19 Mobile Processes: Ubiquitous Wireless Access

The previous chapter showed how it is possible to create anonymous network channels over which agents could be transferred. In this chapter we take that concept one stage further and provide a mechanism whereby a process can be communicated from one node to another within the network. The only requirement is that the receiving node has to run a simple process that loads the mobile process. This is further extended to load a process from a server over a wireless network to a mobile device. The mobile device becomes a member of the server's network for the duration of the interaction. The mobile device scans for accessible wireless networks and then is able to download a process from that network by which it can interact with the service provided.

This technology could be used in a retail environment to let stores make offers to customers, as they walk into the store, based upon their previous shopping patterns. In addition, the store could make offers on surplus items to customers they know might be susceptible to the offer. The only requirement is that the customer has a mobile device into which the process loading process has been installed. The customer would also need to store some means of identifying themselves to the store's systems but with loyalty or reward cards this is not a problem.

The technology could be used in a hospital environment to allow access to electronic patient records by registered users, using their own mobile device. The great advantage being that the location of a person can be determined by the wireless access points that are available and this could result in the most appropriate process being downloaded into the mobile device depending upon the user and their role. Obviously, some form of authentication process would be required to ensure authorised access but the advantage of this style of interaction is that no sensitive data is held in the mobile device.

Finally, it could be used in museums to provide additional resources to visitors about the items on display. In this case, rather than using wi-fi we could use Bluetooth to give more locality of information. The downloaded process could provide additional information in the form of an audio stream giving an aural description of the exhibit, possibly supported by an image that shows the particular part of the object being described. The audio stream could be in any natural language. The particular advantage for the museum is that visitors can use their own mobile devices, provided they have the process to download other processes.

The mobile process capability is provided by a `mobile` package within the `jcsp.net` capability. It deals with the dynamic loading of classes over the network in an efficient manner that is totally transparent to the programmer and of the underlying network technology. Processes are loaded just like any other object as a `Serializable` data object. The processes will include some of the network channels in their definition that will allow the loaded process to communicate from the mobile device to the server. However, channels that enable communication from the server to the loaded mobile process will need to be created dynamically.

This application is different from others because we are running the systems of processors with different resource capability. In particular, the requirement to run Groovy on a mobile device is problematic, given its size and the underlying functionality it requires in terms of reflector requirements. To this end, all the processes that execute on the mobile device are written in pure Java. They do however interact with server processes written in Groovy. Thus this chapter demonstrates that Groovy and Java components can be combined into a single application environment. The dominant requirement being that all devices run using a Java Virtual Machine, which is the case with most mobile devices.

19.1 The Travellers' Meeting System

The meeting system is a service provided by a travel authority such as a railway station or airport that enables people travelling together to find out where other members of a group are located especially in the event of a travel delay. A member of the group registers the name by which the group recognises itself together with the location of where they are to congregate. Other members may try to create another location but will be informed that the group has already been registered and given that location. Other members will just try to find the meeting location and will be informed of its location. People who try to find a meeting that is not yet registered are informed of this case.

The primary requirement is for an initial channel by which a mobile device can register itself with the server network. This is similar to the `Request` channel used in the pervious chapter to make requests to the printer spooling service. All the `PrintUser` processes knew that the access channel was called `Request`. The situation becomes more complex as we move to a more general environment. If the process loaded into the mobile device is to function with all such publicly available service providers then they are all going to have to use the same name for their access channel. In the case of the hospital environment briefly described above this name would not be made publicly available.

Once the initial mobile process has been loaded this can then be used to determine the required service and then further processes can be loaded using private access channels. In the meeting example the initial mobile process will be loaded using the publicly available access channel. The initial mobile process will then determine, by means of a user interaction, whether the user wants to create a new meeting location or find an existing meeting and then load the required process using private channels.

19.2 Ubiquitous Access Client

The Ubiquitous Access Client (UAC) is the process that executes within the mobile device scanning for wireless access points (WAP) connected to networks that are offering services based upon the ubiquitously available access channel. Such a network is defined by the IP address of the node upon which the CNS executes. For the purposes of this description the coding to scan for such a WAP is ignored and we shall just type in the IP address of the node upon which the CNS is running. As indicated above the process is written in Java rather than Groovy. The coding of the process is shown in Listing 19-1, from which all the unnecessary coding has been removed. The process is run as a `main`, rather than using the `CSProcess` interface with a `run` method {1, 3}. The channel `processReceive` {2} will be used to input a process from the server, once the connection to the network has been made. Note the relative complexity of defining class properties, when compared with that required by Groovy, which determines types at run-time.

```
01    public class UASSSClient {

02      private static NetAltingChannelInput processReceive;

03      public static void main(String[] args) {

04        String CNS_IP = Ask.string("Enter IP address of CNS: ");
05        try {
06          Mobile.init(Node.getInstance().init(new TCPIPNodeFactory(CNS_IP)));
07          String processService = "A";
08          NetChannelLocation serverLoc = CNS.resolve(processService);
09          NetChannelOutput toServer = NetChannelEnd.createOne2Net(serverLoc);
10          processReceive =  Mobile.createNet2One();
11          toServer.write(processReceive.getChannelLocation());
12          MobileProcess theProcess = (MobileProcess)processReceive.read();
13          new ProcessManager(theProcess).run();
14        }
15        catch (NodeInitFailedException e)  {
16          System.out.println("Failed to connect to Server");
17          System.exit(-1);
18        }
19      }
20    }
```

Listing 19-1 The Ubiquitous Access Client

The address of the `CNS` server is obtained, simply by reading it in from the console {4}. The package `Mobile` builds upon the `CNS` and enables the creation of network nodes {6} that are able to manipulate mobile processes. It is possible for the creation of a node to fail and thus the coding is enclosed in a `try – catch` block {5, 15-17}. For the purposes of this system, the universal access channel to the server is called "`A`" {7}. The mobile device, running this process, can presume that this channel already exists and thus can `resolve` that channel's location as `serverLoc` {8}. That location can then be used to create a net output channel called `toServer`, which connects the mobile device to the server {9}. The channel `processReceive` {10} is created as a `Net2One` input channel, within the network that contains the server and `CNS` with which the mobile device is interacting.

The location of `processReceive` is then written to the server using the previously determined `toServer` channel {11}. It should be noted that any number of mobile devices could be attempting this connection at the same time. Recall that net input channels are implements as `Any2One` and so this does not cause any problem provided the mobile device only sends one communication over the channel. The server uses the net channel location of `processReceive` to output a `MobileProcess` which can then be read by the mobile device into `theProcess` {12}. Finally, `theProcess` is executed within a `ProcessManager` {13}, which allows a process to be spawned concurrently with the currently executing process. At this point an initial mobile process has been downloaded from the server into the mobile device and the interaction with the user can commence.

Examination of Listing 19-1 shows that the code that has to be executed in a mobile device that permits the downloading of processes from service providers is very simple. The process that scans for available wireless networks that are providing the ubiquitous access capability has not been included but can be run in the same JVM as a parallel process [chalmers]. A reduced form of the JCSP can be incorporated (JCSPme [chalmers]) into the mobile device, thereby enabling a parallel processing capability in a mobile device. The `mobile` package [chalmers] also contains a means of downloading classes across the network in a totally transparent manner, thus any class not loaded as part of the initial process download, say for example a class used within a process, can be dynamically loaded when required. This downloading is undertaken on a class basis rather than requiring the downloading of the whole assemblage of classes that might possibly be required. Thus JCSPme might not contain, for example, the `jcsp.awt` classes but these can be downloaded as they are required over the network. Moreover only the required classes are downloaded, rather than the entire `jar` file containing `jcsp.awt`. JCSPme has a total footprint of about 90Kbytes.

The great advantage of this approach to mobile computing is that the processes executed within the mobile device are created and maintained by the service provider. The resources used within the mobile device can be recovered automatically, once the interaction between mobile device and service provider has terminated. The interaction is fully under the control of the service provider and is not reliant upon on third party software supplier such as a web browser. The data transferred between the mobile device and the service provider's system tends to be much smaller and more focussed than say the transfer of a web page. Furthermore the interaction with the user can be better organised and managed because the precise nature of the data transferred is known. Thus the remainder of this chapter focuses on the processes that are downloaded into the mobile device and the server processes required to support the service.

19.3 The Initial Mobile Process

Like all the processes that are downloaded into the mobile device, the initial mobile process comprises three processes. First a process that causes the other two processes to run in parallel. The second process provides a graphical user interface to the third process which contains the required functional capability. In the case of the initial mobile process all three processes will be described in detail. Other downloaded processes will only have their capability process discussed.

19.3.1 The Access Client Process

Listing 19-2 shows the `AccessClientProcess`, which causes the running, in parallel, of the capability and user interface processes.

```
21    public class AccessClientProcess extends MobileProcess {

22      public void run () {

23        final Any2OneChannel events = Channel.createAny2One();

24        final CSProcess[] network = {
25          new AccessClientCapability ( events.in() ) ,
26          new AccessClientUserInterface ( events.out() )
27        };

28        new Parallel (network).run();
29      }
30    }
```

Listing 19-2 The Initial Mobile Process – AccessClientProcess

The `AccessClientProcess` extends `MobileProcess` {21} and implements a single method `run` {22}. The abstract class `MobilePocess` implements the interfaces `CSProcess` and `Serializable`. The underlying JCSP implementation uses arrays of processes rather then the Groovy list based formulation. Hence, an array of `CSProcesses` is created as `network` {24-27}. This is then executed using the `Parallel` class {28}. The channel `events` provides the connection between the user interface process `AccessClientUserInterface` {26} and the `AccesClientCapability` process {25}. The user interface has two buttons of which only one can be clicked at any one time and so an `Any2One` channel is the most appropriate choice.

19.3.2 The Access Client User Interface Process

Listing 19-3 shows the coding of the user interface process used by the initial mobile process.

```
31     public class AccessClientUserInterface implements CSProcess {

32       private  ChannelOutput buttonEvent;

33       public AccessClientUserInterface(ChannelOutput buttonEvent) {
34           this.buttonEvent = buttonEvent;
35       }

36       public void run() {
37         final ActiveClosingFrame root = new ActiveClosingFrame (
38                                               "Jon's Meeting Service");
39         final Frame mainFrame = root.getActiveFrame();
40         mainFrame.setSize (320, 480);
41         mainFrame.setLayout ( new BorderLayout() );

42         final ActiveButton newButton = new ActiveButton (
43                                    null, buttonEvent, "Create New Meeting" );
44         final ActiveButton findButton = new ActiveButton (
45                                    null, buttonEvent, "Find Existing Meeting" );

46         final Container buttonContainer = new Container();
47         buttonContainer.setSize(320,480);
48         buttonContainer.setLayout ( new GridLayout ( 2,1 ) );
49         buttonContainer.add ( newButton );
50         buttonContainer.add ( findButton );
51         mainFrame.add ( buttonContainer, BorderLayout.CENTER );
52         mainFrame.pack();
53         mainFrame.setVisible(true);

54         final CSProcess [] network = {
55                  root,
56                  newButton,
57                  findButton
58         };

59         new Parallel ( network ).run();
60       }
61     }
```

Listing 19-3 The Access Client User Interface Process

The process implements the `CSProcess` interface {31} and declares `buttonEvent` as a `private ChannelOutput` property {32}. The process constructor is shown next {33-35}. The `run` method {36} is similar to the user interfaces defined previously. It comprises two active buttons `newButton` {42-43} and `findButton` {44-45}, which respectively indicate that the user wants to create a new meeting or find an existing meeting. These buttons are placed in `buttonContainer` {46} and added to the `mainframe` of the interface {52}. An array of `CSProcess`, called `network`, is then defined {54-58} and then executed as a `Parallel` {59}.

19.3.3 The Access Client Capability Process

The `AccesClientCapability` process, shown in Listing 19-4 receives inputs from the user interface process described previously on its `private eventChannel` property {63}, which is the only property that has to be initialised in the class constructor {64-66}. The user interface has only two active buttons, `newButton` {42} and `findButton` {44} both of which output to the `Any2One` channel `events` {23} and thus the capability process only has to determine which of the buttons has been activated. The `run` method {67} reads in the `eventType` from the `eventChannel` {68} and then using a simple `if` statement sets the required `serviceName` to either `N` or `F` {69}. These are the names by which the service request channels for a new-meeting or find-meeting process are accessed. These channels have already been registered with the CNS and thus the access capability process simply has to resolve the required channel.

```
62     public class AccessClientCapability implements CSProcess {

63       private ChannelInput eventChannel;

64       public AccessClientCapability(ChannelInput eventChannel) {
65         this.eventChannel = eventChannel;
66       }

67       public void run () {
68         final String eventType = (String) eventChannel.read();

69         String serviceName = ( eventType == "Create New Meeting" ) ? "N" : "F" ;

70         final NetChannelLocation serverLoc = CNS.resolve(serviceName);
71         final NetChannelOutput toServer =
72                                 NetChannelEnd.createOne2Net(serverLoc);
73         final NetChannelInput processReceive = Mobile.createNet2One();
74         toServer.write(processReceive.getChannelLocation());
75         final MobileProcess theProcess = (MobileProcess)processReceive.read();
76         new ProcessManager(theProcess).run();
77       }
78     }
```

Listing 19-4 The Access Client Capability Process

The output channel location is `resolved` using the determined `serviceName` {70} after which, a net output channel `toServer` can be created {71-72} that connects this mobile device to the service provider's server. A net input channel is then created, `processReceive` using the `Mobile` package, because this channel will be used to load the desired service process capability. The `toServer` channel is then used to `write` {74} the location of the `processReceive` channel to the server by calling the `getChannelLocation` method. The location of a net channel comprises its IP address, port number and a channel number, which together uniquely identify the input location. A net output channel end can be created from this information. The server process will thus create an output channel connected to the `processReceive` input channel upon which it will write the required mobile process.

The required service process is then read into `theProcess` {75}, after which it is executed by creating an instance of the `ProcessManager` class {76}. The `AccessClientCapability` process terminates once the service process itself terminates. The resources used by the mobile device are thereby recovered and no data or process code resulting form these interactions are left in the device.

19.4 New Meeting Processing

The service processes downloaded into the mobile device will have a net channel already created by which it can send data from the mobile device to the server. However, the service process will also need to input data from the server, informing the mobile device user where the group of people is located. In order to do this the location of a net input channel will have to be sent to the server. The simplest way of doing this is to create an object, `MeetingData`, by which data can be transferred between the mobile device and the server.

19.4.1 The MeetingData Class Definition

The properties of the `MeetingData` class, Listing 19-5, comprise the `NetChannelLocation` of the `returnChannel` {80}, which the server uses to return data to the mobile Device. The meeting server will ensure a quality of service by ensuring that it only allows the parallel execution, in different mobile devices, of a specific maximum number of client processes. The identity of the client used is stored in the property `clientId` {81}. Two properties, `meetingName` and `meetingPlace` are used {82, 83} to hold the name of the meeting and the place where they are meeting, with `attendees` indicating the number of people who have already joined the group {84}.

```
79     public class MeetingData implements Serializable {

80       private NetChannelLocation returnChannel;
81       private int clientId;
82       private String meetingName;
83       private String meetingPlace;
84       private int attendees;

85       // constructors omitted
86       // several getter and setter methods omitted

87       public NetChannelLocation getReturnChannel() {
88           return returnChannel;
89       }
90       public void setReturnChannel(NetChannelLocation   returnChannel) {
91           this.returnChannel = returnChannel;
92       }
93     }
```

Listing 19-5 The MeetingData Class (part)

The class `MeetingData` is written in Java because it is accessed within the mobile device and thus requires constructor, get and set methods, most of which have been omitted as they are well understood by Java developers and further can be easily created by an IDE such as Eclipse. The `get` and `set` methods for manipulating the `returnChannel` property have been shown {87-92}. This data object will also be accessed on the server side of the system, which is coded in Groovy, and therefore these properties will also be accessed using the ‘dot’ notation available in Groovy, thereby demonstrating the fact that Groovy and Java coding can be mixed in the same application.

19.4.2 The New Meeting Client Capability

The `NewMeetingClientCapability` process, Listing 19-6, contains two properties that are used by the server. The clientId property {95} is the unique identifier of this instance of the process. The `NetChannelLocation` property {96}, `clientServerLocation`, is a location that this process can use to create a net output channel by which the mobile device outputs data to the server. The remaining properties {97-101} are channels by which the `NewMeetingClientCapability` process communicates with its associated graphical user interface process.

The `run` method {102} initially creates the channel that connects the mobile device to the server as `client2Server` {103-104}. It then creates the `NetChannelInput server2Client` {105} that is used to receive inputs from the server. The corresponding net input channel location will be sent to the server as the `returnChannel` in an instance of `MeetingData` {106} called `clientData`. The properties of `clientData` are set using the appropriate `set` methods {107-110}. The `returnChannel` is set to the channel location of `server2Client` {107}. The identifier of the client being used is set {108}. The name of the meeting and the place where the group is meeting are read from the user interface and stored in the corresponding properties of `clientData` {109, 110}. This data object is now complete and can be written to the server using the `client2Server` net channel {111}. The process then waits for a response from the server, thereby implementing the client-server behaviour. The server process, see the description of the `Meeting` process (21.6.5), creates a net output channel based upon the location for `server2Client`, it receives in the `returnChannel` property of the `clientData` object.

```
94      public class  NewMeetingClientCapability implements CSProcess {

95        private int clientId;
96        private NetChannelLocation clientServerLocation;

97        private ChannelInput meetingNameEvent;
98        private ChannelInput meetingLocationEvent;
99        private ChannelOutput registeredConfigure;
100       private ChannelOutput registeredLocationConfigure;
101       private ChannelOutput attendeesConfigure;

102       public void run () {
103         final NetChannelOutput client2Server =
104                                     Mobile.createOne2Net(clientServerLocation);
105         final NetChannelInput server2Client = Mobile.createNet2One();

106         MeetingData clientData = new MeetingData();
107         clientData.setReturnChannel ( server2Client.getChannelLocation());
108         clientData.setClientId (clientId);
109         clientData.setMeetingName ( (String) meetingNameEvent.read());
110         clientData.setMeetingPlace ( (String) meetingLocationEvent.read());
111         client2Server.write(clientData);

112         final MeetingData replyData = (MeetingData) server2Client.read();
113         if ( replyData.getAttendees() == 1 ) {
114           registeredConfigure.write("Registered");
115         }
116         else {
117           registeredConfigure.write("ALREADY Registered");
118         }
119         registeredLocationConfigure.write(replyData.getMeetingPlace());
120         attendeesConfigure.write(new String (" " + replyData.getAttendees()) );
121       }
122     }
```

Listing 19-6 The New Meeting Client Capability (part)

An object of type `MeetingData` is read from `server2Client` into `replyData` {112}, the contents of which are then used to output appropriate messages on the user interface of the mobile device. The `attendees` property indicates whether this is genuinely a new meeting because the person creating the meeting is the first attendee {113, 114} or whether the meeting has already been registered {117}. The location of the meeting and the number of attendees can be written to the user interface using the configure channels of the `ActiveLabels` contained within the user interface process {119, 120}. Note that the property `meetingPlace` will only be modified by the `Meeting` process if the meeting has been previously registered.

19.5 Find Meeting Processing

Find meeting processing is very similar to that required for creating a new meeting and thus its structure is very similar. The `FindMeetingClientCapability` process is shown in Listing 19-7. The initialisation of the required channels and `clientData` are identical except that a location for the meeting is not known as it will be returned from the `Meeting` process, assuming the meeting has been registered {131-138}. Once the `replyData` has been received {139}, it can be used to output data to the `ActiveLabels` on the associated user interface process using their configure channels. If the number of `attendees` is returned as zero {140} then the meeting has not yet been registered. At this point the process could have automatically loaded the `NewMeetingClientProcess`, but this option has not been chosen for ease of explanation.

```
public class  FindMeetingClientCapability implements CSProcess {

  private NetChannelLocation clientServerLocation;
  private int clientId;

  private ChannelInput meetingNameEvent;
  private ChannelOutput registeredConfigure;
  private ChannelOutput registeredLocationConfigure;
  private ChannelOutput attendeesConfigure;

  public void run () {
    final NetChannelOutput client2Server =
                              Mobile.createOne2Net(clientServerLocation);
    final NetChannelInput  server2Client = Mobile.createNet2One();

    final MeetingData clientData = new MeetingData();
    clientData.setReturnChannel ( server2Client.getChannelLocation() );
    clientData.setClientId ( clientId );
    clientData.setMeetingName ( (String) meetingNameEvent.read() );
    client2Server.write(clientData);

    final MeetingData  replyData = (MeetingData) server2Client.read();
    if ( replyData.getAttendees() == 0 )
    {
      registeredConfigure.write("NOT Registered");
    }
    else
    {
      registeredConfigure.write("Registered");
      registeredLocationConfigure.write(replyData.getMeetingPlace() );
      attendeesConfigure.write(new String (" " + replyData.getAttendees()));
    }
  }
}
```

Listing 19-7 The Find Meeting Client Capability (part)

19.6 The Server Side

The server side processing comprises a number of processes running in parallel as shown in Figure 19-1.

Each capability is managed by a pair of processes, a server and a sender. The server registers the service channel with the `CNSServer` which can be subsequently resolved by the specific capability when providing the service within the mobile device. The access server and sender are specific to the requirements of the initial access by a mobile device because there is no requirement to access the meeting database process. All the processes of the meeting organiser are constructed in Groovy.

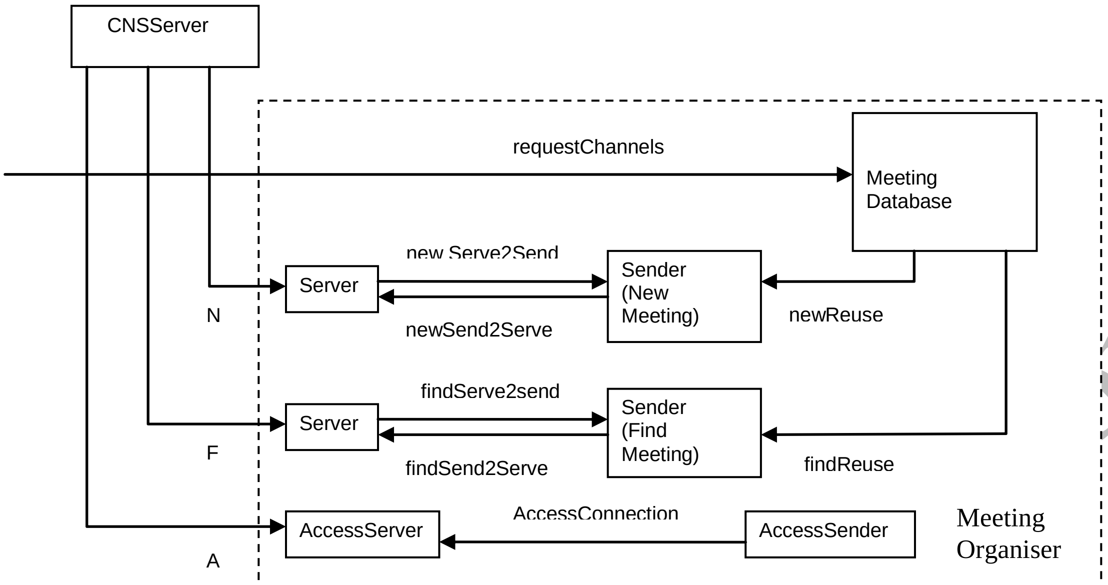

Figure 19-1 The Meeting Organiser Internal Process Architecture

19.6.1 The Access Server Process

The `AccessServer` process is shown in Listing 19-8 and simply creates an instance of the process `MultiMobileProcessServer` from the package `org.jcsp.net.mobile` {155}. The properties of which are the name of the channel by which the service is accessed “`A`” and the name of the channel by which instances of the `AccessClientProcess` are sent to this process. In this case the channel is passed as a property `fromAccessSender` {153} of the `AccessServer` process. Once `theServer` has been constructed it can be executed {156}. An instance of `MultiMobileProcessServer` simply responds to requests received on its named service channel with instances of mobile processes received on its input channel. This simplicity hides the fact that the underlying system is capable of dynamically loading class files over the network. In particular, classes are only loaded as they are needed. Thus in this case the class file for the `AccessClientProcess` will be loaded. When this executes on the mobile device it will be found that the `AccessClientCapability` and `AccessClientUserInterface` processes are required and a request for these processes’ class definitions will be generated automatically by the mobile device, to which the `AccessServer` responds.

```
152    class AccessServer implements CSProcess {

153      def ChannelInput fromAccessSender

154      void run() {
155        def theServer = new MultiMobileProcessServer("A", fromAccessSender)
156        new PAR ([theServer]).run()
157      }
158    }
```

Listing 19-8 The Access Server Process

19.6.2 The Access Sender Process

The `AccessSender` process shown in Listing 19-9 has one property, `toAccessServer` {160} that is the channel that connects it to the `AccessServer` process. The `run` method {161} simply defines an instance of the `AccessClientProcess`, `AClient` {162}, which is then repeatedly written to the `AccessServer` process {164}. Any number of mobile device users can be running the `AccessClientProcess` at the same time without restriction.

```
159    class AccessSender implements CSProcess {

160      def ChannelOutput toAccessServer

161      void run() {
162        def AClient = new AccessClientProcess()
163        while (true) {
164          toAccessServer.write(AClient)
165        }
166      }
167    }
```

Listing 19-9 The Access Sender Process

19.6.3 The Server Process Definition

The Server process shown in Listing 19-10 has three properties {169-171} all of which are passed directly to the instance of the `MultiMobileProcessServer` that is defined as `theServer` { 173-174} and then run {175}. In this construction the `MultiMobileProcessServer` responds to requests on the service channel represented by `serviceName`. It requests an instance of a mobile process from its associated `Sender` process on its `toSender` channel, which it reads from its `fromSender` channel. The mobile process is then written the net channel location obtained from the channel associated with the `serviceName` {64, 65}. This interaction is contained within the process `MultiMobileProcessServer.`

```
168    class Server implements CSProcess {

169      def ChannelInput fromSender
170      def ChannelOutput toSender
171      def String serviceName

172      void run() {
173        def theServer = new MultiMobileProcessServer(serviceName,
174                                                     fromSender, toSender)
175        new PAR ([theServer]).run()
176      }
177    }
```

Listing 19-10 The Server Process

19.6.4 The Sender Process Definition

Listing 19-11 shows the definition of the `Sender` process that implements a relatively crude form of service quality management. The properties `toServer` {179} and `fromServer` {180} are used to create the connections between the `Server` and `Sender` process described above.

```
178    class Sender implements CSProcess {

179      def ChannelOutput toServer
180      def ChannelInput fromServer
181      def List clients
182      def ChannelInput reuse
183
184      void run() {
185        def serviceUnavailable = new NoServiceClientProcess()
186        def n = clients.size()
187        def clientsAvailable = [ ]
188        for (i in 0 ..< n) {
189          clientsAvailable.add(clients[i])
190        }
191        def alt = new ALT ([reuse, fromServer])

192        while (true) {
193          def index = alt.select()
194          if (index == 0 ) {
195            def use = reuse.read()
196            clientsAvailable.add(clients[use])
197          }
198          else {
199            fromServer.read()
200            if (clientsAvailable.size() > 0 ) {
201              toServer.write(clientsAvailable.pop())
202            }
203            else {
204              toServer.write(serviceUnavailable)
205            }
206          }
207        }
208      }
209    }
```

Listing 19-11 The Sender Process

The `List clients` {181} is a list of service processes that this `Sender` process can send to its `Server`. The channel `reuse` {182} is used to inform the `Sender` process that a specific client can be reused. A mobile process called `NoServiceClientProcess` has been defined that informs the user that resources are currently not available and that they should try again later. The definition of this process has not been explained but is similar to the other downloadable processes. An instance of this process is defined as `serviceUnavailable` {185}. The number of client processes in the list `clients` is obtained as `n` {186}. A list of clients that are available is created in another list, `clientsAvailable` {187}, which is populated by adding each element of clients {188-190}.

The `Sender` process can receive inputs on either of its input channels and thus these form the guards of the alternative `alt` {191}. The main loop of the process {192} determines the `index` of the alternative that has been selected {193}. In the case of an input on the `reuse` channel {194}; the identifier of a client that can be re-used can be read from the `reuse` channel {195}. The corresponding `client` is added to `clientsAvailable` {196}.

If the input is a signal on the `fromServer` channel {198}, it is `read` {199}. If `clientsAvailable` is not empty {200} then the next process from `clientsAvailable` is popped and written to the channel `toServer` {201}. Otherwise, the process `serviceUnavailable` is written to the channel `toServer` {204}. In this manner the number of service processes running in parallel on different users mobile devices is restricted to the number of processes initially passed to the Sender process when it is constructed.

19.6.5 The Meeting Database Process

The Meeting process is shown in Listing 19-12. The `List` property `requestChannels` {211} comprises the input ends of the net channels that are passed to each of the `NewMeetingClientProcess` and `FindMeetingClientProcess` as their `clientServerLocation` property {96, 124}. The properties `nReuse` and `fReuse` {212, 214} provide the channel connection between the `Meeting` process and the two

`Sender` processes, see Figure 19-1. The properties `newClients` and `findClients` {213, 215} are the number of `NewMeetingClientProcesses` and `FindMeetingClientProcesses` respectively. `MeetingMap` {217} is the database that maintains the list of registered meetings and their locations. The process alternates over the `requestChannels` {218}. The main loop of the process {219} determines the enabled channel within `alt` {220}. It is presumed that the `requestChannels` are ordered such that inputs from the `NewMeetingClientProcesses` precede those of the `FindMeetingClientProcesses` and that the number of `newClients` can be used to differentiate the required processing {222, 238}.

In both cases the request is read into an object of type `MeetingData` {223, 239}. A `replyData` object is created of the same type {224, 240} and then a net output channel, `reply`, is constructed from the `returnChannel` property of the object that has been read {225, 241}. A test is then undertaken to determine whether or not the meeting already exists {226, 242} and the subsequent processing depends on its outcome.

In the case where the request is to create a new meeting that already exists {227, 228} the required `replyData` is obtained from the existing `meetingMap` entry and the number of attendees incremented; otherwise the `replyData` is constructed from `newMeeting` and the number of `attendees` set to 1 {231, 232}. The `replyData` is then put back into the `meetingMap` {234} and written to the previously created `reply` channel {235}. Finally, the value of `clientId` that was read is written to the `nReuse` channel thereby enabling the re-use of the client {236}.

```
210   class Meeting implements CSProcess {

211     def List requestChannels
212     def ChannelOutput nReuse
213     def int newClients
214     def ChannelOutput fReuse
215     def int findClients

216     void run() {
217       def meetingMap = [ : ]
218       def alt = new ALT (requestChannels)

219       while (true) {
220         def index = alt.select()
221         switch (index) {

222           case 0 ..< newClients :
223             def newMeeting = requestChannels[index].read()
224             def replyData = new MeetingData()
225             def reply = Mobile.createOne2Net(newMeeting.returnChannel )
226             if ( meetingMap.containsKey(newMeeting.meetingName ) ) {
227               replyData = meetingMap.get(newMeeting.meetingName )
228               replyData.attendees = replyData.getAttendees() + 1
229             }
230             else {
231               replyData = newMeeting
232               replyData.attendees = 1
233             }
234             meetingMap.put ( replyData.meetingName, replyData)
235             reply.write(replyData)
236             nReuse.write(replyData.clientId )
237             break

238           case newClients ..< (findClients + newClients) :
239             def findMeeting = requestChannels[index].read()
240             def replyData = new MeetingData()
241             def reply = Mobile.createOne2Net(findMeeting.returnChannel )
242             if ( meetingMap.containsKey(findMeeting.meetingName ) ) {
243               replyData = meetingMap.get(findMeeting.meetingName )
244               replyData.attendees = replyData.attendees + 1
245               meetingMap.put ( replyData.meetingName, replyData)
246             }
247             else {
248               replyData = findMeeting
249               replyData.attendees = 0
250             }
251             reply.write(replyData)
252             fReuse.write(replyData.clientId )
253             break
254         }
255         meetingMap.each{println "Meeting: ${it.key}"}
256       }
257     }
258   }
```

Listing 19-12 The Meeting Database Process

The processing for the case where the input request is from a find meeting client process {238-254} is very similar and depends upon whether or not the meeting has already been created. If the meeting already exists then the number of `attendees` is incremented {244} and the `meetingMap` entry replaced{245}; otherwise the returned value of `attendees` is set to zero and no entry is placed in the `meetingMap` {249}. At the end of each interaction the entries in the `meetingMap` are printed {255} to the console window as a means of checking the operation of the `Meeting` process.

19.6.6 The Meeting Organiser

The script to run the meeting system is shown in Listing 19-13. Initially, the IP address of the node upon which the CNS is running is obtained by means of user interaction {259} and this value is used initialise the node running the Meeting Organiser {260}. The number of concurrent New and Find Meeting Clients is then obtained {261, 262} as `nSize` and `fSize` respectively.

```
259      def CNS_IP = Ask.string("Enter IP address of CNS: ")
260      Mobile.init(Node.getInstance().init(new TCPIPNodeFactory(CNS_IP)))

261      def nSize = Ask.Int("Number of Concurrent New Meeting Clients? ", 1, 2)
262      def fSize = Ask.Int("Number of Concurrent Find Meeting Clients? ", 1, 3)

263      def netChannels = []

264      def NMCList = []
265      for (i in 0 ..< nSize) {
266        def c = Mobile.createNet2One()
267        netChannels << c
268        NMCList << new NewMeetingClientProcess(c.getChannelLocation(), i )
269      }

270      def FMCList = []
271      for (i in 0 ..< fSize) {
272        def c = Mobile.createNet2One()
273        netChannels << c
274        FMCList << new FindMeetingClientProcess(c.getChannelLocation(), i )
275      }

276      def newServe2Send = Channel.createOne2One()
277      def newSend2Serve = Channel.createOne2One()
278      def newReuse = Channel.createOne2One()
279      def findServe2Send = Channel.createOne2One()
280      def findSend2Serve = Channel.createOne2One()
281      def findReuse = Channel.createOne2One()
282      def accessConnection = Channel.createOne2One()

283      def processList = [
284                        new AccessSender(toAccessServer:accessConnection.out()),
285                        new AccessServer(fromAccessSender:accessConnection.in()),
286                        new Server( fromSender:newSend2Serve.in(),
287                                    toSender:newServe2Send.out(),
288                                    serviceName: "N"),
289                        new Sender( toServer:newSend2Serve.out(),
290                                    fromServer:newServe2Send.in(),
291                                    reuse:newReuse.in(), clients: NMCList),
292                        new Server( fromSender:findSend2Serve.in(),
293                                    toSender:findServe2Send.out(),
294                                    serviceName: "F"),
295                        new Sender( toServer:findSend2Serve.out(),
296                                    fromServer:findServe2Send.in(),
297                                    reuse:findReuse.in(), clients: FMCList),
298                        new Meeting( requestChannels: netChannels,
299                                     nReuse: newReuse.out(),
300                                     newClients: nSize,
301                                     fReuse: findReuse.out(),
302                                     findClients: fSize )
303                         ]
304      new PAR(processList).run()
```

Listing 19-13 The Meeting Organiser Script

The `list` `netChannels` {263} is used to hold all the net input channels that form the `requestChannels` property of `Meeting` {298}, see Figure 19-1. The `list` `NMCList` is used to hold the list of `NewMeetingClientProcesses` that will be passed as the `clients` property of a `Sender` {291}. The `for` loop {265-269} iterates over `nSize` and creates a `NetChannelInput c` {266}, which is appended to `netChannels` {267}. An instance of `NewMeetingClientProcess` is then appended to `NMCList` {268}, in which the `clientServerLocation` property {96} is assigned the `NetChannelLocation` of `c` and the `clientId` property {95} is assigned the loop variable `i`. Recall that the `NewMeetingClientProcess` is written in Java and thus there will be an explicit constructor for this process. Lines {270-275} build the same structures for the `FindMeetingClientProcesses`. Note that the `netChannels` list contains the net input channels for both new and find meeting clients in the order expected by the `Meeting` process {222, 238}.

The `One2OneChannels` required to connect the processes within the Meeting Organiser are now defined {276-282} as per Figure 19-1. The `processList` {283} creates the network of processes required to implement the architecture.

19.7 System Operation

In order to observe the operation of the system; it is first necessary to run an instance of CNS. The `MeetingOrganiser` should then be run and a suitable number of new and find meeting client processes should be created as per the interaction {261, 262}. The `MeetingOrganiser` will now be initialised and is awaiting interactions. In the first instance this can be tested in-situ by running an instance of `UASSSClient`. The Access Client User Interface will appear and this can then be used to create a meeting. The console window associated with the `MeetingOrganiser` will show that the meeting of the input name has been created. The windows associated with `UASSSClient` should be closed and it will be noticed that the process terminates completely. Another instance of `UASSSClient` should be run and used to find the meeting that has been created and the response should indicate that there are now 2 attendees at the meeting. By running sufficient `UASSSClient` instances it will be possible to request more instances of a new or find meeting processes that the `Sender` manages, in which case the Unavailable Service process will be transferred to the `UASSSClient`.

19.8 Summary

This chapter has shown that mobile processes that utilise mobile channel connections can be created over communication networks. The `org.jcsp.net.mobile` package completely hides from the programmer any need to understand how the process and data object class definitions are loaded, dynamically, over the network. Furthermore, the package implements a relatively secure means of obtaining class definitions that are not known to a particular node. This means that class definitions are only obtained from nodes that have been previously accessed to find a class definition. In addition, because these classes have to be `Serialized` for transfer over a network it means that a further level of security is available in that any class that is received that does not have the designated class serialisation identification will be rejected. Interested readers are referred to [Chalmers et al, CPA 2007 mobile channels].