contains a CARC- or CSC-sequence for each node in the program.

Fig. 6.25 shows a feasible CARC-sequence of the distributed bounded buffer program in Listing 6.22. Since the *Producer*, *Buffer*, and *Consumer* programs are all executed on the same node, there is a single totally-ordered sequence of CARC-events.

```
(connection,2,withdraw,1)     // buffer connects to mailbox withdraw
(connection,1,deposit,1)      // producer connects to mailbox deposit
(arrival,1,deposit,2)         // producer's first message arrives at mailbox deposit
(receive,2,deposit,2)         // buffer receives producer's first message
(arrival,2,withdraw,3)        // buffer's first message arrives at mailbox withdraw
(receive,3,withdraw,1)        // consumer receives first message from withdraw
(arrival,1,deposit,3)         // producer's second msg arrives at mailbox deposit
(receive,2,deposit,4)         // buffer receives producer's second message
(arrival,1,deposit,4)         // producer's third message arrives at mailbox deposit
(arrival,2,withdraw,5)        // buffer's second message arrives at withdraw
(receive,2,deposit,6)         // buffer receives producer's third message
(close,1,deposit,5)           // producer closes connection with deposit
(arrival,2,withdraw,7)        // buffer's third message arrives at withdraw
(close,2,withdraw,8)          // buffer closes connection with withdraw
(receive,3 ,withdraw,2)       // consumer receives second message from withdraw
(receive,3,withdraw,3)        // consumer receives third message from withdraw
```

Figure 6.25 A CARC-sequence of the distributed bounded buffer program in Listing 6.19.

If the three programs were executed on three separate nodes, the CARC-sequence for the *Producer's* node would be empty, since the *Producer* is always the sending thread for the events it is involved with.

Handling exceptions:

- simulate the exceptions during replay, i.e., throw an exception whenever the trace file shows that an exception is needed.
- trace exceptions but not replay them.

6.5.2 Simple Sequences

During deterministic testing, we check the feasibility of a (complete) CARC- or CSC-sequence. Simpler sequences can be used for replay.

A simple CARC-event for a *TCPMailbox* is denoted as (threadID) where *threadID* is the ID of the thread that executed the CARC-event.

A simple CARC-sequence of node N is denoted by ($threadID_1$), ($threadID_2$), …
where the *TCPMailbox* for each event is some mailbox created on node N.

A simple CARC-sequence of a distributed program DP contains a simple CARC-sequence for each node in the program.

Fig. 6.26 shows the simple CARC-sequence corresponding to the complete CARC-sequence in Fig. 6.25.

(2)
(1)
(1)
(2)
(2)
(1)
(2)
(1)
(1)
(3)
(2)
(2)
(2)
(2)
(3)
(3)

Figure 6.26 Simple CARC-sequence of the distributed bounded buffer program.

For *TCPSynchronousMailboxes* and *TCPSelectableSynchronousMailboxes*, a simple CARC-event is denoted by

(callingThreadID, calledThreadID)

which identifies the IDs of the calling and called threads for the event.

For *connect* and *close* events, there is no called thread.

For *elseDelay* events, there is no calling thread. In these cases, we use -1 as the ID of the missing calling or called thread.

6.5.3 Tracing, Testing, and Replaying CARC-sequences and CSC-sequences

Before a thread can perform a *connect()*, *close()*, *send()*, or *receive()* operation, it must request permission from a control module.

The control module is responsible for reading a complete or simple CARC- or CSC-sequence and forcing the execution to proceed according to this sequence:

- Since we are tracing and replaying one sequence per node, we use one *Controller* per node.
- The *Controller* is a separate program that runs on the node along with the programs being tested and debugged.

6.5.3.1 Modifying Classes *TCPSender* and *TCPMailbox*.

Methods *connect()*, *send()*, *close()*, and *receive()* in classes *TCPSender* and *TCPMailbox* are modified by adding one or more calls to the *Controller*. In replay and test mode, methods *connect()*, *send()* and *close()* issue a call to the *Controller* to request permission to exercise a *connection*, *arrival*, or *close* event, respectively.

Methods *connect()*, *send()* and *close()* do not trace any *connection*, *arrival*, or *close* events during trace mode. Instead, these events are traced by the destination *TCPMailbox*.

6.5.3.2 Modifying Class *TCPSynchronousMailbox*

Since there are no *arrival* events in CSC-sequences, no *arrival* events are traced and *arrival* events are ignored during testing and replay.

6.5.3.3 The Controller Program.

The design of the *Controller* program is almost identical to the *Controller* in Chapter 5.

The major difference is that the *Controller* uses vector timestamps to ensure that the CARC- or CSC-sequence it records in trace mode is consistent with the causal ordering of events. (The recording algorithm is given in Section 6.3.6.)

To handle communications with the user programs the *Controller* use four *TCPSelectableMailboxes*:

- These mailboxes receive trace events, requests for permission to execute events, acknowledgement events, etc
- The ports used by these mailboxes are computed as offsets to a base port (basePort+0, basePort+1, etc), where the base port is hard coded as port 4040.

The *Controller* maintains an open connection between itself and each of the other nodes in the system.

- All the threads on a given node share that node's connection with the *Controller* and use the connection to send messages to the *Controller*.
- During replay and testing, each node uses a *TCPMailbox* to receive replies from the *Controller*. All the threads on a node share this *TCPMailbox*.
- By sharing connections, the number of connections between each node and the *Controller* is limited to one connection for tracing, three connections for replay, and four connections for checking feasibility.

6.5.4 Putting it all Together

To trace, test and debug executions, use the *mode* property to specify the function (tracing, testing or replay) to be performed.

An execution of the distributed mutual exclusion (DME) program in Listing 6.16 is traced as follows.

First, execute the *Controller* program. The *Controller* will start and then immediately wait for trace events to arrive.

Next, start each of the four user programs. The commands to run the *Controller* and the DME programs on a single node are shown below. (Run each program in a separate window.)

```
java -Dmode=trace -DprocessName=Node1 Controller
java -Dmode=trace -DstartID=1 -DprocessName=process0 distributedMutualExclusion0
java -Dmode=trace -DstartID=5 -DprocessName=process1 distributedMutualExclusion1
java -Dmode=trace -DstartID=9 -DprocessName=process2 distributedMutualExclusion2
java -Dmode=trace -DstartID=13 -DprocessName=process3
  distributedMutualExclusion3
```

The *processName* property specifies a unique name for each DME program. This name is used as a prefix for all the names and files generated for that program.

The *startID* property specifies the starting value for the identifier's to be generated for the threads in this program:

Each of the DME programs requires four thread IDs.

- One ID is required for the main thread and one for its helper.
- There is also one ID reserved for each of the two *TCPMailbox* objects.

This ensures that space is allocated for four objects in the vector timestamps.

After ten seconds of inactivity, the *Controller* will assume that the execution has stopped and terminate itself.

In *trace* mode, the *Controller* creates several trace files:

- Node1_channel-replay.txt contains a simple, totally-ordered CARC-sequence.
- Node1_channel-test.txt contains a totally-ordered CARC-sequence.
- Node1_channel-trace.txt contains a trace of additional information, such as the value of the vector timestamp for each event.

Since each DME program uses a monitor to synchronize the communication between the *main* thread and its *helper* thread, trace files are also generated for the monitor events as explained in Chapter 4.

The trace files generated for the monitor in the program named "process1" are:

- process1_monitor-replay.txt // simple M-sequence of the monitor in process1
- process1_monitor-test.txt // M-sequence of the monitor in process1
- process1_monitorID.txt // monitor IDs for process1
- process1_ThreadID.txt // thread IDs for process 1

A similar set of four files is produced for the monitors in the other three programs.

A simple SR-sequence is replayed by running the *Controller* and each user program in *replay* mode:

```
java -Dmode=replay -DprocessName=Node1 Controller
java -Dmode=replay -DstartID=1 -DprocessName=process0 -DackPort=3030
    distributedMutualExclusion0
java -Dmode=replay -DstartID=5 -DprocessName=process1 -DackPort=3032
    distributedMutualExclusion1
java -Dmode=replay -DstartID=9 -DprocessName=process2 -DackPort=3034
    distributedMutualExclusion2
java -Dmode=replay -DstartID=13 -DprocessName=process3 -DackPort=3036
    distributedMutualExclusion3
```

The *processNames* and *startIDs* must be the same ones used during tracing.

The *ackPort* property specifies a port number that the user program can use to receive messages from the *Controller* program. Each program should use a unique *ackPort*.

The *Controller* will read the simple CARC-sequence in file *Node1_channel-replay.txt* and force this sequence to occur during execution. The simple M-sequences of the monitors are replayed as described in Chapter 4.

The feasibility of a CARC-sequence is determined by running the *Controller* and each user program in *test* mode:

```
java -Dmode=test -DprocessName=Node1 Controller
java -Dmode=test -DstartID=1 -DprocessName=process0 -DackPort=3030
    distributedMutualExclusion0
java -Dmode=test -DstartID=5 -DprocessName=process1 -DackPort=3032
    distributedMutualExclusion1
java -Dmode=test -DstartID=9 -DprocessName=process2 -DackPort=3034
    distributedMutualExclusion2
java -Dmode=test -DstartID=13 -DprocessName=process3 -DackPort=3036
    distributedMutualExclusion3
```

The *Controller* will read the CARC-sequence in file *Node1_channel-test.txt* and attempt to force this sequence to occur during execution. If the CARC-sequence is infeasible, the *Controller* will issue a diagnostic and terminate.

The feasibility of the M-sequence is determined as described in Chapter 4.

If the mode is not specified, the default value for the *mode* property will turn tracing, replay, and testing off.

Fig. 6.27a shows a CARC-sequence of the DME program in which each process enters its critical section once. We traced an execution and then grouped, reordered, and commented the events to make the sequence easier to understand. The M-sequence for the monitor in *process1* is shown in Fig. 6.27b.

// Connect events issued by p0:
(connect,1,RequestsFor1,1), (connect,1,RepliesFor1,2), (connect,1,RequestsFor2,3),
(connect,1,RepliesFor2,4), (connect,1,RequestsFor3,5), (connect,1,RepliesFor3,6),
// Connect events issued by p1:
(connect,5,RequestsFor0,1), (connect,5,RepliesFor0,2), (connect,5,RequestsFor2,3),
(connect,5,RepliesFor2,4), (connect,5,RequestsFor3,5), (connect,5,RepliesFor3,6),
// Connect events issued by p2:
(connect,9,RequestsFor0,1), (connect,9,RepliesFor0,2), (connect,9,RequestsFor1,3),
(connect,9,RepliesFor1,4)
(connect,9,RequestsFor3,5), (connect,9,RepliesFor3,6),
// Connect events issued by p3:
(connect,13,RequestsFor0,1),(connect,13,RepliesFor0,2),
(connect,13,RequestsFor1,3),
(connect,13,RepliesFor1,4),(connect,13,RequestsFor2,5), (connect,13,RepliesFor2,6),

(arrival,5,RequestsFor0,8), (arrival,9,RequestsFor0,8), (arrival,13,RequestsFor0,8)
// All three requests have arrived at p0.

(arrival,1,RequestsFor1,8), (arrival,9,RequestsFor1,9), (arrival,13,RequestsFor1,9)
// All three requests have arrived at p1.

(arrival,1,RequestsFor2,9), (arrival,5,RequestsFor2,9), (arrival,13,RequestsFor2,10)
// All three requests have arrived at p2.

(arrival,1,RequestsFor3,10), (arrival,5,RequestsFor3,10),(arrival,9,RequestsFor3,10)
// All three requests have arrived at p3.
// Now each process has had a request arrive from all the other processes.

(receive,8,RequestsFor1,1) // p1's helper receives p0's request and replies
(arrival,8,RepliesFor0,3) // p1's helper's reply arrives at p0
(receive,12,RequestsFor2,1)// p2's helper receives p0's request and replies
(arrival,12,RepliesFor0,3) // p2's helper's reply arrives at p0
(receive,16,RequestsFor3,1)// p3's helper receives p0's request and replies
(arrival,16,RepliesFor0,3) // p3's helper's reply arrives at p0
// Replies from all the processes have arrived at p0.

(receive,12,RequestsFor2,4)// p2's helper receives p1's request and replies
(arrival,12,RepliesFor1,6) // p2's helper's reply arrives at p1
(receive,16,RequestsFor3,4)// p3's helper receives p1's request and replies
(arrival,16,RepliesFor1,6) // p3's helper's reply arrives at p1
// Replies from p2 and p3 have arrived at p1.

(receive,16,RequestsFor3,7)// p3's helper receives p2's request and replies
(arrival,16,RepliesFor2,9) // p3's helper's reply arrives at p2
// Reply from p3 has arrived at p2

(receive,4,RequestsFor0,1) // p0 receives requests from p1, p2, and p3
(receive,4,RequestsFor0,3) // but defers all replies since p0 has priority
(receive,4,RequestsFor0,5)

(receive,1,RepliesFor0,11) // p0 receives replies from p1, p2, and p3
(receive,1,RepliesFor0,12) // and enters/exits its critical section
(receive,1,RepliesFor0,13)

(arrival,1,RepliesFor1,15) // p0 sends all of its deferred replies
(arrival,1,RepliesFor2,16)
(arrival,1,RepliesFor3,17)

(receive,8,RequestsFor1,4) // p1's helper receives requests from p2 and p3 but
(receive,8,RequestsFor1,6) // defers replies since p1 has priority over p2 and p3

(receive,12,RequestsFor2,7)// p2's helper receives rqst from p3, but defers reply

(receive,5,RepliesFor1,11) // p1 receives all replies and enters/exits CS
(receive,5,RepliesFor1,12)
(receive,5,RepliesFor1,13)

(arrival,5,RepliesFor2,15) // p1 sends its deferred replies to p2 and p3
(arrival,5,RepliesFor3,16)

(receive,9,RepliesFor2,11) // p2 receives all replies and enters/exits CS
(receive,9,RepliesFor2,12)
(receive,9,RepliesFor2,13)

(arrival,9,RepliesFor3,15) // p2 sends its deferred reply to p3

(receive,13,RepliesFor3,11)// p3 receives all replies and enters/exits CS
(receive,13,RepliesFor3,12)
(receive,13,RepliesFor3,13)

// Close events issued by p0:
(close,1,RequestsFor1,18), (close,1,RepliesFor1,19), (close,1,RequestsFor2,20),
(close,1,RepliesFor2,21), (close,1,RequestsFor3,22), (close,1,RepliesFor3,23),
// Close events issued by p1:
(close,5,RequestsFor0,17), (close,5,RepliesFor0,18), (close,5,RequestsFor2,19),
(close,5,RepliesFor2,20), (close,5,RequestsFor3,21), (close,5,RepliesFor3,22),
// Close events issued by p2:
(close,9,RequestsFor0,16), (close,9,RepliesFor0,17), (close,9,RequestsFor1,18),

(close,9,RepliesFor1,19), (close,9,RequestsFor3,20), (close,9,RepliesFor3,21),
// Close events issued by p3:
(close,13,RequestsFor0,15), (close,13,RepliesFor0,16), (close,13,RequestsFor1,17),
(close,13,RepliesFor1,18), (close,13,RequestsFor2,19), (close,13,RepliesFor2,20).

Figure 6.27(a) A Feasible CARC-sequence for the distributed mutual exclusion program in Listing 6.19.

(entry,5,Coordinator:chooseNumberAndSetRequesting,NA)
(exit,5,Coordinator:chooseNumberAndSetRequesting,NA)
(entry,8,Coordinator:decideAboutDeferral,NA)
(exit,8,Coordinator:decideAboutDeferral,NA)
(entry,8,Coordinator:decideAboutDeferral,NA)
(exit,8,Coordinator:decideAboutDeferral,NA)
(entry,8,Coordinator:decideAboutDeferral,NA)
(exit,8,Coordinator:decideAboutDeferral,NA)
(entry,5,Coordinator:resetRequesting,NA)
(exit,5,Coordinator:resetRequesting,NA)

Figure 6.27(b) A feasible M-sequence for the monitor of *process1*.

6.5.5 Other Approaches to Replaying Distributed Programs

Multithreaded and distributed Java applications can be replayed by extending the Java Virtual Machine (JVM).

Distributed DejaVu [Konuru et al. 2000] uses a modified Java Virtual Machine that records and replays non-deterministic network operations in the Java TCP and UDP Socket API.

Some of these operations are: *accept*, *connect*, *read*, *write*, *available*, *bind*, *listen*, and *close*.

Multithreaded and distributed Java applications can be traced and replayed using Distributed DejaVu without any modifications to the source code.

Distributed DejaVu does not try to address the feasibility problem. It would be difficult for users to specify test sequences in DejaVu format since DejaVu traces contain low-level implementation details about the network operations.

By contrast, the CARC-sequences used by the mailbox classes are at a higher-level of abstraction.

Combining the precise and transparent control that is enabled at the JVM level with the expressiveness of working at the source-code level is probably the best approach.