Implementing Reliable Broadcast Protocols in *Appia*

A Brief Tutorial V1.1

Nuno Carvalho
Universidade de Lisboa
nunomrc@di.fc.ul.pt

Luís Rodrigues
Universidade de Lisboa
ler@di.fc.ul.pt

3rd November 2003

Abstract

This tutorial is a companion of the book *Abstractions for Distributed Programming*, by Rachid Guerraoui and Luís Rodrigues [1]. It illustrates how some of the broadcast abstractions presented in the book can be implemented in practice. The implementation uses the *Appia* system, a framework for protocol composition and execution [3]. All the code presented in this tutorial can be downloaded from the *Appia* site: `http://appia.di.fc.ul.pt`.

Contents

1 Introduction

Many distributed applications are no longer structured using a basic client-server model, where interactions are only among two entities. On the contrary, multi-participant applications, where data items need to be disseminated to a group of nodes are becoming more and more common.

The broadcast primitives addressed in this tutorial aim at simplifying the task of building such multi-participant applications. By using a broadcast abstraction, a node may disseminate a value among a set of processes with a single protocol invocation. The details of how the message is actually disseminated, and how reliability is ensured in the presence of faults, are encapsulated by the broadcast abstraction.

This tutorial is a companion of the book *Abstractions for Distributed Programming*, by Rachid Guerraoui and Luís Rodrigues [1]. It illustrates how some of the broadcast abstractions presented in the book can be implemented in practice.

The implementation uses the *Appia* system, a framework for protocol composition and execution [3]. For self containment, the tutorial makes an extremely brief introduction to *Appia*. However, the reader may find useful to read additional information about *Appia* in [2].

All the code presented in this tutorial can be downloaded from the *Appia* site: `http://appia.di.fc.ul.pt`. The complete *Appia* distribution is available from the same site. Both the *Appia* kernel and the examples presented here are written in the `Java` programming language. Therefore, all you need to run and experiment the examples is a Java Virtual Machine (JDK 1.4.1 and higher).

1.1 Abstractions Covered by the Tutorial

This document describes the implementation of the following reliable broadcast protocols from [1]:

- Lazy Reliable Broadcast;
- Uniform Reliable Broadcast;
- Indulgent Uniform Reliable Broadcast;
- Probabilistic Broadcast.

1.2 Underlying Abstractions

The protocols described here are based on some other underlying abstractions, namely: Perfect Point to Point Links, Unreliable Point to Point Links, Best Effort Broadcast, and Perfect Failure Detector. Although we will not discuss in detail the implementation of all these abstractions, for

self-containment we will also offer a brief introduction to the *Appia* protocols that implement them. In particular, the protocols presented in this tutorial make use of the following *Appia* protocols: TcpComplete, UdpSimple, and PFD (for Perfect Failure Detector).

1.3 Hands-On

The reader is encouraged to execute and test the implementations described in this tutorial (which can be downloaded from the *Appia* site). To simplify this task, the software distribution also includes a test application, that allows the user to launch several processes and, from any of these processes, broadcast messages to the whole group.

1.4 Structure of the Document

The remaining of the document is structured as follows. Section 2 explains how to run the test application. Section 3 offers a quick introduction to *Appia*. Section 4 describes the protocol used to implement the Link abstraction. The Best Effort Broadcast protocol is described in Section 5 and the implementation of a Perfect Failure Detector is given in Section 6. The Lazy Reliable Broadcast protocol is then described in Section 7, the Uniform Reliable Broadcast in Section 8, the Indulgent Uniform Reliable Broadcast in Section 9, and the Probabilistic Reliable Broadcast in Section 10. Finally, Section 11 concludes the tutorial.

2 Running the Test Application

All the implementations covered by this tutorial may be experimented using the same 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:

```
java applications/SampleAppl -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 in 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).

The `prot` parameter specifies which broadcast abstraction is used by the application. One of the following values can be selected:

beb to use Best Effort Broadcast protocol;

rb to use Lazy Reliable Broadcast protocol;

urb to use Uniform Reliable Broadcast protocol;

iurb to use Indulgent Uniform Reliable Broadcast protocol; and

pb <fanout> <maxrounds> to use the Probabilistic Broadcast protocol with a specified fanout and maximum number of message rounds.

After all processes are launched, a message can be sent from one process to the other processes just by typing a `bcast <string>` in the command line and pressing the `Enter` key. The application also accepts another command, the `startpfd` command, which is introduced later in the text.

3 A Brief Overview of *Appia*

Appia is a protocol composition and execution framework that has been implemented in the Java programming language. *Appia*'s composition model is based on three fundamental abstractions: *layers*, *sessions*, and *events*:

- *Layers* are singleton entities, responsible for declaring the protocol behavior. A stack of layers defines what is called in *Appia* a *QoS* (Quality of Service). The QoS specifies which protocols must act on the messages and the order they must be traversed.

- *Sessions* are protocol instances that maintain the state required to execute the protocol. *Channels* are stacks of sessions. Each channel is defined on behalf of a QoS. The set of sessions composing a channel must respect the set of layers used in the corresponding QoS. Typically, the top of the stack is an application layer and the bottom of the stack is a layer that interfaces the network.

- *Events* are data structures used by sessions to exchange information. The framework supports an open event model, that is, the set of events supported is not defined *a priori*, and can be extended by the programmer. The framework ensures that events exchanged between two sessions in the same channel are delivered respecting FIFO order. Events may flow in the stack in both directions: upwards (from the network to the application) or downwards.

To exchange information between two different processes, *Appia* uses a specific event: the SendableEvent. This event has two additional fields, the source and dest that are used to indicate the source and destination endpoints. A SendableEvent also has a `Message` fielda, which contains the information that will be sent to the communication link.

To build an *Appia* protocol, two classes must be created: a Layer and a Session. The Layer contains static information about the events that the protocol will *accept*, *require* and *provide*. The Session has the state of the protocol. An event is delivered to a protocol (or Session) in the handle(Event e) method. When a protocol wants to send an event to the next protocol, it must call the go() method of the event.

Appia provides also a set of common utility services to simplify the task of protocol development, such as the management of data buffers for messages (with methods to add, extract and inspect headers), management of timers, automatic generation of events to initialize the channels, etc.

More information about *Appia* can be found in the *Appia* home page (http://appia.di.fc.ul.pt) and in the following reports [3, 2].

4 Links

This section briefly describe the *Appia* protocols that implement the Perfect Point to Point Links and Unreliable Point to Point Links abstractions. These protocols glue the remaining *Appia* protocols with the operating system services that give access to the network (typically, TCP or UDP sockets).

4.1 Perfect Point to Point Links

The Perfect Point to Point Links abstraction is implemented in *Appia* by the TcpComplete protocol. As its name implies, this implementation is based on the TCP protocol, more precisely, it uses TCP sockets as communication channels. When a TcpComplete 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 TcpComplete 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 acknowledgements 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.

4.2 Unreliable Point to Point Links

The Unreliable Point to Point Links abstraction is implemented in *Appia* by the UdpSimple protocol. The UdpSimple protocol uses UDP sockets as unreliable communication channels. The way SendableEvents are handled by an UdpSimple session is similar to the TcpComplete session described above. The only difference is that UDP sockets are used instead.

5 Best-Effort Broadcast

A protocol implementing the Best-Effort Broadcast abstraction is used to simplify the task of implementing Reliable Broadcast. We now describe a simple implementation of Best-Effort Broadcast in *Appia*. The protocol uses multiple Perfect Point-to-Point Links to disseminate a message to a set of processes.

5.1 Hands-On

To run the test application and experiment the dissemination of messages among a set of processes using Best-Effort Broadcast, type the following command:

```
java applications/SampleAppl -f <cf> -n <rank> -qos beb
```

This command will start a local process with the specified rank (the address and port of the process are specified in the file `cf`). Issue this command in a different shell (and optionally a different machine, depending on the content of file `cf`) for each member of the multicast group.

5.2 Introduction to the Algorithm

The algorithm to implement Best-Effort broadcast is very simple. Figure 1 illustrates the algorithm. When the application sends a message, a copy of the message is sent to every member of the group, including to the sender itself. To support the transmission of messages, the Best-Effort Broadcast uses Perfect Point-to-Point Links. On the recipient side, when the node receives a message, forwards it to the application (or to the above protocol); this is defined as *delivering* the message.

Implements:
BestEffortBroadcast (beb).

Uses:
PerfectPointToPointLinks (pp2p).

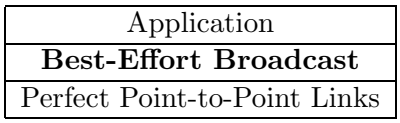
upon event $\langle$ *bebBroadcast, m* $\rangle$ **do**
forall $p_i \in \Pi$ **do** // Π is the set of all system processes
trigger $\langle$ *pp2pSend,* p_i, m $\rangle$;

upon event $\langle$ *pp2pDeliver,* p_i, m $\rangle$ **do**
trigger $\langle$ *bebDeliver,* p_i, m $\rangle$;

Figure 1: Best-Effort Broadcast algorithm.

5.3 Description of the Protocol

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



The implementation of this algorithm closely follows the algorithm of Figure 1. As shown in Listing 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, that is 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 downwards) or the reception of events from the layer below (if the event flows upwards). 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 real code distributed with the tutorial).

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 1: Best-effort broadcast implementation.

```
public class BEBSession extends Session {

    private ProcessSet processes;

    public BEBSession(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().getInetWithPort();
                sendingEvent.dest = processArray[i].getInetWithPort();
                // set the event fields
                sendingEvent.setSource(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();
        }
}
```

6 Perfect Failure Detector

This abstraction permits the notification of crashed processes. When a process crash, the remaining processes will be notified and will do some processing to ensure the reliability properties.

6.1 Description of the protocol

In this case, the Perfect Failure Detector (PFD) is only used with Perfect Point to Point Links (PP2PL), which are builded using TCP channels. When a TCP socket is closed, the protocol that implements PP2PL sends an event to the *Appia* channel. This event is accepted by the PFD protocol, which sends a Crash event to notify other layers. The implementation of this notification is shown in Listing 2. The protocols that need a PFD declare that will accept the Crash event and will process it, as shown in the implementation of the reliable broadcast protocols, which are described in the next Section.

To notify other layers of a closed socket, the PP2PL protocol must first create the corresponding TCP sockets. The way the PP2PL is implemented, these sockets are open 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 PFD session send a message to all other processes when it is started.

Note that the PFD 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. In order to start the perfect failure detector from the test application, must issue the `pfdstart` request on the command line.

Listing 2: Perfect failure detector implementation.

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

  public PerfectFailureDetectorSession(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((InetWithPort) event.who);
      if (p.isCorrect()) {
        p.setCorrect(false);
        Crash crash =
          new Crash(channel,Direction.UP,this,p.getProcessNumber());
        crash.go();
      }
    }
  }
}
```

7 Lazy Reliable Broadcast

The Best-Effort reliable protocol presented in Section 5 does not ensure reliability when the sender crashes. The Lazy Reliable Broadcast solves this problem by having other processes to forward messages on behalf of a crashed sender.

7.1 Hands-On

To run the test application and experiment the dissemination of messages among a set of processes using Lazy Reliable Broadcast, type the following command:

```
java applications/SampleAppl -f <cf> -n <rank> -qos rb
```

This command will start a local process with the specified rank (the address and port of the process are specified in the file `cf`). Issue this command in a different shell (and optionally a different machine, depending on the content of file `cf`) for each member of the multicast group. Do not forget to initiate the PFD at every processes by issuing the `pfdstart` request on the command line.

7.2 Introduction to the Algorithm

The Lazy Reliable Broadcast algorithm is depicted in Figure 2. In this algorithm, when the broadcast of a message is requested, the message is sent using the Best Effort Broadcast (BEB). If the sender does not crash, BEB will ensure the message delivery to all members of the group (including itself). When a message is received for the first time, it is delivered and a copy stored in a log. If duplicates are received due to retransmissions, these are simply discarded. When a process detects that some other process f has failed, it retransmits all messages received from f.

7.3 Description of the protocol

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

Application
Reliable Broadcast
Perfect Failure Detector
Best Effort Broadcast
Perfect Point to Point Links

The implementation of this algorithm, shown in Listing 3, closely follows the algorithm of Figure 2. The protocol accepts four events, namely the ProcessInitEvent, used to initialize the set of processes that participate in

```
Implements:
    ReliableBroadcast (rb).

Uses:
    BestEffortBroadcast (beb).
    PerfectFailureDetector (P).

upon event ⟨ Init ⟩ do
    delivered := ∅;
    correct := Π;
    ∀_{p_i ∈ Π} : from[p_i] := ∅;

upon event ⟨ rbBroadcast, m ⟩ do
    trigger ⟨ bebBroadcast, [ Data, self, m ] ⟩;

upon event ⟨ bebDeliver, p_i, [ Data, s_m, m ] ⟩ do
    if m ∉ delivered then
        delivered := delivered ∪ {m}
        trigger ⟨ rbDeliver, s_m, m ⟩;
        from[p_i] := from[p_i] ∪ { [ s_m, m ] }
        if p_i ∉ correct then
                trigger ⟨ bebBroadcast, [ Data, s_m, m ] ⟩;

upon event ⟨ crash, p_i ⟩ do
    correct := correct \ {p_i}
    forall [s_m, m] ∈ from[p_i]: do
        trigger ⟨ bebBroadcast, [ Data, s_m, m ] ⟩;
```

Figure 2: Lazy Reliable Broadcast algorithm.

the broadcast (this event is triggered by the application after reading the configuration file), the ChannelInit event, that is 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 downwards) or the reception of events from the layer below (if the event flows upwards). 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 real code distributed with the tutorial).

In order to detect duplicates, each message needs to be uniquely identified. In this implementation, the protocol use 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, to broadcast a message consists only in pushing the message identifier and forward the request to the Best-Effort layer. To receive

the message consists in popping the message identifier, check for duplicates, and to log and deliver 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 be crashed, the message is immediately retransmitted.

Listing 3: Lazy reliable broadcast implementation.

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

    public RBSession(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++;
        ((ExtendedMessage)event.getMessage()).pushObject(msgID);
        bebBroadcast(event);
    }

    private void bebDeliver(SendableEvent event) {
        MessageID msgID = (MessageID) ((ExtendedMessage)event.getMessage()).peekObject();
        if ( ! delivered.contains(msgID) ){
            delivered.add(msgID);
            SendableEvent cloned = (SendableEvent) event.cloneEvent();
            ((ExtendedMessage)event.getMessage()).popObject();
            event.go();
            SampleProcess pi = processes.getProcess((InetWithPort) 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.setSource(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);
        }
    }
}
```

7.4 Exercises

Exercise 7.1 *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.*

Exercise 7.2 *Perform a similar optimization with the* from *variable.*

8 Uniform Reliable Broadcast

The protocol described in this section offers stronger properties than the previous protocol. It ensures uniform reliability, that is, no process delivers a message without being sure that all correct processes will also be able to deliver that message. For this purpose, delivery of a message is delayed until an acknowledgement is received from every correct process.

8.1 Hands-On

To run the test application and experiment the dissemination of messages among a set of processes using Uniform Reliable Broadcast, type the following command:

```
java applications/SampleAppl -f <cf> -n <rank> -qos urb
```

This command will start a local process with the specified rank (the address and port of the process are specified in the file `cf`). Issue this command in a different shell (and optionally a different machine, depending on the content of file `cf`) for each member of the multicast group. Do not forget to initiate the PFD at every processes by issuing the `pfdstart` request on the command line.

8.2 Introduction to the Algorithm

Figure 3 shows how the algorithm works. When the protocol sends a message, adds it to a *list of forwarded messages*. When the protocol receives a message, resends it, if it does not belong to the *forward* list. So, all processes forward the received messages using the Best Effort Broadcast protocol. By retransmitting a message, a process is implicitly acknowledging that it has received and stored the message. When the message has been forwarded by all correct processes, it can be safely delivered to the application.

8.3 Description of the protocol

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

Application
Uniform Reliable Broadcast
Perfect Failure Detector
Best Effort Broadcast
Perfect Point to Point Links

The implementation of this protocol is shown in Listing 4. Note that the code in these listing has been simplified. In particular, all exception

```
Implements:
    UniformReliableBroadcast (urb).

Uses:
    BestEffortBroadcast (beb).
    PerfectFailureDetector (P).

function canDeliver(m) returns boolean is
    return (correct ⊂ ack_m) ∧ (m ∉ delivered);

upon event ⟨ Init ⟩ do
    delivered := forward := ∅;
    correct := Π;
    ack_m := ∅, ∀_m;

upon event ⟨ urbBroadcast, m ⟩ do
    forward := forward ∪ {m}
    trigger ⟨ bebBroadcast, [ Data, self, m ] ⟩;

upon event ⟨ bebDeliver, p_i, [ Data, s_m, m ] ⟩ do
    ack_m := ack_m ∪ {p_i}
    if m ∉ forward do
        forward := forward ∪ {m};
        trigger ⟨ bebBroadcast, [ Data, s_m, m ] ⟩;

upon event ⟨ crash, p_i ⟩ do
    correct := correct \{p_i};

upon (canDeliver(m)) do
    delivered := delivered ∪ {m};
    trigger ⟨ urbDeliver, s_m, m ⟩;
```

Figure 3: All ack uniform reliable broadcast algorithm.

handling code was deleted from the listings for clarity (but is included in the real code distributed with the tutorial).

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 4: All ack uniform reliable broadcast implementation.

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


  public URBSession(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) );
  }

  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);
    ((ExtendedMessage) event.getMessage()).pushObject(msgID);
    event.go ();
  }

  private void bebDeliver(SendableEvent event) {
    SendableEvent clone = (SendableEvent) event.cloneEvent();
    MessageID msgID = (MessageID) ((ExtendedMessage) 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.setSource(this);
    event.init();
    event.go();
  }

  private void urbDeliver(SendableEvent event, int sender) {
    event.setDir(Direction.UP);
    event.setSource(this);
    event.source = processes.getProcess(sender).getInetWithPort();
    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((InetWithPort)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;
  }
}
```

8.4 Exercises

Exercise 8.1 *Modify the implementation to keep track just of the last message sent from each process, in the* received *and* delivered *variables.*

Exercise 8.2 *Change the protocol to exchange acknowledgements 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 protocol of Section 7).*

9 Indulgent Uniform Reliable Broadcast

This protocol provides uniform reliability without requiring the availability of a perfect failure detector. Instead, it only allows a minority of processes to crash. In such case, a message can be delivered as soon as an acknowledgment is received from a majority of processes.

9.1 Hands-On

To run the test application and experiment the dissemination of messages among a set of processes using Indulgent Uniform Reliable Broadcast, type the following command:

```
java applications/SampleAppl -f <cf> -n <rank> -qos iurb
```

This command will start a local process with the specified rank (the address and port of the process are specified in the file `cf`). Issue this command in a different shell (and optionally a different machine, depending on the content of file `cf`) for each member of the multicast group.

9.2 Introduction to the Algorithm

The algorithm is very similar to the algorithm of Figure 3. The only difference is in the condition that allows messages to be delivered, as depicted in Figure 4. This change also allows the protocol to execute without the availability of a perfect failure detector as long as only a minority of processes are allowed to crash.

9.3 Description of the protocol

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

Implements:
UniformReliableBroadcast (urb).

Uses:
BestEffortBroadcast (beb).

function canDeliver(m) **returns** boolean **is**
return ($|\text{ack}_m| > \text{N}/2$) $\wedge$ ($m \notin$ delivered);
// Except for the function above, same as Algorithm 3.

Figure 4: Majority ack uniform reliable broadcast algorithm.

Application
Indulgent Uniform Reliable Broadcast
Best Effort Broadcast
Perfect Point to Point Links

The protocol works in the same way as the protocol presented in Section 8, but without being aware of crashed processes. Besides that, the only difference from the previous implementation is the `canDeliver()` method, which can be shown in Listing 5.

Listing 5: Indulgent Uniform reliable broadcast implementation.

```
public class IURBSession 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 the previous protocol
}
```

9.4 Exercises

Exercise 9.1 *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).*

10 Probabilistic Reliable Broadcast

Probabilistic reliable broadcast uses an algorithm quite different from the ones presented in previous sections. Instead of ensuring reliability in a deterministic manner, it offers reliability only with high probability. The most important advantage of this protocol is that it scales much better, i.e., using this approach it is possible to broadcast messages to thousands of nodes.

10.1 Hands-On

To run the test application and experiment the dissemination of messages among a set of processes using Probabilistic Broadcast, type the following command:

```
java applications/SampleAppl -f <cf> -n <rank> -qos pb <fanout>
<maxrounds>
```

This command will start a local process with the specified rank (the address and port of the process are specified in the file `cf`). Issue this command in a different shell (and optionally a different machine, depending on the content of file `cf`) for each member of the multicast group.

10.2 Introduction to the Algorithm

The probabilistic algorithm is illustrated in Figure 5. The message dissemination is initiated by gossiping it to $fanout$ members of the group, selected at random. When the message is received, it is gossiped again, to another $fanout$ random members. This procedure is executed $maxrounds$ times. If $fanout$ and $maxrounds$ are chosen carefully, all processes will receive and deliver the message with high probability.

10.3 Description of the protocol

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
Unreliable Point to Point Links

The protocol has two configurable parameters:

fanout is the number of processes for which the message will be gossiped;

Implements:
ProbabilisticBroadcast (pb).

Uses:
unreliablePointToPointLinks (up2p).

upon event ⟨ *Init* ⟩ **do**
delivered := ∅;

function pick-targets (fanout) **returns** set of processes **do**
targets := ∅;
while | targets | < fanout **do**
candidate := random (Π);
if candidate ∉ targets ∧ candidate ≠ self **then**
targets := targets ∪ { candidate };
return targets;

procedure gossip (msg) **do**
forall t ∈ pick-targets (fanout) **do**
trigger ⟨ *up2pSend, t, msg* ⟩;

upon event ⟨ *pbBroadcast, m* ⟩ **do**
gossip ([Gossip, s_m, *m, maxrounds*−1]);

upon event ⟨ *up2pDeliver,* p_i, [*Gossip,* s_m, *m, r*] ⟩ **do**
if $m \notin$ delivered **then**
delivered := delivered ∪ {m}
trigger ⟨ *pbDeliver,* s_m, m ⟩;
if $r > 0$ **then** gossip ([Gossip, s_m, m, $r-1$]);

Figure 5: Probabilistic broadcast algorithm.

maxrounds is the maximum number of rounds that the message will be retransmitted.

The implementation of this protocol is shown on Listing 6. The gossip() method invokes the pickTargets() method to choose the processes 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 6: Probabilistic broadcast implementation.

```
public class PBSession extends Session {

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

  public PBSession(Layer layer) {
    super(layer);
```

```
  PBLayer pbLayer = (PBLayer) 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((InetWithPort)event.source);
  int round = ((ExtendedMessage) event.getMessage()).popInt();
  MessageID msgID = (MessageID) ((ExtendedMessage) 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();
      ((ExtendedMessage) clone.getMessage()).pushObject(msgID);
      ((ExtendedMessage) 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.setSource(this);
    event.dest = processes.getProcess(dest).getInetWithPort();
    event.init();
    event.go();
  }

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

10.4 Exercises

Exercise 10.1 *The up2pDeliver() method perform two different functions:* i) *delivers the message to the application (if it was not delivered yet) and* ii) *gossips the message to other processes. Change the code such that a node gossip just when it receives a message for the first time. Discuss the impact of the changes.*

Exercise 10.2 *Change the code to limit* i) *the number of messages each node can store;* ii) *the maximum throughput (messages per unit of time) of each node.*

11 Conclusions and Further Work

This tutorial offers an introduction to the *Appia* implementation of several of the algorithms presented in [1]. The book also covers the problem of reliable broadcast in a setting where processes can crash and recover. We invite the reader to build an *Appia* stack to implement such abstractions.

The reader is also invited to compare the protocols provided in this tutorial with the protocols used by the group communication protocols included in the *Appia* distribution.

References

[1] Rachid Guerraoui and Luis Rodrigues. *Abstractions for Distributed Programming*. (in preparation).

[2] H. Miranda, A. Pinto, and L. Rodrigues. *The Appia tutorial*.

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