Carvalho, Guerraoui, Rodrigues, and others

Revised Hands-On Sections

from the 1st edition of the book “Introduction to Reliable Distributed Programming” by Guerraoui and Rodrigues

March 2011

Berlin Heidelberg
2006

Preface

This document provides a revised collection of the Hands-On sections from the first edition of the book Introduction to Reliable Distributed Programming (Guerraoui and Rodrigues 2006).

The first edition of the book included a companion set of running examples implemented in the Java programming language, using the *Appia* protocol composition framework. These examples can be used by students to get a better understanding of many implementation details that are not covered in the high-level description of the algorithms given in the core of the chapters. Understanding such details usually makes a big difference when moving to a practical environment. Instructors can use the protocol layers as a basis for practical experimentations, by suggesting to students to perform optimizations of the protocols already given in the framework, to implement variations of these protocols for different system models, or to develop application prototypes that make use of the protocols. As a result, each chapter of the 1st edition included a section with a brief description of some of these implementation. These are the sections that are now collected in this volume. Note that the implementation of the algorithms may be slightly different from the algorithms presented on the second edition of the book.

Several implementations for the "hands-on" part of the book were developed by, or with the help of, Alexandre Pinto, a key member of the *Appia* team, complemented with inputs from several students, including Nuno Carvalho, Maria João Monteiro, and Luís Sardinha.

For this edition, Nuno Carvalho ported the original Java code to work with the latest *Appia* release (4.1.2), and updated the hands-on sections from the original book accordingly. The Java code that matches this document can be retrieved from the web page (`www.distributedprogramming.net`) for the 2nd edition of the book (Cachin, Guerraoui, and Rodrigues 2011). The *Appia* code is hosted at SourceForge (please check the Appia page for latest updates).

Nuno Carvalho, Rachid Guerraoui and Luís Rodrigues

Contents

1. Introduction

Several of the algorithms that we will be presenting in the book have been implemented and made available as open source code. By using these implementations, the reader has the opportunity to run and experiment with the algorithms in a real setting, review the code, make changes and improvements to it and, eventually, take it as a basis to implement her own algorithms. Note that, when referring to the implementation of an algorithm, we will mainly use the word *protocol*. As noted before, the protocol describes not only the behavior of each participant but also the concrete format of the messages exchanged among participants.

The algorithms have been implemented in the Java programming language with the support of the *Appia* protocol composition and execution framework (Miranda, Pinto, and Rodrigues 2001). *Appia* is a tool that simplifies the development of communication protocols. To start with, *Appia* already implements a number of basic services that are required in several protocols, such as methods to add and extract headers from messages or launch timers: these are the sort of implementation details that may require the writing of a considerable number of lines of code when appropriate libraries are not available. Additionally, *Appia* simplifies the task of composing different protocol modules.

Central to the use of *Appia* is the notion of *protocol composition*. In its simpler form, a protocol composition is a stack of instances of the Layer class. For each different protocol module, a different specialization of the Layer class should be defined. In *Appia*, multiple instances of a given protocol composition may be created at run-time. Each instance is called a *channel*, as it is materialized as a stack of objects of the Session class. In *Appia*, modules communicate through the exchange of *events*. *Appia* defines the class Event, from which all events exchanged in the *Appia* framework must be derived. In order for a module to consume and produce events, a layer must explicitly declare the set of events *accepted*, *provided*, and *required*. When a layer requires an event, *Appia* checks if there is another layer in the composition that provides that event; if not, it generates an exception. This offers a simple way of detecting inconsistencies in the protocol composition.

At this point, it is probably useful to clarify the mapping between the abstract descriptions provided in the book and the corresponding concrete implementation in the *Appia* framework.

- While in the book we use pseudo code to describe the algorithms, in *Appia* the Java programming language is used to implement them.
- In the book we use the module concept to characterize the interface of a service. In the *Appia* implementation this interface is captured in the Layer class. The *Requests* accepted by the module are listed as *accepted* events. The *Indications* and *Confirmations* (if any) provided by the module are listed as *provided* events.
- Typically, there is a projection of the pseudo code that appears in an algorithm and the Java code of the corresponding Session class.

1.1 Print Module

Consider, for instance, the implementation of the Print module. First, we define the events accepted and provided by this module. This is illustrated in Listing 1.1.

Listing 1.1. Events for the Print module

```java
package irdp.protocols.tutorialDA.print;

public class PrintRequestEvent extends Event {
    int rqid;
    String str;

    void setId(int rid);
    void setString(String s);
    int getId();
    String getString();
}

public class PrintConfirmEvent extends Event {
    int rqid;

    void setId(int rid);
    int getId();
}
```

Then, we implement the layer for this module. This is illustrated in Listing 1.2. As expected, the layer accepts the PrintRequestEvent and provides the PrintConfirmEvent. The PrintLayer is also responsible for creating objects of class PrintSession, whose purpose is described in the next paragraphs.

Listing 1.2. PrintLayer

```java
package irdp.protocols.tutorialDA.print;

public class PrintLayer extends Layer {
  public PrintLayer(){
    /* events that the protocol will create */
    evProvide = new Class[1];
    evProvide[0] = PrintConfirmEvent.class;

    /* events that the protocol requires to work.  This is
     * a subset of the accepted events */
    evRequire = new Class[0];

    /* events that the protocol will accept */
    evAccept = new Class[2];
    evAccept[0] = PrintRequestEvent.class;
    evAccept[1] = ChannelInit.class;
  }

  public Session createSession() {
    return new PrintSession(this);
  }

}
```

Layers are used to describe the behavior of each module. The actual methods and the state required by the algorithm is maintained by Session objects. Thus, for every layer, the programmer needs to define the corresponding session. The main method of a session is the handle method, invoked by the *Appia* kernel whenever there is an event to be processed by the session. For the case of our Print module, the implementation of the PrintSession is given in Listing 1.3.

Listing 1.3. PrintSession

```
package irdp.protocols.tutorialDA.print;

public class PrintSession extends Session {

    public PrintSession(Layer layer) {
        super(layer);
    }

    public void handle(Event event){
        if(event instanceof ChannelInit)
            handleChannelInit((ChannelInit)event);
        else if(event instanceof PrintRequestEvent){
            handlePrintRequest ((PrintRequestEvent)event);
        }
    }

    private void handleChannelInit(ChannelInit init) {
        try {
            init.go();
        } catch (AppiaEventException e) {
            e.printStackTrace();
        }
    }

    private void handlePrintRequest(PrintRequestEvent request) {
        try {
            PrintConfirmEvent ack = new PrintConfirmEvent ();

            doPrint (request.getString ());
            request.go();
            ack.setChannel(request.getChannel());
            ack.setDir(Direction.UP);
            ack.setSourceSession(this);
            ack.setId(request.getId());
            ack.init();
            ack.go();
        } catch (AppiaEventException e) {
            e.printStackTrace();
        }
    }
}
```

There are a couple of issues the reader should note in the previous code. First, as in most of our algorithms, every session should be ready to accept the ChannelInit event. This event is automatically generated and should be used to initialize the session state. Second, in *Appia*, the default behavior for a session is to always propagate downward (or upward) in the stack the events it consumes. As it will become clear later in the book, it is often very convenient to have the same event processed by different sessions in sequence.

1.2 BoundedPrint Module

Having defined the events, the layer, and the session for the Print module, we can now perform a similar job for the BoundedPrint module. As before, we start by providing the required events, as depicted in Listing 1.4. Note that we define the PrintAlarmEvent and the PrintStatusEvent.

We now use the opportunity to reinforce a very important feature of *Appia*. In *Appia*, the same event may be processed by several sessions in sequence. Therefore, *Appia* programmers avoid renaming events when they perform similar functions in different layers. In this case, instead of creating a new event BoundedPrintRequestEvent, that would have to be renamed PrintRequestEvent when propagated from the BoundedPrintLayer to the PrintLayer, we simply use the same event, PrintRequestEvent, in both layers. The order in which this event is processed (i.e, the fact that it is first processed by the BoundedPrintLayer and afterward by the PrintLayer) is defined by the composition of the stack, when the *Appia QoS* is declared.

Listing 1.4. Events for the BoundedPrint module

```
package irdp.protocols.tutorialDA.print;

class PrintAlarmEvent extends Event {
}

class PrintStatusEvent extends Event {
    int      r_id;
    Status   stat;

    void setId (int rid);
    void setStatus (Status s);
    int getId ();
    int getStatus ();
}
```

We proceed to define the BoundedPrintLayer, as depicted in Listing 1.5. Since the BoundedPrint module uses the services of the basic Print module, it requires the PrintConfirmEvent produced by that module.

Listing 1.5. Bounded PrintLayer

```
package irdp.protocols.tutorialDA.print;

public class BoundedPrintLayer extends Layer {

  public BoundedPrintLayer(){
    /* events that the protocol will create */
    evProvide = new Class[2];
    evProvide[0] = PrintStatusEvent.class;
    evProvide[1] = PrintAlarmEvent.class;

    /* events that the protocol require to work.
     * This is a subset of the accepted events */
    evRequire = new Class[1];
    evRequire[0] = PrintConfirmEvent.class;

    /* events that the protocol will accept */
    evAccept = new Class[3];
    evAccept[0] = PrintRequestEvent.class;
    evAccept[1] = PrintConfirmEvent.class;
    evAccept[2] = ChannelInit.class;
  }

  public Session createSession() {
    return new BoundedPrintSession(this);
  }

}
```

Subsequently, we can implement the session for the BoundedPrint module, depicted in Listing 1.6.

Listing 1.6. BoundedPrintSession

```
package irdp.protocols.tutorialDA.print;

public class BoundedPrintSession extends Session {
    int bound;

    public BoundedPrintSession(Layer layer) {
        super(layer);
    }

    public void handle(Event event){
        if(event instanceof ChannelInit) {
            handleChannelInit((ChannelInit)event);
        }
        else if(event instanceof PrintRequestEvent) {
            handlePrintRequest ((PrintRequestEvent)event);
        }
        else if(event instanceof PrintConfirmEvent) {
            handlePrintConfirm ((PrintConfirmEvent)event);
        }
    }

    private void handleChannelInit(ChannelInit init) {
        try {
            bound = PredefinedThreshold;

            init .go();
        } catch (AppiaEventException e) {
            e.printStackTrace();
        }
    }

    private void handlePrintRequest(PrintRequestEvent request) {
        if (bound > 0){
            bound = bound −1;
            try {
                request.go ();
            } catch (AppiaEventException e) {
                e.printStackTrace();
            }
            if (bound == 0) {
                PrintAlarmEvent alarm = new PrintAlarmEvent ();
                alarm.setChannel (request.getChannel());
                alarm.setSourceSession (this);
                alarm.setDir(Direction.UP);
                try {
                    alarm.init ();
                    alarm.go ();
                } catch (AppiaEventException e) {
                    e.printStackTrace();
                }
            }
        }
        else {
            PrintStatusEvent status = new PrintStatusEvent ();
            status.setChannel (request.getChannel());
            status.setSourceSession (this);
            status.setDir(Direction.UP);
            status.setId (request.getId ());
            status.setStatus (Status.NOK);
            try {
                status. init ();
                status.go ();
            } catch (AppiaEventException e) {
                e.printStackTrace();
            }
        }
    }

    private void handlePrintConfirm(PrintConfirmEvent conf) {
        PrintStatusEvent status = new PrintStatusEvent ();
        status.setChannel (request.getChannel());
        status.setSourceSession (this);
        status.setDir(Direction.UP);
        status.setId (conf.getId ());
        status.setStatus (Status.OK);
        try {
            status. init ();
            status.go ();
        } catch (AppiaEventException e) {
            e.printStackTrace();
        }
    }
}
```

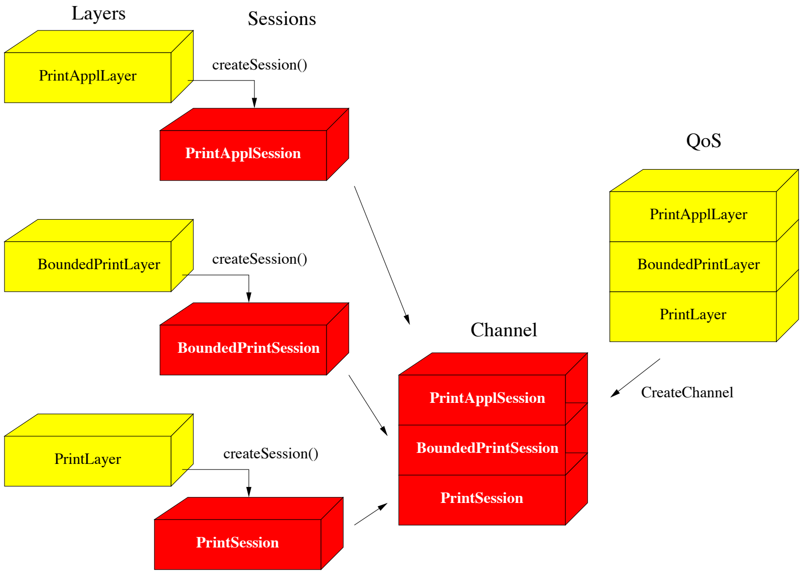

Fig. 1.1: Layers, Sessions, QoS and Channels

1.3 Composing Modules

The two modules that we have described can now be easily composed using the *Appia* framework. The first step consists in creating a protocol composition by stacking the BoundedPrintLayer on top of the PrintLayer. Actually, in order to be able to experiment with these two layers, we further add on top of the stack a simple application layer, named PrintApplicationLayer. This is a simple layer, that listens for strings on *standard input*, creates and sends a print request with those strings, and displays the confirmation and status events received.

A composition of layers in *Appia* is called a QoS (Quality of Service) and can simply be created by providing the desired array of layers, as shown in Listing 1.7. After defining a protocol composition, it is possible to create one or more communication *channels* that use that composition. Therefore, channels can be seen as instances of protocol compositions. Channels are made of *sessions*. When a channel is created from a composition, it is possible to automatically create a new session for every layer in the composition. The relation between Layers, Sessions, QoS, and Channels is illustrated in Figure 1.1. The code required to create a channel is depicted in Listing 1.7.

Listing 1.7. Creating a PrintChannel

```
package irdp.demo.tutorialDA.print;

public class Example {
        public static void main(String[] args) {
            /* Create layers and put them in a array */
            Layer[] qos =
                {new PrintLayer(),
                 new BoundedPrintLayer(),
                 new PrintApplicationLayer()};

            /* Create a QoS */
            QoS myQoS = null;
            try {
                myQoS = new QoS(”Print␣stack”, qos);
            } catch (AppiaInvalidQoSException ex) {
                System.err.println(”Invalid␣QoS”);
                System.err.println(ex.getMessage());
                System.exit(1);
            }

            /* Create a channel. Uses default event scheduler. */
            Channel channel = myQoS.createUnboundChannel(”Print␣Channel”);

            try {
                channel.start ();
            } catch(AppiaDuplicatedSessionsException ex) {
                System.err.println(”Error␣in␣start”);
                System.exit(1);
            }

            /* All set. Appia main class will handle the rest */
            System.out.println(”Starting␣Appia...”);
            Appia.run();
        }
}
```

The reader is now invited to install the *Appia* distribution provided as a companion of this book and try the implementations described above.

Try It

To test the Print and BoundedPrint implementations, use the Example class, located in the demo.tutorialDA package.

To run a simple test, execute the following steps:

1. Open a shell/command prompt.
2. In the shell go to the directory where you have placed the supplied code.
3. Launch the test application:
   `java -cp bin:lib/appia.jar irdp/tutorialDA/Example`
   Note: If the error NoClassDefError has appeared, confirm that you correctly added all the needed jar files and the directory of the compiled code in the classpath parameter of the Java Virtual Machine.

In the output displayed, each line starts with the name of the layer that is writing the output. The PrintApplication layer displays the identification of the next print request between parentheses. After you press the Enter key, that identification applies to the last typed text.

Now that the process is launched and running, you may try the following execution:

1. Type the text `adieu` (print request 1).
   - Note that `Ok` status was received for this text.
2. Type the text `goodbye` (print request 2).
   - Note that `Ok` status was received for this text.
3. Type the text `adios` (print request 3).
   - Note that `Ok` status was received for this text.
4. Type the text `sayonara` (print request 4).
   - Note that `Ok` status was received for this text.
5. Type the text `adeus` (print request 5).
   - Note that an ALARM notification was received because the limit of the BoundedPrint layer, predefined value of 5, was reached.
   - Nevertheless an Ok status was received for this text.
6. Any further typed text will receive a `Not Ok` status.

2. Basic Abstractions

We now describe the implementation of some of the abstractions presented in this chapter. However, before proceeding, we need to introduce some additional components of the *Appia* framework.

2.1 Sendable Event

For the implementation of the protocols that we will be describing, we have defined a specialization of the basic *Appia* event, called SendableEvent. The interface of this event is presented in Listing 2.1.

Listing 2.1. SendableEvent interface

```
package net.sf.appia.events;

public class SendableEvent extends Event implements Cloneable {
    public Object dest;
    public Object source;
    protected Message message;

    public SendableEvent();
    public SendableEvent(Channel channel, int dir, Session source);
    public SendableEvent(Message msg);
    public SendableEvent(Channel channel, int dir, Session source, Message msg);

    public Message getMessage();
    public void setMessage(Message message);
    public Event cloneEvent();
}
```

A SendableEvent owns three relevant attributes: a Message which contains the data to be sent on the network, the source which identifies the sending process, and the destination attribute which identifies the recipient processes.

Since our implementations are based on low-level protocols from the IP family, processes will be identified by a tuple (IP address, port). Therefore, both the source and the dest attributes should contain an object of type java.net.SocketAddress (used by Java TCP and UDP interface).

2.2 Message

The Message component is provided by the *Appia* framework to simplify the task of adding and extracting protocol headers to/from the message payload. Chunks of data can be added to or extracted from the message using the auxiliary MsgBuffer data structure, depicted in Listing 2.2.

Listing 2.2. MsgBuffer interface

```
package net.sf.appia.message;

public class MsgBuffer {
  public byte[] data;
  public int off;
  public int len;

  public MsgBuffer();
  public MsgBuffer(byte[] data, int off, int len);
}
```

The interface of the Message object is partially listed in Listing 2.3. Note the methods to push and popped MsgBuffers to/from a message, as well as methods to fragment and concatenate messages. To ease the programming of distributed protocols in Java, the basic Message class also allow arbitrary objects to be pushed and popped.

Listing 2.3. Message interface (partial)

```
package net.sf.appia.message;

public class Message implements Cloneable {

    public Message();

    public int length();
    public void peek(MsgBuffer mbuf);
    public void pop(MsgBuffer mbuf);
    public void push(MsgBuffer mbuf);
    public void frag(Message m, int length);
    public void join(Message m);
    public void pushObject(Object obj);
    public void pushLong(long l);
    public void pushInt(int i);
    /* ... */
    public Object popObject();
    public long popLong();
    public int popInt();
    /* ... */
    public Object peekObject();
    public long peekLong();
    public int peekInt();
    /* ... */
    public Object clone() throws CloneNotSupportedException;
}
```

2.3 Fair-Loss Point-to-Point Links

The Fair-Loss Point-to-Point Links abstraction is implemented in *Appia* by the UdpSimple protocol. The UdpSimple protocol uses UDP sockets as unreliable communication channels. When a UdpSimple session receives a SendableEvent with the down direction (i.e., a transmission request) it extracts the message from the event and pushes it to the UDP socket. When a message is received from a UDP socket, a SendableEvent is created with the up direction.

2.4 Perfect Point-to-Point Links

The Perfect Point-to-Point Links abstraction is implemented in *Appia* by the TcpBasedPerfectP2P protocol (in fact, this protocol can be found in the current *Appia* distribution with the name TcpComplete. As its name implies, this implementation is based on the TCP protocol; more precisely, it uses TCP sockets as communication channels. When a TcpBasedPerfectP2P session receives a SendableEvent with the down direction (i.e., a transmission request) it extracts the message from the event and pushes it to the TCP socket. When a message is received from a TCP socket, a SendableEvent is created with the up direction.

A TcpBasedPerfectP2P session automatically establishes a TCP connection when requested to send a message to a given destination for the first time. Therefore, a single session implements multiple point-to-point links.

It should be noted that, in pure asynchronous systems, this implementation is just an approximation of the Perfect Point-to-Point Link abstraction. In fact, TCP includes acknowledgments and retransmission mechanisms (to recover from omissions in the network). However, if the other endpoint is unresponsive, TCP breaks the connection, assuming that the corresponding node has crashed. Therefore, TCP makes synchronous assumptions about the system and fails to deliver the messages when it erroneously “suspects” correct processes.

2.5 Perfect Failure Detector

In our case, the Perfect Failure Detector is implemented by the TcpBasedPFD, which is used only with the TcpBasedPerfectP2P protocol described above, built using TCP channels. When a TCP socket is closed, the protocol that implements TcpBasedPerfectP2P sends an event to the *Appia* channel. This event is accepted by the TcpBasedPFD protocol, which sends a Crash event to notify other layers. The implementation of this notification is shown in Listing 2.4. The protocols that use the TcpBasedPFD must declare that they will accept (and process) the Crash event generated by the TcpBasedPFD module. This is illustrated in the implementation of the reliable broadcast protocols, which are described in the next chapter.

To notify other layers of a closed socket, the TcpBasedPerfectP2P protocol must first create the corresponding TCP sockets. The way the TcpBasedPerfectP2P is implemented, these sockets are opened on demand, i.e., when there is the need to send/receive something from a remote peer. To ensure that these sockets are created, the TcpBasedPFD session sends a message to all other processes when it is started.

Note that the TcpBasedPFD abstraction assumes that all processes are started before it starts operating. Therefore, the user must start all processes before activating the perfect failure detector. Otherwise, the detector may detect as failed processes that have not yet been launched. Hence, in subsequent chapters, when using the perfect failure detector in conjunction with other protocols, the user will be requested to explicitly start the perfect failure detector. In most test applications, this is achieved by issuing the `startpfd` request on the command line. The implementation is illustrated in Listing 2.4.

Listing 2.4. Perfect failure detector implementation

```
package irdp.protocols.tutorialDA.tcpBasedPFD;

public class TcpBasedPFDSession extends Session {
  private Channel channel;
  private ProcessSet processes;
  private boolean started;

  public TcpBasedPFDSession(Layer layer) {
    super(layer);
    started = false;
  }

  public void handle(Event event) {
    if (event instanceof TcpUndeliveredEvent)
      notifyCrash((TcpUndeliveredEvent) event);
    else if (event instanceof ChannelInit)
      handleChannelInit((ChannelInit) event);
    else if (event instanceof ProcessInitEvent)
      handleProcessInit((ProcessInitEvent) event);
    else if(event instanceof PFDStartEvent)
      handlePFDStart((PFDStartEvent) event);
  }

  private void handleChannelInit(ChannelInit init) {
    channel = init.getChannel();
    init .go();
  }

  private void handleProcessInit(ProcessInitEvent event) {
    processes = event.getProcessSet();
    event.go();
  }

  private void handlePFDStart(PFDStartEvent event) {
    started = true;
    event.go();
    CreateChannelsEvent createChannels =
      new CreateChannelsEvent(channel,Direction.DOWN,this);
    createChannels.go();
  }

  private void notifyCrash(TcpUndeliveredEvent event) {
    if(started){
      SampleProcess p = processes.getProcess((SocketAddress) event.who);
      if (p.isCorrect()) {
        p.setCorrect(false);
        Crash crash =
          new Crash(channel,Direction.UP,this,p.getProcessNumber());
        crash.go();
      }
    }
  }
}
```

3. Reliable Broadcast

We now describe the implementation, in *Appia*, of several of the protocols introduced in this chapter.

3.1 Basic Broadcast

The communication stack used to illustrate the protocol is the following:

Application
Best-Effort Broadcast **(implemented by Basic Broadcast)**
Perfect Point-to-Point Links (implemented by TcpBasedPerfectP2P)

The implementation of this algorithm closely follows Algorithm “Basic Broadcast”. As shown in Listing 3.1, this protocol only handles three classes of events, namely, the ProcessInitEvent, used to initialize the set of processes that participate in the broadcast (this event is triggered by the application after reading the configuration file), the ChannelInit event, automatically triggered by the runtime when a new channel is created, and the SendableEvent. This last event is associated with transmission requests (if the event flows in the stack downward) or the reception of events from the layer below (if the event flows upward). Note that the code in these listing has been simplified. In particular, all exception handling code was deleted from the listings for clarity (but is included in the code distributed with the book).

The only method that requires some coding is the bebBroadcast() method, which is in charge of sending a series of point-to-point messages to all members of the group. This is performed by executing the following instructions for each member of the group: i) the event being sent is “cloned” (this effectively copies the data to be sent to a new event); ii) the source and destination address of the point-to-point message are set; iii) the event is forwarded to the layer below. There is a single exception to this procedure: if the destination process is the sender itself, the event is immediately delivered to the upper layer. The method to process messages received from the the layer below is very simple: it just forwards the message up.

Listing 3.1. Basic Broadcast implementation

```
package irdp.protocols.tutorialDA.basicBroadcast;

public class BasicBroadcastSession extends Session {

    private ProcessSet processes;

    public BasicBroadcastSession(Layer layer) {
        super(layer);
    }


    public void handle(Event event){
        if(event instanceof ChannelInit)
            handleChannelInit((ChannelInit)event);
        else if(event instanceof ProcessInitEvent)
            handleProcessInitEvent((ProcessInitEvent) event);
        else if(event instanceof SendableEvent){
            if(event.getDir()==Direction.DOWN)
                // UPON event from the above protocol (or application)
                bebBroadcast((SendableEvent) event);
            else
                // UPON event from the bottom protocol (or perfect point2point links)
                pp2pDeliver((SendableEvent) event);
        }
    }

    private void handleProcessInitEvent(ProcessInitEvent event) {
        processes = event.getProcessSet();
        event.go();
    }

    private void handleChannelInit(ChannelInit init) {
        init .go();
    }

    private void bebBroadcast(SendableEvent event) {
        SampleProcess[] processArray = this.processes.getAllProcesses();
        SendableEvent sendingEvent = null;
        for(int i=0 ; i<processArray.length ; i++){
            // source and destination for data message
            sendingEvent = (SendableEvent) event.cloneEvent();
            sendingEvent.source = processes.getSelfProcess().getAddress();
            sendingEvent.dest = processArray[i].getAddress();
            // set the event fields
            sendingEvent.setSourceSession(this); // the session that created the event
            if(i == processes.getSelfRank())
                sendingEvent.setDir(Direction.UP);
            sendingEvent.init();
            sendingEvent.go();
        }
    }

    private void pp2pDeliver(SendableEvent event) {
        event.go();
    }
}
```

Try It

The previous implementation may be experimented with using a simple test application, called SampleAppl. An optional parameter in the command line allows the user to select which protocol stack the application will use. The general format of the command line is the following:

```
./run.sh -f <cf> -n <rank> -qos <prot>
```

The `cf` parameter is the name of a text file with the information about the set of processes, namely, the total number N of processes in the system and, for each of these processes, its "rank" and "endpoint." The rank is a unique logical identifier of each process (an integer from 0 to $N-1$). The "endpoint" is just the host name or IP address and the port number of the process. This information is used by low-level protocols (such as TCP or UDP) to establish the links among the processes. The configuration file has the following format:

```
<number_of_processes>
<rank> <host_name> <port>
...
<rank> <host_name> <port>
```

For example, the following configuration file could be used to define a group of three processes, all running on the local machine:

```
3
0 localhost 25000
1 localhost 25100
2 localhost 25200
```

The `rank` parameter identifies the rank of the process being launched (and, implicitly, the address to be used by the process, taken from the configuration file).

As noted above, `prot` parameter specifies which abstraction is used by the application. There are several possible values. To test our basic broadcast implementation, use the value "*beb*." After all processes are launched, a message can be sent from one process to the other processes by typing a `bcast <string>` in the command line and pressing the `Enter` key.

3.2 Lazy Reliable Broadcast

The communication stack used to illustrate the protocol is the following:

Application
Reliable Broadcast (implemented by Lazy RB)
Perfect Failure Detector (implemented by TcpBasedPFD)
Best-Effort Broadcast (implemented by Basic Broadcast)
Perfect Point-to-Point Links (implemented by TcpBasedPerfectP2P)

The implementation of this algorithm, shown in Listing 3.2, closely follows Algorithm "Lazy Reliable Broadcast". The protocol accepts four events, namely, the ProcessInitEvent, used to initialize the set of processes that participate in the broadcast (this event is triggered by the application after reading the configuration file), the ChannelInit event, automatically triggered by the runtime when a new channel is created, the Crash event, triggered by the PFD when a node crashes, and the SendableEvent. This last event is associated with transmission requests (if the event flows in the stack downward) or the reception of events from the layer below (if the event flows upward). Note that the code in these listings has been simplified. In particular, all exception handling code was deleted from the listings for clarity (but is included in the code distributed with the book).

In order to detect duplicates, each message needs to be uniquely identified. In this implementation, the protocol uses the rank of the sender of the message and a sequence number. This information needs to be pushed into the message header when a message is sent, and then popped again when the message is received. Note that during the retransmission phase, it is possible for the same message, with the same identifier, to be broadcast by different processes.

In the protocol, broadcasting a message consists of pushing the message identifier and forwarding the request to the Best-Effort layer. Receiving the message consists of popping the message identifier, checking for duplicates, and logging and delivering the message when it is received for the first time. Upon a crash notification, all messages from the crashed node are broadcast again. Note that when a node receives a message for the first time, if the sender is already detected to have crashed, the message is immediately retransmitted.

Listing 3.2. Lazy Reliable Broadcast implementation

```
package irdp.protocols.tutorialDA.lazyRB;

public class LazyRBSession extends Session {
    private ProcessSet processes;
    private int seqNumber;
    private LinkedList[] from;
    private LinkedList delivered;

    public LazyRBSession(Layer layer) {
        super(layer);
        seqNumber = 0;
    }

    public void handle(Event event){
        // (...)
    }

    private void handleChannelInit(ChannelInit init) {
        init .go();
        delivered = new LinkedList();
    }

    private void handleProcessInitEvent(ProcessInitEvent event) {
        processes = event.getProcessSet();
        event.go();
        from = new LinkedList[processes.getSize()];
        for (int i=0; i<from.length; i++)
            from[i] = new LinkedList();
    }

    private void rbBroadcast(SendableEvent event) {
        SampleProcess self = processes.getSelfProcess ();
        MessageID msgID = new MessageID(self.getProcessNumber(),seqNumber);
        seqNumber++;
        event.getMessage().pushObject(msgID);
        bebBroadcast(event);
    }

    private void bebDeliver(SendableEvent event) {
        MessageID msgID = (MessageID) event.getMessage().peekObject();
        if( ! delivered.contains(msgID) ){
            delivered.add(msgID);
            SendableEvent cloned = (SendableEvent) event.cloneEvent();
            event.getMessage().popObject();
            event.go();
            SampleProcess pi = processes.getProcess((SocketAddress) event.source);
            int piNumber = pi.getProcessNumber();
            from[piNumber].add(event);
            if( ! pi.isCorrect() ){
                SendableEvent retransmit = (SendableEvent) cloned.cloneEvent();
                bebBroadcast(retransmit);
            }
        }
    }

    private void bebBroadcast(SendableEvent event) {
        event.setDir(Direction.DOWN);
        event.setSourceSession(this);
        event. init ();
        event.go();
    }

    private void handleCrash(Crash crash) {
        int pi = crash.getCrashedProcess();
        System.out.println(”Process␣”+pi+”␣failed.”);
        processes.getProcess(pi).setCorrect(false);
        SendableEvent event = null;
        ListIterator it = from[pi]. listIterator ();
        while(it.hasNext()){
            event = (SendableEvent) it.next();
            bebBroadcast(event);
        }
    from[pi ]. clear ();
    }
}
```

Try It

To test the implementation of the "Lazy Reliable Broadcast" protocol, we will use the same test application that we have used for the basic broadcast. Please refer to the corresponding "try it" section for details.

The `prot` parameter should be used in this case with value "*rb*." Note that the "Lazy Reliable Broadcast" algorithm uses the perfect failure detector module. As described in the previous chapter, this module needs to be activated. For this purpose, the test application also accepts the `startpfd` command; do not forget to initiate the PFD for every process by issuing the `startpfd` request on the command line before testing the protocol.

Hand-On Exercise 3.1 *This implementation of the Reliable Broadcast algorithm has a* delivered *set that is never garbage collected. Modify the implementation to remove messages that no longer need to be maintained in the* delivered *set.*

3.3 All-Ack Uniform Reliable Broadcast

The communication stack used to illustrate the protocol is the following:

Application
Uniform Reliable Broadcast **(implemented by All-Ack Uniform Reliable Broadcast)**
Perfect Failure Detector (implemented by TcpBasedPFD)
Best-Effort Broadcast (implemented by Basic Broadcast)
Perfect Point-to-Point Links (implemented by TcpBasedPerfectP2P)

The implementation of this protocol is shown in Listing 3.3. Note that the code in these listings has been simplified. In particular, all exception handling code was deleted from the listings for clarity (but is included in the code distributed with the book).

The protocol uses two variables, received and delivered to register which messages have already been received and delivered, respectively. These variables only store message identifiers. When a message is received for the first time, it is forwarded as specified in the algorithm. To keep track on who has already acknowledged (forwarded) a given message, a hash table is used. There is an entry in the hash table for each message. This entry keeps the data message itself (for future delivery) and a record of who has forwarded the message.

When a message has been forwarded by every correct process, it can be delivered. This is checked every time a new event is handled (as both the reception of messages and the crash of processes may trigger the delivery of pending messages).

Listing 3.3. All-Ack Uniform Reliable Broadcast implementation

```
package appia.protocols.tutorialDA.allAckURB;

public class AllAckURBSession extends Session {
  private ProcessSet processes;
  private int seqNumber;
  private LinkedList received, delivered;
  private Hashtable ack;


  public AllAckURBSession(Layer layer) {
    super(layer);
  }

  public void handle(Event event) {
    // (...)
    urbTryDeliver();
  }

  private void urbTryDeliver() {
    Iterator it = ack.values().iterator();
    MessageEntry entry=null;
    while( it.hasNext() ){
      entry = (MessageEntry) it.next();
      if(canDeliver(entry)){
        delivered.add(entry.messageID);
        urbDeliver(entry.event, entry.messageID.process);
      }
    }
  }

  private boolean canDeliver(MessageEntry entry) {
    int procSize = processes.getSize();
    for(int i=0; i<procSize; i++)
      if(processes.getProcess(i).isCorrect() && (! entry.acks[i]) )
        return false;
    return ( (! delivered.contains(entry.messageID)) && received.contains(entry.messageID) );
  }

  private void handleChannelInit(ChannelInit init) {
    init.go();
    received = new LinkedList();
    delivered = new LinkedList();
    ack = new Hashtable();
  }

  private void handleProcessInitEvent(ProcessInitEvent event) {
    processes = event.getProcessSet();
    event.go();
  }

  private void urbBroadcast(SendableEvent event) {
    SampleProcess self = processes.getSelfProcess();
    MessageID msgID = new MessageID(self.getProcessNumber(),seqNumber);
    seqNumber++;
    received.add(msgID);
    event.getMessage().pushObject(msgID);
    event.go ();
  }

  private void bebDeliver(SendableEvent event) {
    SendableEvent clone = (SendableEvent) event.cloneEvent();
    MessageID msgID = (MessageID) clone.getMessage().popObject();
    addAck(clone,msgID);
    if( ! received.contains(msgID) ){
      received.add(msgID);
      bebBroadcast(event);
    }
  }

  private void bebBroadcast(SendableEvent event) {
    event.setDir(Direction.DOWN);
    event.setSourceSession(this);
    event.init();
    event.go();
  }

  private void urbDeliver(SendableEvent event, int sender) {
    event.setDir(Direction.UP);
    event.setSourceSession(this);
    event.source = processes.getProcess(sender).getAddress();
    event.init();
    event.go();
  }

  private void handleCrash(Crash crash) {
    int crashedProcess = crash.getCrashedProcess();
    System.out.println(”Process␣”+crashedProcess+”␣failed.”);
    processes.getProcess(crashedProcess).setCorrect(false);
  }

  private void addAck(SendableEvent event, MessageID msgID){
    int pi = processes.getProcess((SocketAddress)event.source).getProcessNumber();
    MessageEntry entry = (MessageEntry) ack.get(msgID);
    if(entry == null){
      entry = new MessageEntry(event, msgID, processes.getSize());
      ack.put(msgID,entry);
    }
    entry.acks[pi] = true;
  }
}
```

Try It

To test the implementation of the “All-Ack Uniform Reliable Broadcast” protocol, we will use the same test application that we have used for the basic broadcast. Please refer to the corresponding “try it” section for details.

The `prot` parameter that should be used in this case is “*urb*.” Note that the All-Ack Uniform Reliable Broadcast protocol uses the perfect failure detector module. As described in the previous chapter, this module needs to be activated. For this purpose, the test application also accepts the `startpfd` command; do not forget to initiate the PFD at every processes by issuing the `startpfd` request on the command line before testing the protocol.

Hand-On Exercise 3.2 *Modify the implementation to keep track just of the last message sent from each process in the* received *and* delivered *variables.*

Hand-On Exercise 3.3 *Change the protocol to exchange acknowledgments when the sender is correct, and only retransmit the payload of a message when the sender is detected to have crashed (just like in the Lazy Reliable Protocol).*

3.4 Majority-Ack URB

The communication stack used to illustrate the protocol is the following (note that a Perfect Failure Detector is no longer required):

Application
Uniform Reliable Broadcast **(implemented by Majority-Ack URB)**
Best-Effort Broadcast (implemented by Basic Broadcast)
Perfect Point-to-Point Links (implemented by TcpBasedPerfectP2P)

The protocol works in the same way as the protocol presented in the prevous section, but without being aware of crashed processes. Besides that, the only difference from the previous implementation is the `canDeliver()` method, which is shown in Listing 3.4.

Listing 3.4. Indulgent Uniform Reliable Broadcast implementation

```
package irdp.protocols.tutorialDA.majorityAckURB;

public class MajorityAckURBSession extends Session {

  private boolean canDeliver(MessageEntry entry) {
    int N = processes.getSize(), numAcks = 0;
    for(int i=0; i<N; i++)
      if(entry.acks[i])
        numAcks++;
    return (numAcks > (N/2)) && ( ! delivered.contains(entry.messageID) );
  }

    // Except for the method above, and for the handling of the crash event, same
    // as in the previous protocol
}
```

Try It

To test the implementation of the Majority-Ack uniform reliable broadcast protocol, we will use the same test application that we have used for the basic broadcast. Please refer to the corresponding “try it” section for details. The `prot` parameter that should be used in this case is “*iurb*.”

Hand-On Exercise 3.4 *Note that if a process does not acknowledge a message, copies of that message may have to be stored for a long period (in fact,* *if a process crashes, copies need to be stored forever). Try to devise a scheme to ensure that no more than $N/2 + 1$ copies of each message are preserved in the system (that is, not all members should be required to keep a copy of every message).*

3.5 Probabilistic Reliable Broadcast

This protocol is based on probabilities and is used to broadcast messages in large groups. Instead of creating Perfect Point-to-Point Links, it use Unreliable Point-to-Point Links (UP2PL) to send messages just for a subset of the group. The communication stack used to illustrate the protocol is the following:

Application
Probabilistic Broadcast (implemented by Eager PB)
Unreliable Point-to-Point Links (implemented by UdpSimple)

The protocol has two configurable parameters: i) *fanout* is the number of processes for which the message will be gossiped about; ii) *maxrounds*, is the maximum number of rounds that the message will be retransmitted.

The implementation of this protocol is shown on Listing 3.5. The gossip() method invokes the pickTargets() method to choose the processes to which the message is going to be sent and sends the message to those targets. The pickTargets() method chooses targets randomly from the set of processes. Each message carries its identification (as previous reliable broadcast protocols) and the remaining number of rounds (when the message is gossiped again, the number of rounds is decremented).

Listing 3.5. Probabilistic Broadcast implementation

```
SocketAddresspackage irdp.protocols.tutorialDA.eagerPB;

public class EagerPBSession extends Session {

  private LinkedList delivered;
  private ProcessSet processes;
  private int fanout, maxRounds, seqNumber;

  public EagerPBSession(Layer layer) {
    super(layer);
    EagerPBLayer pbLayer = (EagerPBLayer) layer;
    fanout = pbLayer.getFanout();
    maxRounds = pbLayer.getMaxRounds();
    seqNumber = 0;
  }

  public void handle(Event event){
    // (...)
  }
```

```
private void handleChannelInit(ChannelInit init) {
  init.go();
  delivered = new LinkedList();
}

private void handleProcessInitEvent(ProcessInitEvent event) {
  processes = event.getProcessSet();
  fanout = Math.min (fanout, processes.getSize ());
  event.go();
}

private void pbBroadcast(SendableEvent event) {
  MessageID msgID = new MessageID(processes.getSelfRank(),seqNumber);
  seqNumber++;
  gossip(event, msgID, maxRounds−1);
}

private void up2pDeliver(SendableEvent event) {
  SampleProcess pi = processes.getProcess((SocketAddress)event.source);
  int round = event.getMessage().popInt();
  MessageID msgID = (MessageID) event.getMessage().popObject();
  if( ! delivered.contains(msgID) ){
    delivered.add(msgID);
    SendableEvent clone = null;
    clone = (SendableEvent) event.cloneEvent();
    pbDeliver(clone,msgID);
  }
  if(round > 0)
    gossip(event,msgID,round−1);
}

private void gossip(SendableEvent event, MessageID msgID, int round){
    int[] targets = pickTargets();
    for(int i=0; i<fanout; i++){
      SendableEvent clone = (SendableEvent) event.cloneEvent();
      clone.getMessage().pushObject(msgID);
      clone.getMessage().pushInt(round);
      up2pSend(clone,targets[i]);
    }
}

private int[] pickTargets() {
  Random random = new Random(System.currentTimeMillis());
  LinkedList targets = new LinkedList();
  Integer candidate = null;
  while(targets.size() < fanout){
    candidate = new Integer(random.nextInt(processes.getSize()));
    if( (! targets.contains(candidate)) &&
                          (candidate.intValue() != processes.getSelfRank()) )
      targets.add(candidate);
  }
  int[] targetArray = new int[fanout];
  ListIterator it = targets.listIterator();
  for(int i=0; (i<targetArray.length) && it.hasNext(); i++)
    targetArray[i] = ((Integer)it.next()).intValue();
  return targetArray;
}

private void up2pSend(SendableEvent event, int dest) {
  event.setDir(Direction.DOWN);
  event.setSourceSession(this);
  event.dest = processes.getProcess(dest).getAddress();
  event.init();
  event.go();
}
```

```
  private void pbDeliver(SendableEvent event, MessageID msgID) {
    event.setDir(Direction.UP);
    event.setSourceSession(this);
    event.source = processes.getProcess(msgID.process).getAddress();
    event. init ();
    event.go();
  }
}
```

Try It

To test the implementation of the probabilistic reliable broadcast protocol, we will use the same test application that we have used for the basic broadcast. Please refer to the corresponding “try it” section for details. The `prot` parameter that should be used in this case is “*pb <fanout> <maxrounds>*”, where the parameters are used to specify the fanout and the maximum number of message rounds.

Hands-On Exercises

Hand-On Exercise 3.5 *The up2pDeliver() method performs two different functions:* i) *it delivers the message to the application (if it has not been delivered yet) and* ii) *it gossips about the message to other processes. Change the code such that a node gossips just when it receives a message for the first time. Discuss the impact of the changes.*

Hand-On Exercise 3.6 *Change the code to limit* i) *the number of messages each node can store, and* ii) *the maximum throughput (messages per unit of time) of each node.*

3.6 No-Waiting Causal Broadcast

The communication stack used to implement the protocol is the following:

Application
Reliable Causal Order **(implemented by No-Waiting CB)**
Delay
Reliable Broadcast (implemented by Lazy RB)
Perfect Failure Detector (implemented by TcpBasedPFD)
Best-Effort Broadcast (implemented by Basic Broadcast)
Perfect Point-to-Point Links (implemented by TcpBasedPerfectP2P)

The role of each of the layers is explained below.

SampleAppl: This layer implements the test application mentioned previously.

NoWaitingCO: This layer implements the causal order protocol. Each message, in the protocol, is uniquely identified by its source and a sequence number, as each process in the group has its own sequence number. The events that walk through the stack are not serializable, so we have chosen the relevant information of those events to send, as the past list. A message coming from the protocol is represented in Figure 3.1.

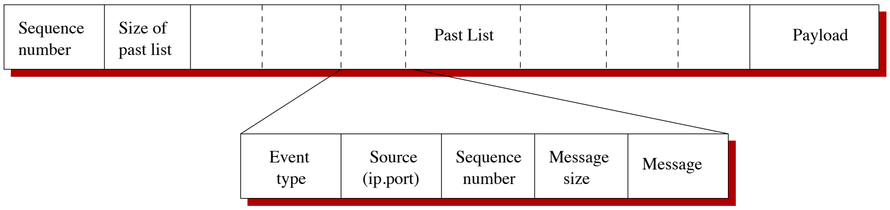

Fig. 3.1: Format of messages exchanged by CONoWaiting protocol

Delay: Test protocol used to delay the messages of one process/source when delivering them to one process/destination. This is used to check that messages are really delivered in the right order, even when delays are present. In short, this layer simulates network delays. Note that this layer does not belong to the protocol stack; it was developed just for testing.

LazyRB: Protocol that implements the reliable broadcast algorithm. The remaining layers are required by this protocol (see Chapter 3).

The protocol implementation is depicted in Listing 3.6.

Listing 3.6. No-Waiting Reliable Causal Order Broadcast implementation

```
SocketAddress
public class CONoWaitingSession extends Session {
    Channel channel;
    SocketAddress myID;
    int seqNumber=0;
    LinkedList delivered; /* Set of delivered messages */
    LinkedList myPast; /* Set of messages in past */

    public CONoWaitingSession(Layer l) {
        super(l);
    }

    public void handle(Event e) {
        if (e instanceof ChannelInit)
            handleChannelInit((ChannelInit)e);
        else if (e instanceof RegisterSocketEvent)
            handleRegisterSocketEvent((RegisterSocketEvent)e);
        else if (e instanceof SendableEvent){
            if (e.getDirection().direction==Direction.DOWN)
                handleSendableEventDOWN((SendableEvent)e);
            else
                handleSendableEventUP((SendableEvent)e);
        }
    }

    public void handleChannelInit (ChannelInit e){
        delivered=new LinkedList();
        myPast=new LinkedList();
        this.channel = e.getChannel();
        e.go();
    }

    public void handleRegisterSocketEvent (RegisterSocketEvent e){
        myID=new SocketAddress(InetAddress.getLocalHost(),e.port);
        e.go();
    }

    public void handleSendableEventDOWN (SendableEvent e){
        //cloning the event to be sent in oder to keep it in the mypast list...
        SendableEvent e_aux= (SendableEvent)e.cloneEvent();
        Message om=e.getMessage();


        //inserting myPast list in the msg:
        for(int k=myPast.size();k>0;k--){
            Message om_k= ((ListElement)myPast.get(k-1)).getSE().getMessage();

            om.push(om_k.toByteArray());
            om.pushInt(om_k.toByteArray().length);
            om.pushInt(((ListElement)myPast.get(k-1)).getSeq());
            om.pushObject((ListElement)myPast.get(k-1).getSE().source,om);
            om.pushString(((ListElement)myPast.get(k-1)).getSE().getClass().getName());
        }
        om.pushInt(myPast.size());
        om.pushInt(seqNumber);
        e.setMessage(om);

        e.go();

        //add this message to the myPast list:
        e_aux.source=myID;
        ListElement le=new ListElement(e_aux,seqNumber);
        myPast.add(le);

        //increments the global seq number
        seqNumber++;
    }

    public void handleSendableEventUP (SendableEvent e){
        Message om=e.getMessage();
        int seq=om.popInt();

        //checks to see if this msg has been already delivered...
        if (! isDelivered ((SocketAddress)e.source,seq)){

            //size of the past list of this msg
            int pastSize=om.popInt();
            for(int k=0;k<pastSize;k++){
                String className=om.popString();
                SocketAddress msgSource=(SocketAddress)om.popObject();
                int msgSeq=om.popInt();
                int msgSize=om.popInt();
                byte[] msg=(byte[])om.pop();
                //if this msg hasn't been already delivered, we must deliver it prior to the one that just arrived!
                if(!isDelivered(msgSource,msgSeq)){
                    //composing and sending the msg!
                    SendableEvent se=null;
                    se = (SendableEvent) Class.forName(className).newInstance();
                    se.setChannel(channel);
                    se.setDirection(new Direction(Direction.UP));
                    se.setSourceSession(this);

                    Message aux_om = new Message();
                    aux_om.setByteArray(msg,0,msgSize);
                    se.setMessage(aux_om);
                    se.source=msgSource;

                    se.init();
                    se.go();
                }

                //this msg has been delivered!
                ListElement le=new ListElement(se,msgSeq);
                delivered.add(le);
                myPast.add(le);
            }
        }

        //cloning the event just received to keep it in the mypast list
        SendableEvent e_aux=null;
        e_aux=(SendableEvent)e.cloneEvent();

        // deliver the event
        e.setMessage(om);
        e.go();

        ListElement le=new ListElement(e_aux,seq);
        delivered.add(le);

        if(!e_aux.source.equals(myID))
            myPast.add(le);
    }

    boolean isDelivered(SocketAddress source,int seq){
        for(int k=0;k<delivered.size();k++){
            SocketAddress iwp_aux=(SocketAddress) ((ListElement)delivered.get(k)).getSE().source;
            int seq_aux=((ListElement)delivered.get(k)).getSeq();
            if( iwp_aux.equals(source) && seq_aux==seq)
                return true;
        }
        return false;
    }
}

class ListElement{
    SendableEvent se;
    int seq;

    public ListElement(SendableEvent se, int seq){
        this.se=se;
        this.seq=seq;
    }

    SendableEvent getSE(){
        return se;
    }

    int getSeq(){
        return seq;
    }
}
```

Try It

To test the implementation of the no-waiting reliable causal order broadcast, we will use the same test application that we have used for the basic broadcast. Please refer to the corresponding "try it" section for details. The `prot` parameter that should be used in this case is "**conow**." Note that this protocol uses the perfect failure detector module. As described in the previous chapter, this module need to be activated. For this purpose, the test application also accepts the `startpfd` command; do not forget to initiate the PFD at every processes by issuing the `startpfd` request on the command line before testing the protocol.

To run some simple tests, execute the following steps:

1. Open three shells/command prompts.
2. In each shell go to the directory where you have placed the supplied code.
3. In each shell launch the test application, *SampleAppl*, giving a different n value (0, 1 or 2) and specifying the *qos* as **conow**.
   - In shell 0 execute:

     ```
     ./run.sh  -f etc/procs -n 0 -qos conow
     ```

   - In shell 1 execute:

     ```
     ./run.sh -f etc/procs -n 1 -qos conow
     ```

   - In shell 2 execute:

     ```
     ./run.sh -f etc/procs -n 2 -qos conow
     ```

   Note: If the error NoClassDefError has appeared, confirm that you are at the root of the supplied code.

Now that processes are launched and running, you may try the following two distinct executions:

1. Execution I:
   a) In shell 0, send a message **M1** (type **bcast M1** and press enter).
      - Note that all processes received **M1**.
   b) In shell 1, send a message **M2**.
      - Note that all processes received **M2**.
   c) Confirm that all processes have received **M1** and then **M2**, and note the continuous growth of the size of the messages sent.

2. Execution II: For this execution it is necessary to first modify file SampleAppl.java in package demo.tutorialDA. The sixth line of method getCOnoWChannel should be uncommented in order to insert a test layer that allows the injection of delays in messages sent between process 0 and process 2. After modifying the file, it is necessary to compile it.
   a) In shell 0, send a message **M1**.
      - Note that process 2 did not receive **M1**.
   b) In shell 1, send a message **M2**.
      - Note that all processes received **M2**.
   c) Confirm that all processes received **M1** and then **M2**. Process 2 received **M1** because it was appended to **M2**.

3.7 No-Waiting Causal Broadcast with Garbage Collection

The next protocol we present is an optimization of the previous one. It intends to circumvents its main disadvantage by deleting messages from the past list. When the protocol delivers a message, it broadcasts an acknowledgment to all other processes; when an acknowledgment for the same message has been received from all correct processes, this message is removed from the past list.

The communication stack used to implement the protocol is the following:

SampleAppl
Reliable CO **(implemented by Garbage Collection Of Past)**
Delay
Reliable Broadcast (implemented by Lazy RB)
Perfect Failure Detector (implemented by TcpBasedPFD)
Best-Effort Broadcast (implemented by Basic Broadcast)
Perfect Point-to-Point Links (implemented by TcpBasedPerfectP2P)

The protocol implementation is depicted in Listing 3.7.

Listing 3.7. No-Waiting Reliable Causal Order Broadcast with Garbage Collection implementation

```
package irdp.protocols.tutorialDA.gcPastCO;

public class GCPastCOSession extends Session {
    Channel channel;
    int seqNumber=0;
    LinkedList delivered; // Set of delivered messages.
    LinkedList myPast; // Set of messages processed by this element.
    private ProcessSet correct; // Set of the correct processes.
    LinkedList acks; // Set of the msgs not yet acked by all correct processes.

    public GCPastCOSession(Layer l) {
        super(l);
    }

    public void handle(Event e) {
        if (e instanceof ChannelInit)
            handleChannelInit((ChannelInit)e);
        else if (e instanceof ProcessInitEvent)
            handleProcessInit((ProcessInitEvent)e);
        else if (e instanceof AckEvent)
            handleAck((AckEvent)e);
        else if (e instanceof SendableEvent){
            if (e.getDir()==Direction.DOWN)
                handleSendableEventDOWN((SendableEvent)e);
            else
                handleSendableEventUP((SendableEvent)e);
        }else if (e instanceof Crash)
            handleCrash((Crash)e);
        else{
            e.go();
        }
    }

    public void handleChannelInit (ChannelInit e){
        e.go();
        this.channel = e.getChannel();
        delivered=new LinkedList();
        myPast=new LinkedList();
        acks=new LinkedList();
    }

    public void handleProcessInit (ProcessInitEvent e){
        correct = e.getProcessSet();
        e.go();
    }

    public void handleSendableEventDOWN (SendableEvent e){
        // same as the handleSendableEventDOWN method of CONoWaitingSession
        // ...
    }

    public void handleSendableEventUP (SendableEvent e){
        Message om=e.getMessage();
        int seq=om.popInt();

        //checks to see if this msg has been already delivered...
        if (! isDelivered ((SocketAddress)e.source,seq)){

            //size of the past list of this msg
            int pastSize=om.popInt();
            for(int k=0;k<pastSize;k++){
                String className=om.popString();
                SocketAddress msgSource=SocketAddress.pop(om);
                int msgSeq=om.popInt();
                int msgSize=om.popInt();
                byte[] msg=(byte[])om.pop();

                // if this msg hasn't been already delivered,
                // we must deliver it prior to the one that just arrived!
                if (! isDelivered (msgSource,msgSeq)){
                    //composing and sending the msg!
                    SendableEvent se =
                        (SendableEvent) Class.forName(className).newInstance();
                    se.setChannel(channel);
                    se.setDir(Direction.UP);
                    se.setSourceSession(this);
                    Message aux_om = new Message();
                    aux_om.setByteArray(msg,0,msgSize);
                    se.setMessage(aux_om);
                    se.source=msgSource;

                    se. init ();
                    se.go();

                    //this msg has been delivered!
                    ListElement le=new ListElement(se,msgSeq);
                    delivered.add(le);
                    myPast.add(le);

                    //let's send the ACK for this msg
                    sendAck(le);
                }
            }

            //cloning the event just received to keep it in the mypast list
            SendableEvent e_aux=(SendableEvent)e.cloneEvent();

            e.setMessage(om);
            e.go();

            ListElement le=new ListElement(e_aux,seq);
            delivered.add(le);

            //this msg is already in the past list. It was added on the sending.
            if(!e_aux.source.equals(correct.getSelfProcess().getAddress()))
                myPast.add(le);

            //let's send the ACK for this msg
            sendAck(le);
        }

    }

    private void sendAck(ListElement le){
        int index=−1;
        //search for it in the acks list:
        for(int i=0;i<acks.size(); i++){
            if(((AckElement)acks.get(i)).seq==le.seq &&
               ((AckElement)acks.get(i)).source.equals((SocketAddress)le.se.source)){
                index=i;
                i=acks.size();
            }
        }

        if(index==−1){
            //let's create it!
            AckElement ael=new AckElement(le.seq,(SocketAddress)le.se.source);
            acks.add(ael);
            index=acks.size()−1;
        }

        ((AckElement)acks.get(index)).regAck(correct.getSelfProcess().getAddressSocketAddress());

        AckEvent ae=new AckEvent(channel, Direction.DOWN, this);
        Message om = new Message();
        om.pushObject(SocketAddressle.se.source);
        om.pushInt(le.seq);
        ae.setMessage(om);
```

```
        ae. init ();
        ae.go();
    }

    boolean isDelivered(SocketAddress source,int seq){
        // equal to the isDelivered method of CONoWaitingSession class
        // ...
    }

    public void handleAck(AckEvent e){
        //my ACK was already registered when the AckEvent was sent
        if(e.source.equals( correct . getSelfProcess (). getAddress()))
            return;

        Message om=e.getMessage();
        int seq=om.popInt();
        SocketAddress iwp=(SocketAddress) om.popObject();

        int index=−1;
        //search for it in the acks list :
        for(int i=0;i<acks.size (); i++){
            if((( AckElement)acks.get(i)).seq==seq &&
               (( AckElement)acks.get(i)).source.equals(iwp)){
                index=i;
                i=acks.size ();
            }
        }

        if(index==−1){
            //let's create it!
            AckElement ael=new AckElement(seq,iwp);
            acks.add(ael);
            index=acks.size()−1;
        }

        ((AckElement)acks.get(index)).regAck((SocketAddress)e.source);

        // if all correct processes have already acked this msg
        if(getCorrectSize()==((AckElement)acks.get(index)).processes.size()) {
            // removes the entry for this msg from the myPast list
            for(int k=0;k<myPast.size();k++){
                if((( ListElement)myPast.get(k)).se.source.equals(iwp) &&
                   (( ListElement)myPast.get(k)).seq==seq){
                    myPast.remove(k);
                    k=myPast.size();
                }
            }
            // removes the entry for this msg from the acks list
            acks.remove(index);
        }
    }

    public void handleCrash(Crash e){
        correct .setCorrect(e.getCrashedProcess(),false);
        e.go();
    }

    private int getCorrectSize() {
        int i;
        int count=0;
        for (i=0 ; i < correct.getSize() ; i++)
            if (correct .getProcess(i ). isCorrect ())
                count++;
        return count;
    }
}//end CONoWaitingSession
```

```
class ListElement{
    SendableEvent se;
    int seq;

    public ListElement(SendableEvent se, int seq){
        this.se=se;
        this.seq=seq;
    }

    SendableEvent getSE(){
        return se;
    }

    int getSeq(){
        return seq;
    }
}

class AckElement{
    int seq;
    SocketAddress source;
    LinkedList processes; // the set of processes that already acked this msg

    public AckElement(int seq, SocketAddress source){
        this.seq=seq;
        this.source=source;
        processes=new LinkedList();
    }

    void regAck(SocketAddress p){
        processes.add(p);
    }
}
```

Try It

To test the implementation of the no-waiting reliable causal order broadcast with the garbage collection protocol, we will use the same test application that we have used for the basic broadcast. Please refer to the corresponding "try it" section for details.

The `prot` parameter that should be used in this case is "*conowgc.*" Note that this protocol uses the perfect failure detector module. As described in the previous chapter, this module needs to be activated. For this purpose, the test application also accepts the `startpfd` command; do not forget to initiate the PFD at every processes by issuing the `startpfd` request on the command line before testing the protocol.

To run some simple tests, execute the following steps:

1. Open three shells/command prompts.
2. In each shell go to the directory where you have placed the supplied code.
3. In each shell launch the test application, *SampleAppl*, giving a different n value (0, 1 or 2) and specifying the *qos* as **conowgc**.
   - In shell 0 execute:

```
./run.sh -f etc/procs -n 0 -qos conowgc
```

- In shell 1 execute:

```
./run.sh -f etc/procs -n 1 -qos conowgc
```

- In shell 2 execute:

```
./run.sh -f etc/procs -n 2 -qos conowgc
```

Note: If the error NoClassDefError has appeared, confirm that you are at the root of the supplied code.

Now that processes are launched and running, you may try the following three distinct executions:

1. Execution I and II: Since this protocol is very similar with the previous one, the two executions presented in the previous section can be applied to this protocol. Note that the line of code in *demo/tutorialDA/SampleAppl.java* that has to be altered is now the seventh of the *getCOnoWGCChannel* method.
2. Execution III:(this execution is to be done with the delay layer in the protocol stack.)
   a) In shell 0, send a message **M1** (type **bcast M1** and press enter).
      - Note that process 2 did not receive **M1**.

   b) In shell 1, send a message **M2**.
      - Note the size of the message that was sent and note also that all processes received **M2**.

   c) In shell 2, send a message **M3**.
      - Note the smaller size of the message that was sent and note also that all processes received **M3**.

   d) Confirm that all processes received **M1**, **M2**, and **M3** in the correct order.

3.8 Waiting Causal Broadcast

The communication stack used to implement the protocol is the following:

SampleAppl
Reliable Casual Order **(implemented by Waiting CO)**
Delay
Reliable Broadcast (implemented by Lazy RB)
Perfect Failure Detector (implemented by TcpBasedPFD)
Best-Effort Broadcast (implemented by Basic Broadcast)
Perfect Point-to-Point Links (implemented by TcpBasedPerfectP2P)

The protocol implementation is depicted in Listing 3.8.

Listing 3.8. Waiting Causal Broadcast implementation

```
package irdp.protocols.tutorialDA.waitingCO;

public class WaitingCOSession extends Session{
    Channel channel;
    private ProcessSet correct; // Set of the correct processes.
    private LinkedList pendingMsg; // The list of the msg that are waiting to be delivered
    int [] vectorClock;

    public WaitingCOSession(Layer l){
        super(l);
    }

    public void handle(Event e) {
        if (e instanceof ChannelInit)
            handleChannelInit((ChannelInit)e);
        else if (e instanceof ProcessInitEvent)
            handleProcessInit((ProcessInitEvent)e);
        else if (e instanceof SendableEvent){
            if (e.getDir()==Direction.DOWN)
                handleSendableEventDOWN((SendableEvent)e);
            else
                handleSendableEventUP((SendableEvent)e);
        }else{
            e.go();
        }
    }

    public void handleChannelInit (ChannelInit e){
        e.go();
        this.channel = e.getChannel();
        pendingMsg=new LinkedList();
    }

    public void handleProcessInit (ProcessInitEvent e){
        correct=e.getProcessSet();
        vectorClock=new int[correct.getSize()];
        Arrays. fill (vectorClock,0);
        e.go();
    }

    public void handleSendableEventDOWN (SendableEvent e){
        //i'm sending a msg therefore increments my position in the vector clock!
        vectorClock[correct.getSelfRank()]++;
        //add the vector clock to the msg from the appl
        Message om=e.getMessage();
        om.push(vectorClock);

        e.go();
    }

    public void handleSendableEventUP (SendableEvent e){
        //get the vector clock of this msg!
        Message om=e.getMessage();
        int[] vc_msg=(int[]) om.pop();

        if(canDeliver(correct.getRank((SocketAddress)e.source),vc_msg)){
            e.go();

            if(!e.source.equals(correct.getSelfProcess().getAddress()))
                vectorClock[correct.getRank((SocketAddress)e.source)]++;

            checkPending();
        }else{
            om.push(vc_msg);
            pendingMsg.add(e);
        }
    }

    private boolean canDeliver(int rankSource,int[] vc_msg){
        boolean ret=false;
        if(vectorClock[rankSource] >= vc_msg[rankSource]−1)
            ret=true;

        for(int i=0;i < vectorClock.length;i++){
            if(i!=rankSource && vc_msg[i] > vectorClock[i])
                ret=false;
        }

        return ret;
    }

    private void checkPending(){
        // this list will keep the information about
        // which messages can be removed from the pending list!!!
        boolean[] toRemove=new boolean[pendingMsg.size()];
        Arrays.fill(toRemove,false);
        SendableEvent e_aux;

        //runs through the pending List to search for msgs that can already be delivered
        for(int i=0;i<pendingMsg.size();i++){
            e_aux=(SendableEvent) pendingMsg.get(i);
            Message om=e_aux.getMessage();
            int[] vc_msg=(int[])om.pop();

            int sourceRank=correct.getRank((SocketAddress)e_aux.source);

            if(canDeliver(sourceRank,vc_msg)){
                e_aux.go();

                if(!e_aux.source.equals(correct.getSelfProcess().getAddress()))
                    vectorClock[correct.getRank((SocketAddress)e_aux.source)]++;

                toRemove[i]=true;
            }else{
                om.push(vc_msg);
            }
        }

        int countRemoved=0;
```

```
            //now, let's check the toRemove list to clean the pendingMsg list
            for(int k=0;k<toRemove.length;k++){
                if(toRemove[k]){
                    pendingMsg.remove(k−countRemoved);
                    countRemoved++;
                }
            }
        }
}
```

Try It

To test the implementation of the waiting reliable causal order broadcast with garbage collection protocol, we will use the same test application that we have used for the basic broadcast. Please refer to the corresponding "try it" section for details.

The `prot` parameter that should be used in this case is "*cow.*" Note that this protocol uses the perfect failure detector module. As described in the previous chapter, this module needs to be activated. For this purpose, the test application also accepts the `startpfd` command; do not forget to initiate the PFD at every process by issuing the `startpfd` request on the command line before testing the protocol.

To run some simple tests, follow the same steps as described in the two previous "try it" sections, except that the *qos* given must be **cow**. You may try the two following executions:

1. Execution I:
   a) In shell 0, send a message **M1** (type **bcast M1** and press enter).
      - Note that all processes received **M1**.

   b) In shell 1, send a message **M2**.
      - Note that all processes received **M2**.

   c) Confirm that all processes received **M1** and then **M2**.
2. Execution II: For this execution it is necessary to first modify file *demo/tutorialDA/SampleAppl.java*. The seventh line of the *getCOWChannel* method should be uncommented in order to insert a test layer that allows the injection of delays in messages sent between process 0 and process 2. After modifying the file, it is necessary to compile it.
   a) In shell 0, send a message **M1**.
      - Note that process 2 did not receive **M1**.

   b) In shell 1, send a message **M2**.
      - Note that process 2 also did not receive **M2**.

   c) Wait for process 2 to receive the messages.
      - Note that process 2 did not receive **M1**, immediately, due to the presence of the Delay layer; and it also did not receive **M2** immediately, because it had to wait for **M1** to be delivered, as **M1** preceded **M2**.

4. Shared Memory

4.1 $(1, N)$ Regular Register

The $(1, N)$ Regular Register algorithm implemented was the “Read-One Write-All.” The communication stack used to implement this protocol is the following:

Application
$(1, N)$**RegularRegister** **(implemented by Read-One Write-All)**
Perfect Failure Detector (implemented by TcpBasedPFD)
Best-Effort Broadcast (implemented by Basic Broadcast)
Perfect Point-to-Point Links (implemented by TcpBasedPerfectP2P)

This implementation uses two different *Appia* channels because the algorithm uses the *best-effort broadcast* (which in turn is based on *perfect point-to-point links*) also the *perfect point-to-point links* directly (without best-effort reliability).

The protocol implementation is depicted in Listing 4.1. It follows Algorithm in the book very closely. The only significant difference is that values are generic instead of integers, thus allowing registers to contain *strings*.

Listing 4.1. Read-One Write-All $(1, N)$ Regular Register implementation

```
package irdp.protocols.tutorialDA.readOneWriteAll1NRR;

public class ReadOneWriteAll1NRRSession extends Session {
    public static final int NUM_REGISTERS=20;

    public ReadOneWriteAll1NRRSession(Layer layer) {
        super(layer);
    }

    private Object[] value=new Object[NUM_REGISTERS];
    private HashSet[] writeSet=new HashSet[NUM_REGISTERS];
    private ProcessSet correct=null;

    private Channel mainchannel=null;
    private Channel pp2pchannel=null;
    private Channel pp2pinit=null;

    public void handle(Event event) {
        if (event instanceof ChannelInit)
            handleChannelInit((ChannelInit)event);
        else if (event instanceof ProcessInitEvent)
            handleProcessInit((ProcessInitEvent)event);
        else if (event instanceof Crash)
            handleCrash((Crash)event);
        else if (event instanceof SharedRead)
            handleSharedRead((SharedRead)event);
        else if (event instanceof SharedWrite)
            handleSharedWrite((SharedWrite)event);
        else if (event instanceof WriteEvent)
            handleWriteEvent((WriteEvent)event);
        else if (event instanceof AckEvent)
            handleAckEvent((AckEvent)event);
        else {
            event.go();
        }
    }

    public void pp2pchannel(Channel c) {
        pp2pinit=c;
    }

    private void handleChannelInit(ChannelInit init) {
        if (mainchannel == null) {
            mainchannel=init.getChannel();
            pp2pinit.start();
        } else {
            if (init.getChannel() == pp2pinit) {
                pp2pchannel=init.getChannel();
            }
        }

        init.go();
    }

    private void handleProcessInit(ProcessInitEvent event) {
        correct=event.getProcessSet();
        init();
        event.go();
    }

    private void init() {
        int i;
        for (i=0 ; i < NUM_REGISTERS ; i++) {
            value[i]=null;
            writeSet[i]=new HashSet();
        }
    }

    private void handleCrash(Crash event) {
        correct.setCorrect(event.getCrashedProcess(), false);
        event.go();

        allCorrect();
    }

    private void handleSharedRead(SharedRead event) {
        SharedReadReturn ev=new SharedReadReturn(mainchannel, Direction.UP, this);
        ev.reg=event.reg;
        ev.value=value[event.reg];
        ev.go();
    }
```

```
    private void handleSharedWrite(SharedWrite event) {
        WriteEvent ev=new WriteEvent(mainchannel, Direction.DOWN, this);
        ev.getMessage().pushObject(event.value);
        ev.getMessage().pushInt(event.reg);
        ev.go();
    }

    private void handleWriteEvent(WriteEvent event) {
        int reg=event.getMessage().popInt();
        Object val=event.getMessage().popObject();

        value[reg]=val;

        AckEvent ev=new AckEvent(pp2pchannel, Direction.DOWN, this);
        ev.getMessage().pushInt(reg);
        ev.dest=event.source;
        ev.go();
    }

    private void handleAckEvent(AckEvent event) {
        SampleProcess p_j=correct.getProcess((SocketAddress)event.source);
        int reg=event.getMessage().popInt();

        writeSet[reg]. add(p_j);

        allCorrect ();
    }

    private void allCorrect() {
        int reg;
        for (reg=0 ; reg < NUM_REGISTERS ; reg++) {
            boolean allAcks=true;
            int i;
            for (i=0 ; (i < correct.getSize()) && allAcks ; i++) {
                SampleProcess p=correct.getProcess(i);
                if (p.isCorrect() && !writeSet[reg].contains(p))
                    allAcks=false;
            }
            if (allAcks) {
                writeSet[reg]. clear ();

                SharedWriteReturn ev=new SharedWriteReturn(mainchannel, Direction.UP, this);
                ev.reg=reg;
                ev.go();

            }
        }
    }
}
```

Try It

1. Setup
   a) Open three shells/command prompts.
   b) In each shell go to the directory where you have placed the supplied code.
   c) In each shell launch the test application, *SampleAppl*, giving a different n value (0, 1, or 2) and specifying the *qos* as **r1nr**.

- In shell 0, execute

  ```
  ./run.sh -f etc/procs -n 0 -qos r1nr
  ```

- In shell 1, execute

  ```
  ./run.sh -f etc/procs -n 1 -qos r1nr
  ```

- In shell 2, execute

  ```
  ./run.sh  -f etc/procs -n 2 -qos r1nr
  ```

d) If the error NoClassDefError has appeared, confirm that you are at the root of the supplied code.

e) Start the *prefect failure detector* by writing **startpfd** in each shell.

2. Run: Now that processes are launched and running, let us try the following execution:

   a) In shell 0, write the value **S1** to register **2** (type **write 2 S1** and press Enter).

   b) In shell 1, read the value stored in register **2** (type **read 2** and press enter).
      - The shell displays that the value S1 is stored in register 2.

   c) In shell 0, write the value **S2** to register **5** (type **write 5 S2** and press Enter).

   d) In shell 1, write the value **S5** to register **5** (type **write 5 S5** and press Enter).

   e) In shell 0, read the value stored in register **5** (type **read 5** and press Enter).
      - Despite the fact that process 0 has written the value S2, the displayed content of register 5 is S5 because process 1 has afterward written to that register, .

4.2 $(1, N)$ Atomic Register

The $(1, N)$ Atomic Register algorithm implemented was the *Read-Impose Write-All*. The communication stack used to implement this protocol is the following:

Application
$(1, N)$**AtomicRegister** **(implemented by Read-Impose Write-All)**
Perfect Failure Detector (implemented by TcpBasedPFD)
Best-Effort Broadcast (implemented by Basic Broadcast)
Perfect Point-to-Point Links (implemented by TcpBasedPerfectP2P)

The protocol implementation is depicted in Listing 4.2. It follows the Algorithms in the book very closely. The only significant difference is that values are again generic instead of integers, thus allowing the registers to contain *strings*.

Listing 4.2. Read-Impose Write-All $(1, N)$ Atomic Register implementation

```
package irdp.protocols.tutorialDA.readImposeWriteAll1NAR;

public class ReadImposeWriteAll1NARSession extends Session {
   public static final int NUM_REGISTERS = 20;

   public ReadImposeWriteAll1NARSession(Layer layer) {
      super(layer);
   }

   private Object[] v = new Object[NUM_REGISTERS];
   private int[] ts = new int[NUM_REGISTERS];
   private int[] sn = new int[NUM_REGISTERS];
   private Object[] readval = new Object[NUM_REGISTERS];
   private int[] rqid = new int[NUM_REGISTERS];
   private boolean[] reading = new boolean[NUM_REGISTERS];
   private HashSet[] writeSet = new HashSet[NUM_REGISTERS];
   private ProcessSet correct = null;
   private Channel mainchannel = null;
   private Channel pp2pchannel = null;
   private Channel pp2pinit = null;

   public void handle(Event event) {
      if (event instanceof ChannelInit)
         handleChannelInit((ChannelInit) event);
      else if (event instanceof ProcessInitEvent)
         handleProcessInit((ProcessInitEvent) event);
      else if (event instanceof Crash)
         handleCrash((Crash) event);
      else if (event instanceof SharedRead)
         handleSharedRead((SharedRead) event);
      else if (event instanceof SharedWrite)
         handleSharedWrite((SharedWrite) event);
      else if (event instanceof WriteEvent)
         handleWriteEvent((WriteEvent) event);
      else if (event instanceof AckEvent)
         handleAckEvent((AckEvent) event);
      else {
         event.go();
      }
   }

   public void pp2pchannel(Channel c) {
      pp2pinit = c;
   }

   private void handleChannelInit(ChannelInit init) {
      if (mainchannel == null) {
         mainchannel = init.getChannel();
         pp2pinit. start ();
      } else {
         if ( init .getChannel() == pp2pinit) {
            pp2pchannel = init.getChannel();
         }
      }

      init .go();
   }
```

```
private void handleProcessInit(ProcessInitEvent event) {
    correct = event.getProcessSet();
    init ();
    event.go();
}

private void init() {
    int r;
    for (r = 0; r < NUM_REGISTERS; r++) {
        v[r] = readval[r] = null;
        ts[r] = sn[r] = rqid[r] = 0;
        reading[r] = false;
        writeSet[r] = new HashSet();
    }
}

private void handleCrash(Crash event) {
    correct.setCorrect(event.getCrashedProcess(), false);
    event.go();

    allCorrect ();
}

private void handleSharedRead(SharedRead event) {
    rqid[event.reg]++;
    reading[event.reg] = true;
    readval[event.reg] = v[event.reg];

    WriteEvent ev = new WriteEvent(mainchannel, Direction.DOWN, this);
    ev.getMessage().pushObject(v[event.reg]);
    ev.getMessage().pushInt(sn[event.reg]);
    ev.getMessage().pushInt(rqid[event.reg]);
    ev.getMessage().pushInt(event.reg);
    ev.go();
}

private void handleSharedWrite(SharedWrite event) {
    rqid[event.reg]++;
    ts[event.reg]++;

    WriteEvent ev = new WriteEvent(mainchannel, Direction.DOWN, this);
    ev.getMessage().pushObject(event.value);
    ev.getMessage().pushInt(ts[event.reg]);
    ev.getMessage().pushInt(rqid[event.reg]);
    ev.getMessage().pushInt(event.reg);
    ev.go();
}

private void handleWriteEvent(WriteEvent event) {
    int r = event.getMessage().popInt();
    int id = event.getMessage().popInt();
    int tstamp = event.getMessage().popInt();
    Object val = event.getMessage().popObject();

    if (tstamp > sn[r]) {
        v[r] = val;
        sn[r] = tstamp;
    }

    AckEvent ev = new AckEvent(pp2pchannel, Direction.DOWN, this);
    ev.getMessage().pushInt(id);
    ev.getMessage().pushInt(r);
    ev.dest = event.source;
    ev.go();
}
```

```
    private void handleAckEvent(AckEvent event) {
        SampleProcess p_j = correct.getProcess((SocketAddress) event.source);
        int r = event.getMessage().popInt();
        int id = event.getMessage().popInt();

        if (id == rqid[r]) {
            writeSet[r].add(p_j);

            allCorrect ();
        }
    }

    private void allCorrect() {
        int reg;
        for (reg = 0; reg < NUM_REGISTERS; reg++) {
            boolean allAcks = true;
            int i;
            for (i = 0; (i < correct.getSize()) && allAcks; i++) {
                SampleProcess p = correct.getProcess(i);
                if (p.isCorrect() && !writeSet[reg].contains(p))
                    allAcks = false;
            }
            if (allAcks) {
                writeSet[reg].clear ();

                if (reading[reg]) {
                    reading[reg] = false;

                    SharedReadReturn ev =
                        new SharedReadReturn(mainchannel, Direction.UP, this);
                    ev.reg = reg;
                    ev.value = readval[reg];
                    ev.go();
                } else {
                    SharedWriteReturn ev =
                        new SharedWriteReturn(mainchannel, Direction.UP, this);
                    ev.reg = reg;
                    ev.go();
                }
            }
        }
    }
}
```

Try It

Perform the same steps as suggested for the Regular $(1, N)$ Register. Please note that you should now specify the *qos* as **a1nr**.

4.3 (N, N) Atomic Register

The (N, N) Atomic Register algorithm implemented was the“Read-Impose Write-Consult.” The communication stack used to implement this protocol is the following:

Application
(N,N) **Atomic Register** **(implemented by Read-Impose Write-Consult**
Perfect Failure Detector (implemented by TcpBasedPFD)
Best-Effort Broadcast (implemented by Basic Broadcast)
Perfect Point-to-Point Links (implemented by TcpBasedPerfectP2P)

The protocol implementation is depicted in Listing 4.3. It follows Algorithms in the book very closely. The only significant difference is that values are again generic instead of integers.

Listing 4.3. Read-Impose Write-Consult (N,N) Atomic Register implementation

```
package irdp.protocols.tutorialDA.readImposeWriteConsultNNAR;

public class ReadImposeWriteConsultNNARSession extends Session {
    public static final int NUM_REGISTERS = 20;

    public ReadImposeWriteConsultNNARSession(Layer layer) {
        super(layer);
    }

    private ProcessSet correct = null;
    private int i = −1;
    private HashSet[] writeSet = new HashSet[NUM_REGISTERS];
    private boolean[] reading = new boolean[NUM_REGISTERS];
    private int[] reqid = new int[NUM_REGISTERS];
    private Object[] readval = new Object[NUM_REGISTERS];
    private Object[] v = new Object[NUM_REGISTERS];
    private int[] ts = new int[NUM_REGISTERS];
    private int[] mrank = new int[NUM_REGISTERS];

    private Channel mainchannel = null;
    private Channel pp2pchannel = null;
    private Channel pp2pinit = null;

    public void handle(Event event) {
         if (event instanceof ChannelInit)
            handleChannelInit((ChannelInit) event);
        else if (event instanceof ProcessInitEvent)
            handleProcessInit((ProcessInitEvent) event);
        else if (event instanceof Crash)
            handleCrash((Crash) event);
        else if (event instanceof SharedRead)
            handleSharedRead((SharedRead) event);
        else if (event instanceof SharedWrite)
            handleSharedWrite((SharedWrite) event);
        else if (event instanceof WriteEvent)
            handleWriteEvent((WriteEvent) event);
        else if (event instanceof AckEvent)
            handleAckEvent((AckEvent) event);
        else {
            event.go();
  }
    }

    public void pp2pchannel(Channel c) {
        pp2pinit = c;
```

```
}

private void handleChannelInit(ChannelInit init) {
    if (mainchannel == null) {
        mainchannel = init.getChannel();
        pp2pinit. start ();
    } else {
        if ( init .getChannel() == pp2pinit) {
            pp2pchannel = init.getChannel();
        }
    }
    init .go();
}

private void handleProcessInit(ProcessInitEvent event) {
    correct = event.getProcessSet();
    init ();
    event.go();
}

private void init() {
    i=correct.getSelfRank();

    int r;
    for (r = 0; r < NUM_REGISTERS; r++) {
        writeSet[r] = new HashSet();
        reqid[r] = ts[r] = 0;
        mrank[r] = -1;
        readval[r] = null;
        v[r] = null;
        reading[r] = false;
    }
}

private void handleCrash(Crash event) {
    correct .setCorrect(event.getCrashedProcess(), false);
    event.go();

    allAcked();
}

private void handleSharedRead(SharedRead event) {
    reqid[event.reg]++;
    reading[event.reg] = true;
    writeSet[event.reg]. clear ();
    readval[event.reg] = v[event.reg];

    WriteEvent ev = new WriteEvent(mainchannel, Direction.DOWN, this);
    ev.getMessage().pushObject(v[event.reg]);
    ev.getMessage().pushInt(mrank[event.reg]);
    ev.getMessage().pushInt(ts[event.reg ]);
    ev.getMessage().pushInt(reqid[event.reg ]);
    ev.getMessage().pushInt(event.reg);
    ev.go();
}

private void handleSharedWrite(SharedWrite event) {
    reqid[event.reg]++;
    writeSet[event.reg]. clear ();

    WriteEvent ev = new WriteEvent(mainchannel, Direction.DOWN, this);
    ev.getMessage().pushObject(event.value);
    ev.getMessage().pushInt(i);
    ev.getMessage().pushInt(ts[event.reg]+1);
    ev.getMessage().pushInt(reqid[event.reg ]);
    ev.getMessage().pushInt(event.reg);
```

```
        ev.go();
    }

    private void handleWriteEvent(WriteEvent event) {
        int r = event.getMessage().popInt();
        int id = event.getMessage().popInt();
        int t = event.getMessage().popInt();
        int j = event.getMessage().popInt();
        Object val = event.getMessage().popObject();

        if ((t > ts[r]) || ((t == ts[r]) && (j < mrank[r]))) {
            v[r]=val;
            ts[r]=t;
            mrank[r]=j;
        }

        AckEvent ev = new AckEvent(pp2pchannel, Direction.DOWN, this);
        ev.getMessage().pushInt(id);
        ev.getMessage().pushInt(r);
        ev.dest = event.source;
        ev.go();
    }

    private void handleAckEvent(AckEvent event) {
        SampleProcess p_j = correct.getProcess((SocketAddress) event.source);
        int r = event.getMessage().popInt();
        int id = event.getMessage().popInt();

        if (id == reqid[r]) {
            writeSet[r].add(p_j);

            allAcked();
        }
    }

    private void allAcked() {
        int reg;
        for (reg = 0; reg < NUM_REGISTERS; reg++) {
            boolean allAcks = true;
            int i;
            for (i = 0; (i < correct.getSize()) && allAcks; i++) {
                SampleProcess p = correct.getProcess(i);
                if (p.isCorrect() && !writeSet[reg].contains(p))
                    allAcks = false;
            }
            if (allAcks) {
                writeSet[reg].clear();

                if (reading[reg]) {
                    reading[reg] = false;

                    SharedReadReturn ev =
                        new SharedReadReturn(mainchannel, Direction.UP, this);
                    ev.reg = reg;
                    ev.value = readval[reg];
                    ev.go();
                } else {
                    SharedWriteReturn ev =
                        new SharedWriteReturn(mainchannel, Direction.UP, this);
                    ev.reg = reg;
                    ev.go();
                }
            }
        }
    }
}
```

Try It

Perform the same steps as suggested for the Regular $(1, N)$ Register. Please note that you should now specify the *qos* as **annr**.

5. Consensus

5.1 Flooding Regular Consensus Protocol

The communication stacks used to implement the regular *flooding* consensus protocol is depicted in the following:

Application
Consensus **(implemented by Flooding Consensus)**
Perfect Failure Detector (implemented by TcpBasedPFD)
Best-Effort Broadcast (implemented by Basic Broadcast)
Perfect Point-to-Point Links (implemented by TcpBasedPerfectP2P)

The FloodingConsensus layer implements the flooding consensus algorithm. It follows the Algorithm in the book very closely. It operates in rounds, and in each round it tries to gather the proposals from all correct members. This is achieved by each member sending all the proposals he knows. If a member fails, it advances to the next round. If in a round it gathers messages from all other correct processes, it decides by choosing in a deterministic way. In the implementation, all proposals must derive from the Proposal class, which forces the proposals to implement the int compareTo(Object o) method that allows comparison between proposals. The implemented algorithm chooses the lowest proposal. For instance, the proposal sent by the test application consists of a String, and therefore the algorithm chooses the one with the lowest lexicographical value. When the decision is made, it is also broadcasted by all members.

The protocol implementation is depicted in Listing 5.1.

Listing 5.1. Flooding Regular Consensus implementation

```
package irdp.protocols.tutorialDA.floodingConsensus;

public class FloodingConsensusSession extends Session {

    public FloodingConsensusSession(Layer layer) {
        super(layer);
    }
```

```
private int round=0;
private ProcessSet correct=null;
private Comparable decided=null;
private HashSet[] correct_this_round=null;
private HashSet[] proposal_set=null;

public void handle(Event event) {
    if (event instanceof ProcessInitEvent)
        handleProcessInit((ProcessInitEvent)event);
    else if (event instanceof Crash)
        handleCrash((Crash)event);
    else if (event instanceof ConsensusPropose)
        handleConsensusPropose((ConsensusPropose)event);
    else if (event instanceof MySetEvent)
        handleMySet((MySetEvent)event);
    else if (event instanceof DecidedEvent)
        handleDecided((DecidedEvent)event);
    else {
        event.go();
    }
}

private void init() {
    int max_rounds=correct.getSize()+1;
    correct_this_round=new HashSet[max_rounds];
    proposal_set=new HashSet[max_rounds];
    int i;
    for (i=0 ; i < max_rounds ; i++) {
        correct_this_round[i]=new HashSet();
        proposal_set[i]=new HashSet();
    }
    for (i=0 ; i < correct.getSize() ; i++) {
        SampleProcess p=correct.getProcess(i);
        if (p.isCorrect())
            correct_this_round [0]. add(p);
    }
    round=1;
    decided=null;

    count_decided=0;
}

private void handleProcessInit(ProcessInitEvent event) {
    correct=event.getProcessSet();
    init ();
    event.go();
}

private void handleCrash(Crash crash) {
    correct.setCorrect(crash.getCrashedProcess(),false);
    crash.go();

    decide(crash.getChannel());
}

private void handleConsensusPropose(ConsensusPropose propose) {
    proposal_set[round].add(propose.value);

    MySetEvent ev=new MySetEvent(propose.getChannel(),Direction.DOWN,this);
    ev.getMessage().pushObject(proposal_set[round]);
    ev.getMessage().pushInt(round);
    ev.go();

    decide(propose.getChannel());
}
```

```
private void handleMySet(MySetEvent event) {
    SampleProcess p_i=correct.getProcess((SocketAddress)event.source);
    int r=event.getMessage().popInt();
    HashSet set=(HashSet)event.getMessage().popObject();

    correct_this_round [r ]. add(p_i);
    proposal_set [r ]. addAll(set );

    decide(event.getChannel());
}

private void decide(Channel channel) {
    int i ;

    if (decided != null)
        return;

    for (i=0 ; i < correct.getSize() ; i++) {
        SampleProcess p=correct.getProcess(i);
        if ((p != null) && p.isCorrect() && !correct_this_round[round].contains(p))
            return;
    }

    if ( correct_this_round [round].equals( correct_this_round [round−1])) {
        Iterator iter =proposal_set[round]. iterator ();
        while (iter .hasNext()) {
            Comparable proposal=(Comparable)iter.next();
            if (decided == null)
                decided=proposal;
            else
                if (proposal.compareTo(decided) < 0)
                    decided=proposal;
        }

        ConsensusDecide ev=new ConsensusDecide(channel,Direction.UP,this);
        ev. decision=(Proposal)decided;
        ev.go();

        DecidedEvent ev=new DecidedEvent(channel,Direction.DOWN,this);
        ev.getMessage().pushObject(decided);
        ev.go();
    } else {
        round++;
        proposal_set [round].addAll(proposal_set[round−1]);

        MySetEvent ev=new MySetEvent(channel,Direction.DOWN,this);
        ev.getMessage().pushObject(proposal_set[round]);
        ev.getMessage().pushInt(round);
        ev.go();

        count_decided=0;
    }
}

private void handleDecided(DecidedEvent event) {
    // Counts the number os Decided messages received and reinitiates the algorithm
    if ((++count_decided >= correctSize()) && (decided != null)) {
        init ();
        return;
    }

    if (decided != null)
        return;

    SampleProcess p_i=correct.getProcess((SocketAddress)event.source);
```

```
        if (!p_i.isCorrect())
            return;

        decided=(Comparable)event.getMessage().popObject();

        ConsensusDecide ev=new ConsensusDecide(event.getChannel(),Direction.UP,this);
        ev.decision=(Proposal)decided;
        ev.go();

        DecidedEvent ev=new DecidedEvent(event.getChannel(),Direction.DOWN,this);
        ev.getMessage().pushObject(decided);
        ev.go();

        round=0;
    }

    // Used to count the number of Decided messages received, therefore determining when
    // all processes have decided and therefore allow a new decision process.
    private int count_decided;
    private int correctSize() {
        int size=0,i;
        SampleProcess[] processes=correct.getAllProcesses();
        for (i=0 ; i < processes.length ; i++) {
            if ((processes[i] != null) && processes[i].isCorrect())
                ++size;
        }
        return size;
    }
}
```

Try It

1. Setup
   a) Open three shells/command prompts.
   b) In each shell, go to the directory where you have placed the supplied code.
   c) In each shell, launch the test application, *SampleAppl*, giving a different *n* value (0, 1, or 2) and specifying the *qos* as **fc**.
      - In shell 0, execute
        ```
        ./run.sh -f etc/procs -n 0 -qos fc
        ```
      - In shell 1, execute
        ```
        ./run.sh -f etc/procs  -n 1  -qos fc
        ```
      - In shell 2, execute
        ```
        ./run.sh  -f etc/procs -n 2 -qos fc
        ```
   d) If the error NoClassDefError has appeared, confirm that you are at the root of the supplied code.
   e) Start the *prefect failure detector* by writing **startpfd** in each shell.
2. Run: Now that the processes are launched and running, let us try this execution:

a) In shell 0, propose the value **B** (type **consensus B** and press Enter).
b) In shell 1, propose the value **C** (type **consensus C** and press Enter).
c) In shell 2, propose the value **D** (type **consensus D** and press Enter).
d) All processes display that a decision was made and that it is **B**.
e) Wait a while to ensure that all messages related to the last decision are sent and received, and do not interfere with the next decision.
f) In shell 0, propose the value **E**.
g) In shell 1, propose the value **F**.
h) Note that a decision has not been made yet.
i) In shell 2, kill the test process.
j) The remaining processes display that a decision was made and that it is **E**. When the failure notification reaches them they start another round without the failed process.

5.2 Hierarchical Regular Consensus Protocol

The communication stacks used to implement the regular *hierarchical* consensus protocol is depicted in the following:

Application
Consensus **(implemented by Hierarchical Consensus)**
Perfect Failure Detector (implemented by TcpBasedPFD)
Best-Effort Broadcast (implemented by Basic Broadcast)
Perfect Point-to-Point Links (implemented by TcpBasedPerfectP2P)

The HierarchicalConsensus layer implements the hierarchical consensus algorithm. It follows the Algorithm in the book very closely. It also operates in rounds, and in each round one of the members chooses a proposal, either the one chosen in the previous round, if such a choice was made, or is own proposal. The process that chooses in each round is the one with its *rank* equal to the round. For this reason the first round is round 0. The protocol implementation is depicted in Listing 5.2.

Listing 5.2. Hierarchical Regular Consensus implementation

```
package irdp.protocols.tutorialDA.hierarchicalConsensus;

public class HierarchicalConsensusSession extends Session {

    public HierarchicalConsensusSession(Layer layer) {
        super(layer);
    }

    private int round=−1;
    private int prop_round=−1;
    private ProcessSet processes=null;
    private HashSet suspected=new HashSet();
    private boolean[] broadcast=null;
    private boolean[] delivered=null;
    private Comparable proposal=null;

    public void handle(Event event) {
        if (event instanceof ProcessInitEvent)
            handleProcessInit((ProcessInitEvent)event);
        else if (event instanceof Crash)
            handleCrash((Crash)event);
        else if (event instanceof ConsensusPropose)
            handleConsensusPropose((ConsensusPropose)event);
        else if (event instanceof DecidedEvent)
            handleDecided((DecidedEvent)event);
        else {
            event.go();
        }
    }

    private void init() {
        int max_rounds=processes.getSize();

        //suspected
        round=0;
        proposal=null;
        prop_round=−1;

        delivered=new boolean[max_rounds];
        Arrays. fill (delivered ,false );
        broadcast=new boolean[max_rounds];
        Arrays. fill (broadcast,false );
    }

    private void handleProcessInit(ProcessInitEvent event) {
        processes=event.getProcessSet();
        init ();
        event.go();
    }

    private void handleCrash(Crash crash) {
        processes.setCorrect(crash.getCrashedProcess(),false);
        suspected.add(new Integer(crash.getCrashedProcess()));
        crash.go();

        suspected_or_delivered ();
        decide(crash.getChannel());
    }

    private void handleConsensusPropose(ConsensusPropose propose) {
        if (proposal != null)
            return;

        proposal=propose.value;

        decide(propose.getChannel());
    }

    private void decide(Channel channel) {
        if (broadcast[round])
            return;
        if (proposal == null)
            return;
        if (round != processes.getSelfRank())
            return;
        broadcast[round]=true;
        ConsensusDecide ev=new ConsensusDecide(channel,Direction.UP,this);
        ev.decision=(Proposal)proposal;
        ev.go();

        DecidedEvent ev=new DecidedEvent(channel,Direction.DOWN,this);
        ev.getMessage().pushObject(proposal);
        ev.getMessage().pushInt(round);
        ev.go();
    }

    private void suspected_or_delivered() {
        if (suspected.contains(new Integer(round)) || delivered [round])
            round++;

        if (round >= delivered.length) {
            init ();
        }
    }

    private void handleDecided(DecidedEvent event) {
        SampleProcess p_i=processes.getProcess((SocketAddress)event.source);
        int r=event.getMessage().popInt();
        Comparable v=(Comparable)event.getMessage().popObject();

        if ((r < processes.getSelfRank()) && (r > prop_round)) {
            proposal=v;
            prop_round=r;
        }
        delivered [r]=true;

        suspected_or_delivered ();
        decide(event.getChannel());
    }
}
```

Try It

1. Setup
   a) Open three shells/command prompts.
   b) In each shell, go to the directory where you have placed the supplied code.
   c) In each shell, launch the test application, *SampleAppl*, as with the flooding algorithm, but specifying the *qos* as **hc**.
   - In shell 0, execute

     ```
     ./run.sh -f etc/procs  -n 0 -qos hc
     ```

   - In shell 1, execute

     ```
     ./run.sh -f etc/procs -n 1 -qos hc
     ```

   - In shell 2, execute

     ```
     ./run.sh  -f etc/procs  -n 2  -qos hc
     ```

   d) If the error NoClassDefError has appeared, confirm that you are at the root of the supplied code.
   e) Start the *prefect failure detector* by typing **startpfd** in each shell.

2. Run: Now that the processes are launched and running, let us try this execution:
   a) In shell 0, propose the value **B** (type **consensus B** and press Enter).
   b) Note that all processes decide **B**. Because the proposal came from the process with the lowest rank, a decision is almost immediate.
   c) In shell 2, propose the value **G**.
   d) Note that no decision as yet been made.
   e) In shell 1, propose the value **H**.
   f) Again, note that no decision as yet been made.
   g) In shell 0, propose the value **I**.
   h) All processes decide **I**.
   i) In shell 1, propose the value **J**.
   j) No decision has yet been made.
   k) In shell 0, kill the test process.
   l) The remaining processes display that a decision was made and that it is **J**. Because **J** was proposed by the living process with the lowest rank, as soon as it is detected that all other processes with lower ranks have failed, the proposal becomes a decision.

5.3 Flooding Uniform Consensus

The communication stack used to implement the flooding uniform consensus protocol is depicted in the following:

SampleAppl
Uniform Consensus **(implemented by Flooding UC)**
Perfect Failure Detector (implemented by TcpBasedPFD)
Best-Effort Broadcast (implemented by Basic Broadcast)
Perfect Point-To-Point Links (implemented by TcpBasedPerfectP2P)

The *FloodingUniformConsensus* implements the uniform flooding consensus algorithm. It follows the Algorithm in the book very closely. Its main difference when compared with the regular flooding algorithm is that it runs through N rounds, N being the number of processes. Due to this, it is not necessary to broadcast the decision. The protocol implementation is depicted in Listing 5.3.

Listing 5.3. Flooding Uniform Consensus implementation

```
package irdp.protocols.tutorialDA.floodingUniformConsensus;

public class FloodingUniformConsensusSession extends Session {
    public FloodingUniformConsensusSession(Layer layer) {
        super(layer);
    }

    private int round=-1;
    private ProcessSet correct=null;
    private Comparable decided=null;
    private HashSet[] delivered=null;
    private HashSet proposal_set=null;

    public void handle(Event event) {
        if (event instanceof ProcessInitEvent)
            handleProcessInit((ProcessInitEvent)event);
        else if (event instanceof Crash)
            handleCrash((Crash)event);
        else if (event instanceof ConsensusPropose)
            handleConsensusPropose((ConsensusPropose)event);
        else if (event instanceof MySetEvent)
            handleMySet((MySetEvent)event);
        else {
            event.go();
        }
    }

    private void init() {
        int max_rounds=correct.getSize();
        delivered=new HashSet[max_rounds];
        proposal_set=new HashSet();
        int i;
        for (i=0 ; i < max_rounds ; i++) {
            delivered[i]=new HashSet();
        }
        round=0;
        decided=null;
    }

    private void handleProcessInit(ProcessInitEvent event) {
        correct=event.getProcessSet();
        init();
        event.go();
    }

    private void handleCrash(Crash crash) {
        correct.setCorrect(crash.getCrashedProcess(),false);
        crash.go();

        decide(crash.getChannel());
    }

    private void handleConsensusPropose(ConsensusPropose propose) {
        proposal_set.add(propose.value);

        MySetEvent ev=new MySetEvent(propose.getChannel(),Direction.DOWN,this);
        ev.getMessage().pushObject(proposal_set);
        ev.getMessage().pushInt(round);
        ev.go();
    }

    private void handleMySet(MySetEvent event) {
        SampleProcess p_i=correct.getProcess((SocketAddress)event.source);
        int r=event.getMessage().popInt();
        HashSet newSet=(HashSet)event.getMessage().popObject();
        proposal_set.addAll(newSet);
        delivered[r].add(p_i);

        decide(event.getChannel());
    }

    private void decide(Channel channel) {
        int i;

        if (decided != null)
            return;

        for (i=0 ; i < correct.getSize() ; i++) {
            SampleProcess p=correct.getProcess(i);
            if ((p != null) && p.isCorrect() && !delivered[round].contains(p))
                return;
        }

        if (round == delivered.length−1) {
            Iterator iter=proposal_set.iterator();
            while (iter.hasNext()) {
                Comparable proposal=(Comparable)iter.next();
                if (decided == null)
                    decided=proposal;
                else
                    if (proposal.compareTo(decided) < 0)
                        decided=proposal;
            }

            ConsensusDecide ev=new ConsensusDecide(channel,Direction.UP,this);
            ev.decision=(Proposal)decided;
            ev.go();

            init();
        } else {
            round++;

            MySetEvent ev=new MySetEvent(channel,Direction.DOWN,this);
            ev.getMessage().pushObject(proposal_set);
            ev.getMessage().pushInt(round);
            ev.go();
        }
    }
}
```

Try It

The same executions suggested for flooding regular consensus can be experimented with. Remember to specify the *qos* as **ufc**.

5.4 Hierarchical Uniform Consensus

The communication stack used to implement the uniform hierarchical consensus protocol is depicted in the following:

Application
Uniform Consensus (implemented by Hierarchical UC)
Perfect Failure Detector (implemented by TcpBasedPFD)
Best-Effort Broadcast (implemented by Basic Broadcast)
Perfect Point-to-Point Links (implemented by TcpBasedPerfectP2P)

This implementation uses two different *Appia* channels because it requires *best-effort broadcast* and *perfect point-to-point links*, and each channel can only offer one of those properties, despite the fact that best-effort broadcast uses perfect point-to-point links. The *HierarchicalUniformConsensus* layer implements the uniform hierarchical consensus algorithm. It follows Algorithm in the book very closely. The protocol implementation is depicted in Listing 5.4.

Listing 5.4. Hierarchical Uniform Consensus implementation

```
package irdp.protocols.tutorialDA.hierarchicalUniformConsensus;

public class HierarchicalUniformConsensusSession extends Session {
    public HierarchicalUniformConsensusSession(Layer layer) {
        super(layer);
    }

    private Comparable proposal=null;
    private Comparable decided=null;
    private int round=−1;
    private HashSet suspected=new HashSet();
    private HashSet ack_set=new HashSet();
    private int prop_round=−1;
    private ProcessSet processes=null;

    private Channel mainchannel=null;
    private Channel rbchannel=null;
    private Channel rbinit=null;

    public void handle(Event event) {
        if (event instanceof ChannelInit)
            handleChannelInit((ChannelInit)event);
        else if (event instanceof ProcessInitEvent)
            handleProcessInit((ProcessInitEvent)event);
        else if (event instanceof Crash)
            handleCrash((Crash)event);
        else if (event instanceof ConsensusPropose)
            handleConsensusPropose((ConsensusPropose)event);
        else if (event instanceof ProposeEvent)
            handleProposeEvent((ProposeEvent)event);
        else if (event instanceof DecidedEvent)
            handleDecided((DecidedEvent)event);
        else {
            event.go();
        }
    }

    public void rbchannel(Channel c) {
        rbinit=c;
    }

    private void handleChannelInit(ChannelInit init) {
        if (mainchannel == null) {
            mainchannel=init.getChannel();
            rbinit.start();
        } else {
            if (init.getChannel() == rbinit) {
                rbchannel=init.getChannel();

                if (processes != null) {
                    ProcessInitEvent ev=new ProcessInitEvent(rbchannel,Direction.DOWN,this);
                    ev.setProcessSet(processes);
                    ev.go();
                } catch (AppiaEventException ex) {
                    ex.printStackTrace();
                }
            }
        }

        init.go();
    }

    private void handleProcessInit(ProcessInitEvent event) {
        processes=event.getProcessSet();
        init();
        event.go();

        if (rbchannel != null) {
            ProcessInitEvent ev=new ProcessInitEvent(rbchannel,Direction.DOWN,this);
            ev.setProcessSet(processes);
            ev.go();
        }
    }

    private void init() {
        int max_rounds=processes.getSize();

        proposal=null;
        decided=null;
        round=0;
        //suspected
        ack_set=new HashSet();
        prop_round=-1;

        count_decided=0;
    }

    private void handleCrash(Crash crash) {
        processes.setCorrect(crash.getCrashedProcess(),false);
        suspected.add(new Integer(crash.getCrashedProcess()));

        crash.go();

        suspected_or_acked();
        propose();
        decide();
    }

    private void handleConsensusPropose(ConsensusPropose propose) {
        if (proposal != null)
            return;

        proposal=propose.value;

        propose();
```

```
}

private void propose() {
    if (decided != null)
        return;
    if (proposal == null)
        return;
    if (round != processes.getSelfRank())
        return;

    ProposeEvent ev=new ProposeEvent(mainchannel,Direction.DOWN,this);
    ev.getMessage().pushObject(proposal);
    ev.getMessage().pushInt(round);
    ev.go();
}

private void handleProposeEvent(ProposeEvent event) {
    int p_i_rank=processes.getRank((SocketAddress)event.source);
    int r=event.getMessage().popInt();
    Comparable v=(Comparable)event.getMessage().popObject();

    ack_set.add(new Integer(p_i_rank));
    if ((r < processes.getSelfRank()) && (r > prop_round)) {
        proposal=v;
        prop_round=r;
    }

    suspected_or_acked();
    propose();
    decide();
}

private void suspected_or_acked() {
    if (suspected.contains(new Integer(round)) || ack_set.contains(new Integer(round)))
        round++;
}

private void decide() {
    int i;
    for (i=0 ; i < processes.getSize() ; i++) {
        int p_i_rank=processes.getProcess(i).getProcessNumber();
         if (!suspected.contains(new Integer(p_i_rank)) &&
            !ack_set.contains(new Integer(p_i_rank))) {
            return;
        }
    }

    DecidedEvent ev=new DecidedEvent(rbchannel,Direction.DOWN,this);
    ev.getMessage().pushObject(proposal);
    ev.go();
}

private void handleDecided(DecidedEvent event) {
    // Counts the number os Decided messages received and reinitiates the algorithm
    if ((++count_decided >= correctSize()) && (decided != null)) {
         init();
        return;
    }

    if (decided != null)
        return;

    decided=(Comparable)event.getMessage().popObject();

    ConsensusDecide ev=new ConsensusDecide(mainchannel,Direction.UP,this);
    ev.decision=(Proposal)decided;
```

```
        ev.go();
    }

    // Used to count the number of Decided messages received, therefore determining when
    // all processes have decided and therefore allow a new decision process.
    private int count_decided;
    private int correctSize() {
        int size=0,i;
        for (i=0 ; i < processes.getSize() ; i++) {
            if ((processes.getProcess(i) != null) && processes.getProcess(i).isCorrect())
                ++size;
        }
        return size;
    }
}
```

Try It

The same executions suggested for hierarchical regular consensus can be experimented with. Remember to specify the *qos* as **uhc**.

6. Consensus Variants

6.1 Uniform Total Order Broadcast

The communication stack used to implement the protocol is the following:

Application
Uniform Total Order **(implemented by Consensus-Based UTO)**
Delay
Uniform Consensus (implemented by Flooding UC)
Uniform Reliable Broadcast (implemented by All-Ack URB)
Perfect Failure Detector (implemented by TcpBasedPFD)
Best-Effort Broadcast (implemented by Basic Broadcast)
Perfect Point-to-Point Links (implemented by TcpBasedPerfectP2P)

The protocol implementation is depicted in Listing 6.1.

Listing 6.1. Uniform Total Order Broadcast implementation

```
SocketAddresspackage irdp.protocols.tutorialDA.consensusUTO;

public class ConsensusUTOSession extends Session{

    SocketAddress iwp;
    Channel channel;
    /*global sequence number of the message ssent by this process*/
    int seqNumber;
    /*Sequence number of the set of messages to deliver in the same round!*/
    int sn;
    /*Sets the beginning and the end of the rounds*/
    boolean wait;
    /* Set of delivered messages. */
    LinkedList delivered;
    /* Set of unordered messages. */
    LinkedList unordered;

    public ConsensusTOSession(Layer l) {
        super(l);
    }
```

```
public void handle(Event e) {
    if (e instanceof ChannelInit)
        handleChannelInit((ChannelInit)e);
    else if(e instanceof ProcessInitEvent)
        handleProcessInitEvent((ProcessInitEvent)e);
    else if (e instanceof SendableEvent){
        if (e.getDir()==Direction.DOWN)
            handleSendableEventDOWN((SendableEvent)e);
        else
            handleSendableEventUP((SendableEvent)e);
    } else if (e instanceof ConsensusDecide)
        handleConsensusDecide((ConsensusDecide)e);
    else{
        e.go();
    }
}

public void handleChannelInit (ChannelInit e){
    e.go();

    this.channel = e.getChannel();

    delivered=new LinkedList();
    unordered=new LinkedList();

    sn=1;
    wait=false;
}

public void handleProcessInitEvent (ProcessInitEvent e){
    iwp=e.getProcessSet().getSelfProcess().getAddress();
    e.go();
}

public void handleSendableEventDOWN (SendableEvent e){
    Message om=e.getMessage();
    //inserting the global seq number of this msg
    om.pushInt(seqNumber);
    e.go();

    //increments the global seq number
    seqNumber++;
}

public void handleSendableEventUP (SendableEvent e){
    Message om=e.getMessage();
    int seq=om.popInt();

    //checks if the msg has already been delivered.
    ListElement le;
    if (! isDelivered ((SocketAddress)e.source,seq)){
        le=new ListElement(e,seq);
        unordered.add(le);
    }

    //let's see if we can start a new round!
    if(unordered.size()!=0 && !wait){
        wait=true;
        //sends our proposal to consensus protocol!
        ConsensusPropose cp;
        byte[] bytes=null;
        cp = new ConsensusPropose(channel, Direction.DOWN, this);
        bytes=serialize(unordered);

        OrderProposal op=new OrderProposal(bytes);
        cp.value=op;

        cp.go();
    }
}

public void handleConsensusDecide (ConsensusDecide e){
    LinkedList decided=deserialize(((OrderProposal)e.decision).bytes);

    //The delivered list must be complemented with the msg in the decided list!
    for(int i=0;i<decided.size();i++){
        if(!isDelivered( (SocketAddress)((ListElement)decided.get(i)).se.source,
                         ((ListElement)decided.get(i)).seq )){
            //if a msg that is in decided doesn't yet belong to delivered, add it!
            delivered.add(decided.get(i));
        }
    }

    //update unordered list by removing the messages that are in the delivered list
    for(int j=0;j<unordered.size();j++){
        if(isDelivered( (SocketAddress)((ListElement)unordered.get(j)).se.source,
                        ((ListElement)unordered.get(j)).seq )){
            unordered.remove(j);
            j--;
        }
    }

    decided=sort(decided);

    //deliver the messages in the decided list, which is already ordered!
    for(int k=0;k<decided.size();k++){
        ((ListElement)decided.get(k)).se.go();
    }
    sn++;
    wait=false;
}

boolean isDelivered(SocketAddress source,int seq){
    for(int k=0;k<delivered.size();k++){
        if( ((ListElement)delivered.get(k)).getSE().source.equals(source) &&
            ((ListElement)delivered.get(k)).getSeq()==seq)
            return true;
    }

    return false;
}

LinkedList sort(LinkedList list){
    return list;
}

byte[] intToByteArray(int i) {
    byte[] ret = new byte[4];

    ret[0] = (byte) ((i & 0xff000000) >> 24);
    ret[1] = (byte) ((i & 0x00ff0000) >> 16);
    ret[2] = (byte) ((i & 0x0000ff00) >> 8);
    ret[3] = (byte) (i & 0x000000ff);

    return ret;
}

int byteArrayToInt(byte[] b, int off) {
    int ret = 0;

    ret |= b[off] << 24;
```

```
        ret |= (b[off+1] << 24) >>> 8; // must be done this way because of
        ret |= (b[off+2] << 24) >>> 16; // java's sign extension of <<
        ret |= (b[off+3] << 24) >>> 24;

        return ret;
    }

    private byte[] serialize(LinkedList list) {
        ByteArrayOutputStream data=new ByteArrayOutputStream();
        byte[] bytes=null;

        //number of elements of the list:int
        data.write(intToByteArray(list.size()));

        //now, serialize each element
        for(int i=0;i<list.size();i++){
            //getting the list element
            ListElement le=(ListElement)list.get(i);
            //sequence number:int
            data.write(intToByteArray(le.seq));
            //class name
            bytes=le.se.getClass().getName().getBytes();
            data.write(intToByteArray(bytes.length));
            data.write(bytes,0,bytes.length);
            //source port:int
            data.write(intToByteArray(((InetSocketAddress)le.se.source).getPort()));
            //source host:string
            String host=((InetSocketAddress)le.se.source).getHostName();
            bytes=host.getBytes();
            data.write(intToByteArray(bytes.length));
            data.write(bytes,0,bytes.length);
            // message
            bytes=le.se.getMessage().toByteArray();
            data.write(intToByteArray(bytes.length));
            data.write(bytes,0,bytes.length);
        }

        //creating the byte[]
        bytes=data.toByteArray();
        return bytes;
    }

    private LinkedList deserialize(byte[] data) {
        LinkedList ret=new LinkedList();
        int curPos=0;

        //getting the size of the list
        int listSize=byteArrayToInt(data, curPos);
        curPos+=4;

        //getting the elements of the list
        for(int i=0;i<listSize;i++){
            //seq number
            int seq=byteArrayToInt(data, curPos);
            curPos+=4;
            //class name
            int aux_size=byteArrayToInt(data, curPos);
            String className=new String(data,curPos+4,aux_size);
            curPos+=aux_size+4;
            //creating the event
            SendableEvent se=null;

            se = (SendableEvent) Class.forName(className).newInstance();
            // format known event attributes
            se.setDir(Direction.UP);
            se.setSourceSession(this);
            se.setChannel(channel);

            //source:port
            int port=byteArrayToInt(data, curPos);
            curPos+=4;
            //source:host
            aux_size=byteArrayToInt(data, curPos);
            String host=new String(data,curPos+4,aux_size);
            curPos+=aux_size+4;
            se.source=new InetSocketAddress(InetAddress.getByName(host),port);
            // finally, the message
            aux_size=byteArrayToInt(data, curPos);
            curPos+=4;

            se.getMessage().setByteArray(data,curPos,aux_size);
            curPos+=aux_size;
            se.init();
            //creating the element that is the unit of the list
            ListElement le=new ListElement(se,seq);
            //adding this element to the list to return
            ret.add(le);
        }
        return ret;
    }
}

class ListElement{
    SendableEvent se;
    int seq; /*sequence number*/

    public ListElement(SendableEvent se, int seq){
        this.se=se;
        this.seq=seq;
    }

    SendableEvent getSE(){
        return se;
    }

    int getSeq(){
        return seq;
    }
}
```

Try It

1. Uncomment the eighth line of the *getUnTOChannel* method in file SampleAppl.java, in package demo.tutorialDA. This will insert a test layer that allows the injection of delays in messages sent between process 0 and process 2. After modifying the file, it is necessary to compile it.
2. Open three shells/command prompts.
3. In each shell, go to the directory where you have placed the supplied code.
4. In each shell, launch the test application, SampleAppl, giving a different n value (0, 1, or 2) and specifying the *qos* as **uto**.
   - In shell 0, execute:

     ```
     ./run.sh  -f etc/procs  -n 0  -qos uto
     ```

- In shell 1, execute:

```
./run.sh  -f etc/procs -n 1 -qos uto
```

- In shell 2, execute:

```
./run.sh -f etc/procs  -n 2 -qos uto
```

Note: If the error NoClassDefError has appeared, confirm that you are at the root of the supplied code.

Now that processes are launched and running, you may try the following execution:

1. In shell 0, send a message **M1** (**bcast M1** and press Enter).
   - Note that no process received the message **M1**.
2. In shell 1, send a message **M2**.
   - Note that all processes received message **M1**. The consensus decided to deliver **M1**.
3. In shell 1, send a message **M3**.
   - Note that all processes received message **M2** and **M3**. Now, the consensus decided to deliver these both messages.
4. Confirm that all processes received **M1**, **M2**, and **M3** in the same order.
5. You can keep repeating these steps, in order to introduce some delays, and checking that all processes receive all messages in the same order.

6.2 Consensus-Based Non-blocking Atomic Commit

The communication stack used to implement the protocol is the following:

Application
NBAC **(implemented by Consensus-Based NBAC)**
Uniform Consensus (implemented by Flooding UC)
Perfect Failure Detector (implemented by TcpBasedPFD)
Best-Effort Broadcast (implemented by Basic Broadcast)
Perfect Point-to-Point Links (implemented by TcpBasedPerfectP2P)

The protocol implementation is depicted in Listing 6.2.

Listing 6.2. Consensus-based Non-Blocking Atomic Commit implementation

```
package irdp.protocols.tutorialDA.consensusNBAC;

public class ConsensusNBACSession extends Session {
    public ConsensusNBACSession(Layer layer) {
        super(layer);
    }

    public void handle(Event event) {
        if (event instanceof ProcessInitEvent)
            handleProcessInit((ProcessInitEvent)event);
        else if (event instanceof Crash)
            handleCrash((Crash)event);
        else if (event instanceof NBACPropose)
            handleNBACPropose((NBACPropose)event);
        else if (event instanceof ConsensusDecide)
            handleConsensusDecide((ConsensusDecide)event);
        else {
            event.go();
        }
    }

    private HashSet delivered=null;
    private ProcessSet correct=null;
    private int proposal;

    private void init() {
        delivered=new HashSet();
        proposal=1;
    }

    private void handleProcessInit(ProcessInitEvent event) {
        correct=event.getProcessSet();
        init ();
        event.go();
    }

    private void handleCrash(Crash crash) {
        correct .setCorrect(crash.getCrashedProcess(),false);
        crash.go();

        all_delivered (crash.getChannel());
    }

    private void handleNBACPropose(NBACPropose event) {
        if (event.getDir() == Direction.DOWN) {
            event.getMessage().pushInt(event.value);

            event.go();
        } else {
            SampleProcess p_i=correct.getProcess((SocketAddress)event.source);
            int v=event.getMessage().popInt();

            delivered.add(p_i);
            proposal*=v;

            all_delivered (event.getChannel());
        }
    }

    private void all_delivered(Channel channel) {
        boolean all_correct=true;
        int i;
        for (i=0 ; i < correct.getSize() ; i++) {
            SampleProcess p=correct.getProcess(i);
            if (p != null) {
                if (p.isCorrect()) {
                    if (!delivered.contains(p))
                        return;
                } else {
                    all_correct =false;
                }
            }
        }

        if (! all_correct )
            proposal=0;

        ConsensusPropose ev=new ConsensusPropose(channel, Direction.DOWN, this);
        ev.value=new IntProposal(proposal);
        ev.go();

        init ();
    }

    private void handleConsensusDecide(ConsensusDecide event) {
        NBACDecide ev=new NBACDecide(event.getChannel(), Direction.UP, this);
        ev.decision=((IntProposal)event.decision).i;
        ev.go();
    }
}
```

Try It

1. Setup
   a) Open three shells/command prompts.
   b) In each shell, go to the directory where you have placed the supplied code.
   c) In each shell, launch the test application, *SampleAppl*, giving a different *n* value (0, 1, or 2) and specifying the *qos* as **nbac**.
      - In shell 0, execute

        ```
        ./run.sh  -f etc/procs  -n 0 -qos nbac
        ```

      - In shell 1, execute

        ```
        ./run.sh -f etc/procs  -n 1 -qos nbac
        ```

      - In shell 2, execute

        ```
        ./run.sh -f etc/procs -n 2 -qos nbac
        ```

   d) If the error NoClassDefError has appeared, confirm that you are at the root of the supplied code.
   e) Start the *prefect failure detector* by typing **startpfd** in each shell.
2. Run: Now that processes are launched and running, let us try the following execution:
   a) In shell 0, propose **1** (type **atomic 1** and press enter). Any value different from 0 is considered 1, and 0 is 0.

b) In shell 1, propose **1**. (type **atomic 1** and press Enter).
c) In shell 2, propose **1**. (type **atomic 1** and press Enter).
d) Note that all processes *commit* the value **1**.
e) In shell 0, propose **1**.
f) In shell 1, propose **1**.
g) In shell 2, propose **0**. (type **atomic 0** and press Enter).
h) Note that all processes *commit* the value **0**.
i) In shell 1, propose **1**.
j) In shell 2, propose **1**.
k) In shell 0, kill the test application process.
l) Note that all processes *commit* the value **0**.

6.3 Consensus-Based Group Membership

The communication stack used to implement the protocol is the following:

Application
Group Membership **(Consensus-based GM)**
Uniform Consensus (implemented by Flooding UC)
Perfect Failure Detector (implemented by TcpBasedPFD)
Best-Effort Broadcast (implemented by Basic Broadcast)
Perfect Point-to-Point Links (implemented by TcpBasedPerfectP2P)

The protocol implementation is depicted in Listing 6.3. It follows the Algorithm in the book very closely. The *view* is represented by an object of class View from package appia.protocols.tutorialDA.membershipUtils. It contains two attributes, the members that compose it, and its *id*. The *id* is an integer. The View class extends Proposal, permitting the comparison between views and allowing its use as a proposal value for consensus. When comparing views, the lowest is the one with the highest id.

Listing 6.3. Consensus-Based Group Membership

```
package irdp.protocols.tutorialDA.consensusMembership;

public class ConsensusMembershipSession extends Session {
    public ConsensusMembershipSession(Layer layer) {
        super(layer);
    }

    public void handle(Event event) {
        if (event instanceof ProcessInitEvent) {
            handleProcessInit((ProcessInitEvent)event);
            return;
        }
        if (event instanceof Crash) {
            handleCrash((Crash)event);
            return;
        }
        if (event instanceof ConsensusDecide) {
            handleConsensusDecide((ConsensusDecide)event);
            return;
        }

        event.go();
    }

    private View view=null;
    private ProcessSet correct=null;
    private boolean wait;

    private void handleProcessInit(ProcessInitEvent event) {
        event.go();

        view=new View();
        view.id=0;
        view.memb=event.getProcessSet();
        correct=event.getProcessSet();
        wait=false;

        ViewEvent ev=new ViewEvent(event.getChannel(), Direction.UP, this);
        ev.view=view;
        ev.go();
    }

    private void handleCrash(Crash crash) {
        correct.setCorrect(crash.getCrashedProcess(),false);
        crash.go();

        newMembership(crash.getChannel());
    }

    private void newMembership(Channel channel) {
        if (wait)
            return;

        boolean crashed=false;
        int i;
        for (i=0 ; i < correct.getSize() ; i++) {
            SampleProcess p=correct.getProcess(i);
            SampleProcess m=view.memb.getProcess(p.getAddress());
            if (!p.isCorrect() && (m != null)) {
                crashed=true;
            }
        }
        if (!crashed)
            return;

        wait=true;

        int j;
        ProcessSet trimmed_memb=new ProcessSet();
        for (i=0,j=0 ; i < correct.getSize() ; i++) {
            SampleProcess p=correct.getProcess(i);
            if (p.isCorrect())
                trimmed_memb.addProcess(p,j++);
        }

        View v=new View();
        v.id=view.id+1;
        v.memb=trimmed_memb;

        ConsensusPropose ev=new ConsensusPropose(channel,Direction.DOWN,this);
        ev.value=v;
        ev.go();
    }

    private void handleConsensusDecide(ConsensusDecide event) {
        view=(View)event.decision;

        wait=false;

        ViewEvent ev=new ViewEvent(event.getChannel(),Direction.UP,this);
        ev.view=view;
        ev.go();
    }
}
```

Try It

1. Setup
   a) Open three shells/command prompts.
   b) In each shell, go to the directory where you have placed the supplied code.
   c) In each shell launch the test application, *SampleAppl*, giving a different *n* value (0, 1, or 2) and specifying the *qos* as **cmem**.
   - In shell 0, execute

     ```
     ./run.sh  -f etc/procs -n 0  -qos cmem
     ```

   - In shell 1, execute

     ```
     ./run.sh -f etc/procs -n 1 -qos cmem
     ```

   - In shell 2, execute

     ```
     ./run.sh -f etc/procs -n 2 -qos cmem
     ```

   d) If the error NoClassDefError has appeared, confirm that you are at the root of the supplied code.
   e) Start the *prefect failure detector* by typing **startpfd** in each shell.
2. Run: Now that processes are launched and running, let us try the following execution:
   a) Observe that an initial view with all members is shown.
   b) In shell 0, kill the test application process.
   c) Observe that a new view without the failed process is shown.
   d) Now, in shell 1, kill the test application process.
   e) Observe that in the remaining process a new view, with only one member, is delivered.

6.4 TRB-Based View Synchrony

The communication stack used to implement the protocol is the following:

Application
View Synchrony **(implemented by TRB-based VS)**
Membership (implemented by Consensus-based Membership)
Reliable Causal Order (implemented by Garbage Collection Of Past)
Reliable Broadcast (implemented by Lazy RB)
TRB **(implemented by Consensus-based TRB)**
Uniform Consensus (implemented by Flooding UC)
Perfect Failure Detector (implemented by TcpBasedPFD)
Best-Effort Broadcast (implemented by Basic Broadcast)
Perfect Point-to-Point Links (implemented by TcpBasedPerfectP2P)

The role of the significant layers is explained below.

ConsensusTRB. This layer implements the consensus-based terminating reliable broadcast algorithm. It follows the Algorithm in the book very closely. It functions by assuming that all processes propose the same message for delivery. The result is the message proposed or a failure notification. The layer receives the message proposal within an event of class TRBEvent. It then asks for a consensus decision, either proposing the message or a failure notification. The consensus decision is then notified upward in another TRBEvent.

TRBViewSync. This layer implements the TRB-based View Synchrony algorithm. It follows the Algorithm in the book closely. Whenever a new view is given by the underlying membership layer it flushes each member by requesting it to stop sending messages. This is done with a BlockEvent. When the application acknowledges, by sending a BlockOkEvent, it starts to disseminate the messages delivered. This is done in rounds, in which the process with rank equal to the round number disseminates the messages it has delivered, using the *Terminating Reliable Broadcast* primitive. To do this, the layer sends a TRBEvent either with the set, of delivered messages if the round number is equal to its rank, or with *null*. It then waits for the return TRBEvent, delivers any message not yet delivered contained in the received set and advances to the next round.

When all rounds are terminated the new view is sent upward. To identify the messages, so as to be able to determine if the message was already delivered or not, each message sent by a process is given a unique identifier, implemented as a simple sequence number. Therefore, the pair (*sender_process*, *id*) globally identifies each message.

The Consensus-Based Terminating Reliable Broadcast implementation is depicted in Listing 6.4 and the TRB-Based View Synchrony implementation is depicted in Listing 6.5.

Listing 6.4. Consensus-Based Terminating Reliable Broadcast implementation

```
package irdp.protocols.tutorialDA.consensusTRB;

public class ConsensusTRBSession extends Session {
    public ConsensusTRBSession(Layer layer) {
        super(layer);
    }

    public void handle(Event event) {
        if (event instanceof ProcessInitEvent)
            handleProcessInit((ProcessInitEvent) event);
        else if (event instanceof Crash)
            handleCrash((Crash) event);
        else if (event instanceof ConsensusDecide)
            handleConsensusDecide((ConsensusDecide) event);
        else if (event instanceof TRBSendableEvent)
            handleTRBSendableEvent((TRBSendableEvent) event);
        else if (event instanceof TRBEvent)
            handleTRBEvent((TRBEvent) event);
        else {
            event.go();
        }
    }

    private TRBProposal proposal;
    private ProcessSet correct = null;
    private SampleProcess src;

    private void handleProcessInit(ProcessInitEvent event) {
        event.go();

        correct = event.getProcessSet();
        init ();
    }

    private void init() {
        proposal = null;
        src = null;
        // correct filled at handleProcessInit
    }

    private void handleCrash(Crash crash) {
        correct.setCorrect(crash.getCrashedProcess(), false);
        crash.go();

        failed (crash.getChannel());
    }

    private void handleTRBEvent(TRBEvent event) {
        src = correct.getProcess(event.p);

        if (src. isSelf ()) {
            TRBSendableEvent ev =
                new TRBSendableEvent(event.getChannel(), Direction.DOWN, this);
            ev.getMessage().pushObject(event.m);
            ev.go();
        }

        failed (event.getChannel());
    }

    private void handleTRBSendableEvent(TRBSendableEvent event) {
        if (proposal != null)
            return;

        proposal = new TRBProposal(event.getMessage().popObject());

        ConsensusPropose ev =
            new ConsensusPropose(event.getChannel(), Direction.DOWN, this);
        ev.value = proposal;
        ev.go();
    }

    private void failed(Channel channel) {
        if (proposal != null)
            return;
        if ((src == null) || src.isCorrect())
            return;

        proposal = new TRBProposal(true);

        ConsensusPropose ev = new ConsensusPropose(channel, Direction.DOWN, this);
        ev.value = proposal;
        ev.go();
    }

    private void handleConsensusDecide(ConsensusDecide event) {
        if (event.decision instanceof TRBProposal) {
            TRBEvent ev = new TRBEvent(event.getChannel(), Direction.UP, this);
            ev.p = src.getAddress();
            if (((TRBProposal) event.decision).failed)
                ev.m = null;
            else
                ev.m = ((TRBProposal) event.decision).m;
            ev.go();

            init ();
        } else {
            event.go();
  }
    }
}
```

Listing 6.5. TRB-Based View Synchrony implementation

```
package irdp.protocols.tutorialDA.trbViewSync;

public class TRBViewSyncSession extends Session {
    public TRBViewSyncSession(Layer layer) {
        super(layer);
    }

    public void handle(Event event) {
        if (event instanceof ViewEvent)
            handleView((ViewEvent)event);
        else if (event instanceof BlockOkEvent)
            handleBlockOkEvent((BlockOkEvent)event);
        else if (event instanceof TRBEvent)
            handleTRBEvent((TRBEvent)event);
        else if (event instanceof SendableEvent) {
            if (event.getDir() == Direction.DOWN)
                handleCSVSBroadcast((SendableEvent)event);
            else
                handleRCODeliver((SendableEvent)event);
        } else {
        event.go();
        }
    }

    private LinkedList pending_views=new LinkedList();
    private HashSet delivered=new HashSet();
    private View current_view=null;
    private View next_view=null;
    private boolean flushing=false;
    private boolean blocked=true;
    /**
     * trb_done counts the number of processes for wich TRB was used.
     * This is possible because the TRB is used one at a time.
     */
    private int trb_done;

    private int msg_id=0;

    private void handleView(ViewEvent event) {
        if (event.view.id == 0) {
            current_view=event.view;
            blocked=false;

            event.go();

            ReleaseEvent ev=new ReleaseEvent(event.getChannel(), Direction.UP, this);
            ev.go();
        } else {
            pending_views.addLast(event.view);
        }

        moreViews(event.getChannel());
    }

    private void handleCSVSBroadcast(SendableEvent event) {
        if (blocked)
            return;
        // Assert we have a view
        if (current_view == null)
            return;

        // Chooses a unique identifier for the message
        ++msg_id;

        Msg m = new Msg(current_view.memb.getSelfProcess(),
                        msg_id,
                        event.getClass().getName(),
                        event.getMessage().toByteArray());
        delivered.add(m);

        SendableEvent ev=(SendableEvent)event.cloneEvent();
        ev.source=current_view.memb.getSelfProcess().getAddress();
        ev.setDir(Direction.UP);
        ev.setSourceSession(this);
        ev.init();
        ev.go();
```

```
        event.getMessage().pushInt(msg_id);
        event.getMessage().pushInt(current_view.id);
        event.go();
    }

    private void handleRCODeliver(SendableEvent event) {
        SampleProcess src=current_view.memb.getProcess((SocketAddress)event.source);
        Message m=event.getMessage();
        int vid=m.popInt();
        // Message identifier
        int mid=m.popInt();

        if (current_view.id != vid)
            return;

        Msg msg = new Msg(src, mid, event.getClass().getName(), m.toByteArray());
        if (delivered.contains(msg))
            return;
        delivered.add(msg);

        event.go();
    }

    private void moreViews(Channel channel) {
        if (flushing)
            return;
        if (pending_views.size() == 0)
            return;

        next_view=(View)pending_views.removeFirst();
        flushing=true;

        BlockEvent ev=new BlockEvent(channel, Direction.UP, this);
        ev.go();
    }

    private void handleBlockOkEvent(BlockOkEvent event) {
        blocked=true;
        trb_done=0;

        SampleProcess p_i = current_view.memb.getProcess(trb_done);
        TRBEvent ev = new TRBEvent(event.getChannel(), Direction.DOWN, this);
        ev.p = p_i.getAddress();
        if (p_i.isSelf()) {
            ev.m=delivered;
        } else {
            // pushes an empty set
            ev.m=new HashSet();
        }
        ev.go();
    }

    private void handleTRBEvent(TRBEvent event) {
        HashSet del=(HashSet)event.m;
        trb_done++;

        if (del != null) {
            Iterator iter = del.iterator();
            while (iter.hasNext()) {
                Msg m = (Msg) iter.next();

                if (!delivered.contains(m)) {
                    delivered.add(m);

                    SendableEvent ev = (SendableEvent) Class.forName(m.eventName).newInstance();
                    ev.getMessage().setByteArray(m.data, 0, m.data.length);
                    ev.setChannel(event.getChannel());
                    ev.setDir(Direction.UP);
                    ev.setSourceSession(this);
                    ev. init ();
                    ev.go();
                }
            }
        }

        if (trb_done < current_view.memb.getSize()) {
            SampleProcess p_i = current_view.memb.getProcess(trb_done);
            TRBEvent ev = new TRBEvent(event.getChannel(), Direction.DOWN, this);
            ev.p = p_i.getAddress();
            ev.m = new Message();
            if ( p_i . isSelf ()) {
                ev.m=delivered;
            } else {
                // proposes an empty set
                ev.m=new HashSet();
            }
            ev.go();
        } else {
            ready(event.getChannel());
        }
    }

    private void ready(Channel channel) {
        if (!blocked)
            return;
        // because TRB was used one at a time,
        // and this function is only invoqued when all are
        // done, its not required to check the "trb_done" variable.

        current_view=next_view;
        flushing=false;
        blocked=false;

        ViewEvent ev1=new ViewEvent(channel, Direction.UP, this);
        ev1.view=current_view;
        ev1.go();

        ReleaseEvent ev2=new ReleaseEvent(channel, Direction.UP, this);
        ev2.go();

        moreViews(channel);
    }
}
```

References

Cachin, C., R. Guerraoui, and L. Rodrigues (2011). *Introduction to Reliable and Secure Distributed Programming* (2nd ed.). Springer.

Guerraoui, R. and L. Rodrigues (2006). *Introduction to Reliable Distributed Programming*. Secaucus, NJ, USA: Springer-Verlag New York, Inc.

Miranda, H., A. Pinto, and L. Rodrigues (2001, April). Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, Phoenix, Arizona, pp. 707–710. IEEE.