9 External Events: Handling Data Sources

External event handling is shown to be very amenable to concurrent and parallel programming by:

- managing events as communications rather than interrupts
- exploitiong the client-server pattern to build architectures that embody event handling
- separating event handling from processing using a buffer that is a pure server, thereby ensuring deadlock freedom
- enabling analysis of system behaviour
- permitting determination on the time upper bound for event procesisng
- processing events from many different sources

Traditionally, real-time systems that respond to external stimuli have utilised interrupts. An interrupt is a hardware signal that indicates that a device needs to be serviced and which causes the processors' central processing unit to interrupt the current program and invoke the device's service routine returning to the original program once the device has been serviced. Over the years a great deal of effort has been expended in trying to make interrupt based systems more efficient and easier to program. However, the basic problem still remains that an interrupt causes the halting of the current program, saving its state and then starting an interrupt service routine. The problems become more complex when an interrupt service routine is itself interrupted by a device with a higher priority. The approach has been to reduce the amount of time when interrupts are disabled. This in itself leads to further problems because it is very difficult then to foresee the precise nature of interactions between interrupts that can then take place. It is these indeterminate interactions that cause problems when systems are running because it is impossible to test for all the possible interactions, especially in highly complex systems.

The framework built so far, using parallelism and alternation to capture non-deterministic behaviour, provides a means of describing, implementing and analysing such event driven systems. Rather than building a system that interrupts itself on receipt of an event notification; build a system that expects such events to occur so that programmers can better reason about its behaviour. In effect, the external event is considered to be the same as a channel communication. Furthermore, the client-server design pattern gives us a handle by which the system can be analysed to ensure that unwanted interactions between events do not occur.

9.1 An Event Handling Design Pattern

The aim of the pattern is to allow the system to respond to external events as quickly as possible. However, the situation has to be considered that events may occur so rapidly that the system cannot deal with all the events. Such a situation tends to overwhelm interrupt based systems. The pattern also has to take account of any priority requirement the application may have, thereby influencing the order in which events are handled. Such ordering of the handling of events may result in some events being lost. However, if the designer is aware of this situation then steps can be taken at design time to ameliorate their effects.

The key to building an event handling system is that the process dealing with receipt of the event has to be ready, waiting, for the associated channel (event) communication, so it can be read and the associated data passed on to another process. The event receiving process can then return to the state of waiting for the next event communication. If we connect the event receiving process directly to the event processing process then the event receiver might be delayed by having to wait for the processing of another event to finish. We thus require an intermediate stage that separates event receiving from event processing. This can be implemented by some form of buffer. More specifically, the buffer should always be ready to receive a communication from the event receiver process. This may mean that previous buffered values may be overwritten. In addition, a mechanism by which buffered values can be requested from the buffer process, in a manner similar to that used in the Queue process described in Chapter 5. The resulting process structure is shown in Figure 9-1, to which a client-server labelling has been added. It demonstrates that there are no client-server loops in the architecture.

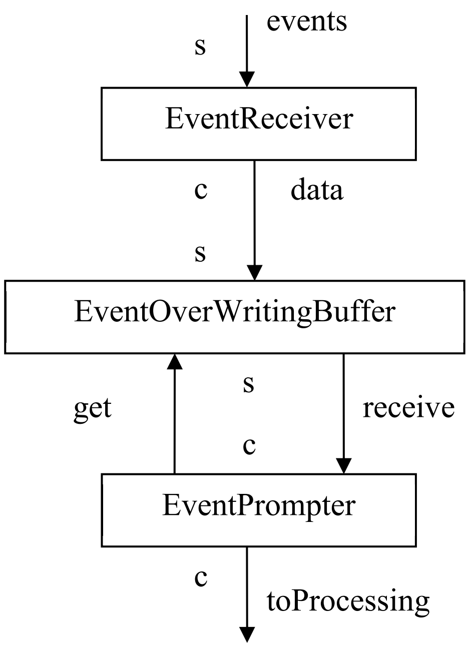

Figure 9-1 Event Handling Design Pattern

Events are received by the EventReceiver and immediately sent to the EventOverwritingBuffer so that EventReceiver is ready to read the next event. The EventPrompter indicates that it wants to get some data, which it will receive immediately from the EventOverwritingBuffer if data is already buffered or it will have to wait until an event has been input. The EventPrompter then writes the data to the rest of the system where it is processed. Thus it is EventPrompter that has to wait until subsequent processing can be undertaken, allowing EventReceiver always ready to read an event. Later we shall show that the time required to process events can in fact be calculated to give an absolute upper bound on the performance of the system. Such a bound cannot be calculated for interrupt based systems. In addition, the client-server labelling shows that the pattern has no deadlock or livelock inherent within it and thus provided the rest of the system is also deadlock and livelock free ensures that the system will behave as expected. This is easily deduced because the EventOverwritingBuffer is a pure server and hence any client-server circuit cannot exist.

9.2 Utilising the Event Handing Pattern

The pattern can be transformed easily into a set of processes that achieves its effect.

9.2.1 The EventReceiver Process

Listing 9-1 shows the definition of the `EventReceiver` process. The process has `eventIn` {12} and `eventOut` {13} channel properties. In this implementation, every input that is `read` from `eventIn` is immediately `writ`ten to the `eventOut` channel {17}. In a realistic implementation it would probably be necessary to determine the source of the event and possibly read some data from a hardware register. However, we can presume that such additional processing would not create any substantial delay within the system because the event would not be raised if there was no reason.

```
10 class EventReceiver implements CSProcess {
11
12   def ChannelInput eventIn
13   def ChannelOutput eventOut
14
15   void run() {
16     while (true){
17       eventOut.write(eventIn.read())
18     }
19   }
20 }
```

Listing 9-1 The EventReceiver Process

9.2.2 The Event Overwriting Buffer Process

The implementation of EventOWBuffer is shown in Listing 9-2.

```
class EventOWBuffer implements CSProcess {

  def ChannelInput inChannel
  def ChannelInput getChannel
  def ChannelOutput outChannel

  void run () {
    def owbAlt = new ALT ( [inChannel, getChannel] )

    def INCHANNEL = 0
    def GETCHANNEL = 1
    def preCon = new boolean[2]
    preCon[INCHANNEL] = true
    preCon[GETCHANNEL] = false
    def e = new EventData ()
    def missed = -1

    while (true) {
      def index = owbAlt.priSelect ( preCon )

30        switch ( index ) {
31          case INCHANNEL:
32            e = inChannel.read().copy()
33            missed = missed + 1
34            e.missed = missed
35            preCon[GETCHANNEL] = true
36            break
37
38          case GETCHANNEL:
39            def s = getChannel.read()
40            outChannel.write ( e )
41            missed = -1
42            preCon[GETCHANNEL] = false
43            break
44
45        } // end switch
46      } // end while
47    } // end run
48 }
```

Listing 9-2 The EventOWBuffer Process

The channel `inChannel` {12} inputs data from the `EventReceiver` process. The `getChannel` {13} receives a signal from `EventPrompter` whenever that process requires data. The response to `EventPrompter` is to output event data on the channel `outChannel` {14}. The process receives inputs on its input channels over which it must alternate as the order in which such inputs are read cannot be determined. This is captured in the definition of `owbAlt` {17}. The `EventOWBuffer` also has to capture the behaviour that requests for data from the `EventPrompter` process can only be allowed when the buffer contains data. To this end we used pre-conditions on `owbAlt` in a manner similar to that used in the `Queue` process described in Chapter 5. The constants `INCHANNEL` {19} and `GETCHANNEL` {20} are used to access the elements of the `preCon` {21} `boolean` array and also to identify the `cases` within the `switch` that implements the main processing loop. The initial values of the `preCon` elements can be specified as follows. The process is always willing to accept inputs on its `inChannel` and thus this element is always `true` {22}. Initially there is no data in the buffer and thus requests to get data from `EventPrompter` must not be permitted and thus that pre-condition has to be set `false` {23}. The actual buffer is represented by the variable `e` {24} and is of type `EventData`, see Listing 9-4. The variable `missed` {25} will count the number of times the data in the buffer `e` was overwritten and will be passed through the system so that its performance can be analysed. It is initialised to -1 so that when the next event is read its value will be considered not to have been overwritten because the value of `missed` will then be 0.

The main loop of EventOWBuffer {27–46} initially determines the `index` of the enabled channel, with priority being given to `inChannel` {17} because we always want `EventReceiver` to be ready to read the next event. In that case, the event data is read from `inChannel` {32} and a deep copy is made into the buffer variable `e`. The interface `JCSPCopy`, defined in `org.jcsp.groovy`, defines an abstract method `copy()` that can be used to make a deep copy of an object. Recall that if an object is transferred from one process to another then if these processes are on the same processor then this communication is achieved by passing an object reference. We must ensure that two processes do not access the same object at the same time and hence the need to make a deep copy of the object. The value of `missed` is incremented {33} and saved in the buffer variable `e` {34}. The `preCon` element `GETCHANNEL` can now be set `true` {35} because there is data in the buffer that can be sent to `EventPrompter` following a request for data.

Once the buffer contains data then requests for data can be read from the `getChannel` {39} and the contents of the buffer are immediately written to the `outChannel` {40}. This interaction ensures the process behaves like a server. The `preCon` element `GETCHANNEL` must now be set `false` {42} because there is no longer any data in the buffer and likewise the variable `missed` must be reset to -1 {41}.

9.2.3 The Event Prompter Process

This process is shown in Listing 9-3. This process has channel properties {12–14} that reflect the process structure shown in Figure 9-1.

```
10 class EventPrompter implements CSProcess {
11
12   def ChannelInput inChannel
13   def ChannelOutput getChannel
14   def ChannelOutput outChannel
15
16   void run () {
17     def s = 1
18     while (true) {
19       getChannel.write(s)
20       def e = inChannel.read().copy()
21       outChannel.write( e )
22     }
23   }
24 }
```

Listing 9-3 The EventPrompter Process

A signal is written to the `getChannel` {19}, the completion of which may be delayed until the `EventOWBuffer` contains event data. The response from `EventOWBuffer` is immediately read into a variable `e` {20} and also uses the `copy()` method to ensure that the data cannot be modified as it resides within the `EventPrompter` before being output to the next process. The data is then written {21} to the `outChannel`, where yet again a delay may be incurred due to the processing system not being in a state where the data from this event source can be processed.

9.2.4 The EventData Class

`EventData` contains three properties for this explanatory description comprising `source` {12}, `data` {13} and `missed` {14}, shown in Listing 9-4. The `source` is used to indicate from which event source the event came. The actual data value sent by the event is contained in `data`. The class implements the `Serializable` interface so that `EventData` objects can be communicated over networks. The interface `JCSPCopy` is implemented so that the `copy` method {16–21} can be defined that makes a deep copy of `EventData` objects. A `toString` method has also been provided so that event data can be more easily output {23–29}.

```
10 class EventData implements Serializable, JCSPCopy {
11
12   def int source = 0
13   def int data = 0
14   def int missed = -1
15
16   def copy() {
17     def e = new EventData ( source: this.source,
18                             data: this.data,
19                             missed: this.missed )
```

```
    return e
  }

  def String toString() {
    def s = "EventData -> [source: "
    s = s + source + ", data: "
    s = s + data + ", missed: "
    s = s + missed + "]"
    return s
  }

}
```

Listing 9-4 The EventData Class Definition

9.2.5 The EventHandler Process

The `EventHandler` process is the parallel composition of the the processes shown in Figure 9-1, as shown in Listing 9-5.

The Event Handler process is written to accept input events on its `inChannel` {12}. The events are then output on its `outChannel` {13}. The process has three internal channels, `get`, `transfer` and `toBuffer` {16–18}. These are used to connect the processes that are created in the `HandlerList` {20–28}. The processes used in the `HandlerList` have been previously described in Sections 9.2.1 to 9.2.3. The processes are then invoked by a `PAR` {29}.

```
10 class EventHandler implements CSProcess {
11
12   def ChannelInput inChannel
13   def ChannelOutput outChannel
14
15   void run () {
16     def get = Channel.one2one()
17     def transfer = Channel.one2one()
18     def toBuffer = Channel.one2one()
19
20     def handlerList = [ new EventReceiver ( eventIn: inChannel,
21                                             eventOut: toBuffer.out()),
22                         new EventOWBuffer ( inChannel: toBuffer.in(),
23                                             getChannel: get.in(),
24                                             outChannel: transfer.out() ),
25                         new EventPrompter ( inChannel: transfer.in(),
26                                             getChannel: get.out(),
27                                             outChannel: outChannel )
28                       ]
29     new PAR ( handlerList ).run()
30   }
31 }
```

Listing 9-5 The EventHandler Process Definition

9.3 Analysing Performance Bounds

The ability of the design pattern to handle repeated events can be determined for two different cases. The first and simplest case occurs when there is no outstanding request for data from the `EventPrompter` process. The time to handle an event can be calculated by adding together the processing times of lines Listing 9-1 17, and Listing 9-2 lines 32–35 plus the time to undertake a single communication from `EventReceiver` to `EventOWBuffer`. This value can be determined by calculation if the time to execute each statement can be determined.

The second case is slightly more complex and concerns the situation when `EventOWBuffer` has just accepted a request to get data from `EventPrompter` and an event arrives at `EventReceiver`. The consequent processing delay comprises Listing 9-2 lines 39–42 and Listing 9-3 line 20 plus the time taken to undertake a single communication from `EventOWBuffer` to `EventPrompter`. Thus this time, plus the time to actually process the event, which is the same as the first case, gives the total time that is required to handle an event. This therefore gives an upper bound for the time to process an event and thus the maximum rate at which events can be handled in the worst case scenario. On a modern processor these times will be measured in nanoseconds. The fact that the processing system might not be able to keep up with such a rate merely points to a possible deficiency in the system design and not a failure of the ability to use parallel processing techniques to handle events.

9.4 Simple Demonstration of the Event Handling System

The demonstration comprises an `EventHandler` process which is fed with 'events' by an `EventGenerator` process that outputs data values according to a uniformly distributed delay strategy. The `EventHandler` outputs its 'events' to another process that simulates the time it takes to process an event according to a different uniformly distributed delay strategy. Finally, the processed 'events' are printed using a GPrint process.

9.4.1 The Event Generator Process

The `EventGenerator` process, shown in Listing 9-6, itself comprises two parallel processes, `EventStream` and `UniformlyDistributedDelay`. The properties of the process are passed directly to these processes and will thus be described in the next sections.

```
10 class EventGenerator implements CSProcess {
11
12   def ChannelOutput outChannel
13   def int source = 0
14   def int initialValue = 0
15   def int minTime = 100
16   def int maxTime = 1000
17   def int iterations = 10
18
```

```
  void run () {
    def es2udd = Channel.one2one()
    println "Event Generator for source $source has started"

    def eventGeneratorList = [
          new EventStream ( source: source,
                           initialValue: initialValue,
                           iterations: iterations,
                           outChannel: es2udd.out() ),
          new UniformlyDistributedDelay ( minTime: minTime,
                                       maxTime: maxTime,
                                       inChannel: es2udd.in(),
                                       outChannel: outChannel )
          ]

    new PAR (eventGeneratorList).run()
  }
}
```

Listing 9-6 The EventGenerator Process

9.4.2 The Event Stream Process

Listing 9-7 shows the `EventStream` process, in which the `source` property {12} is used to identify the stream and which has an `initialValue` {13}. The process will output a stream of length `iterations` {14}. The stream of 'events' will be output on the channel `outChannel` {15}. The process uses the `upto` method to create the loop {20}. An event `e` is constructed {21} and then written to `outChannel` {22}. On completion the process outputs a message {25} as this will prove invaluable in understanding how the system functions.

```
10 class EventStream implements CSProcess {
11
12   def int source = 0
13   def int initialValue = 0
14   def int iterations = 10
15   def ChannelOutput outChannel
16
17   void run () {
18     def i = initialValue
19
20     1.upto(iterations) {
21       def e = new EventData ( source: source, data: i )
22       outChannel.write(e)
23       i = i + 1
24     }
25     println "Source $source has finished"
26   }
27 }
```

Listing 9-7 The EventStream Process

9.4.3 The Uniformly Distributed Delay Process

The `UniformlyDistributedDelay` process, shown in Listing 9-8, uses a random number generator {19} to produce a `delay` between `minTime` and `maxTime` {23}. The event data is read from `inChannel` {22} and after waiting for the `delay` {24} period it is output on `outChannel` {25}.

```
10 class UniformlyDistributedDelay implements CSProcess {
11
12   def ChannelInput inChannel
13   def ChannelOutput outChannel
14   def int minTime = 100
15   def int maxTime = 1000
16
17   void run () {
18     def timer = new CSTimer()
19     def rng = new Random()
20
```

```
    while (true) {
      def v = inChannel.read().copy()
      def delay = minTime + rng.nextInt ( maxTime - minTime )
      timer.sleep (delay)
      outChannel.write( v )
    }
  }
}
```

Listing 9-8 The UniformlyDistributedDelay Process

The effect of the `UniformlyDistributedDelay` process is to ensure that events are generated with delays that vary uniformly between `minTime` and `maxTime`.

9.4.4 Demonstration of a Single Stream Event Processing System

The script that invokes the system with a single source of events is shown in Listing 9-9. The collection of processes comprises the processes already described, executed in parallel. An additional `UniformlyDistributedDelay` process has been included to represent the varying time it takes to process an event. The events are passed to a `GPrint` process where they are simply printed. Of particular interest is the number of events that are missed.

```
def eg2h = Channel.one2one()
def h2udd = Channel.one2one()
def udd2prn = Channel.one2one()
def eventTestList = [
         new EventGenerator ( source: 1,
                              initialValue: 100,
                              iterations: 100,
                              outChannel: eg2h.out(),
                              minTime: 100,
                              maxTime:200 ),

         new EventHandler ( inChannel: eg2h.in(),
                            outChannel: h2udd.out() ),

         new UniformlyDistributedDelay ( inChannel:h2udd.in(),
                                         outChannel: udd2prn.out(),
                                         minTime: 1000,
                                         maxTime: 2000 ),

         new GPrint ( inChannel: udd2prn.in(),
                      heading : "Event Output",
                      delay: 0)
         ]

new PAR ( eventTestList ).run()
```

Listing 9-9 The Srcript Used to Invoke the Single Stream Event Handling System

The events are generated with a delay that varies between 100 and 200 milliseconds {18, 19}. The simulation of processing time {26, 27} varying between 1000 and 2000 milliseconds means we would expect around 8 or 9 events to be missed but will depend on the actual random values. A sample output from the system is shown in Output 9-1.

The first two events pass through the system without any delay because that is the time when the buffers within the system are being filled. Thereafter, data appears with varying numbers of events missed and in general these match what would be expected. The last three events are produced after the event generator has finished because they are buffered up within the system. It should be noted that a check of correctness of operation is possible because the data value, after the first, is equal to the previous output data value plus the number missed plus 1.

```
Event Output
Event Generator for source 1 has started
EventData -> [source: 1, data: 100, missed: 0]
EventData -> [source: 1, data: 101, missed: 0]
EventData -> [source: 1, data: 110, missed: 8]
EventData -> [source: 1, data: 122, missed: 11]
EventData -> [source: 1, data: 128, missed: 5]
EventData -> [source: 1, data: 140, missed: 11]
EventData -> [source: 1, data: 149, missed: 8]
EventData -> [source: 1, data: 159, missed: 9]
EventData -> [source: 1, data: 168, missed: 8]
EventData -> [source: 1, data: 176, missed: 7]
Source 1 has finished
EventData -> [source: 1, data: 186, missed: 9]
EventData -> [source: 1, data: 195, missed: 8]
EventData -> [source: 1, data: 199, missed: 3]
```

Output 9-1 A Sample Output from the Event Handling System

9.5 Processing Multiple Event Streams

Now that we have seen how to process a single stream of events; it becomes noteworthy to process multiple streams of events that are generated with varying uniformly distributed delays. We combine the event generation and event handling into a single process called `EventSource` shown in Listing 9-10.

```
10 class EventSource implements CSProcess {
11
12   def source
13   def iterations = 99
14   def minTime = 100
15   def maxTime = 250
16   def ChannelOutput outChannel
17
18   void run() {
19     def eg2h = Channel.one2one()
20     def sourceList = [ new EventGenerator ( source: source,
21                                             initialValue: 100 * source,
22                                             iterations: iterations,
23                                             minTime: minTime,
24                                             maxTime: maxTime,
25                                             outChannel: eg2h.out()),
26                        new EventHandler ( inChannel: eg2h.in(),
27                                           outChannel: outChannel)
28                      ]
29
30     new PAR (sourceList).run()
31   }
32 }
```

Listing 9-10 The Process that Generates Events and Handles Them

The `EventSource` process uses the processes `EventGenerator` and `EventHandler`, previously described in Sections 9.4.1 and 9.2.5. Each source can be assigned its own `minTime` {14, 23} and `maxTime` {15, 24} used in the `UniformlyDistributedDelay` process contained within each `EventGenerator`.

The output from each `EventSource` is output on its `outChannel` {16} to an `EventProcessing` process shown in Listing 9-11.

```
10 class EventProcessing implements CSProcess{
11
12   def ChannelInputList eventStreams
13   def minTime = 500
14   def maxTime = 750
15
16   void run() {
17     def mux2udd = Channel.one2one()
18     def udd2prn = Channel.one2one()
19     def pList = [
20                 new FairMultiplex ( inChannels: eventStreams,
21                                     outChannel: mux2udd.out() ),
22                 new UniformlyDistributedDelay ( inChannel:mux2udd.in(),
23                                                 outChannel: udd2prn.out(),
24                                                 minTime: minTime,
25                                                 maxTime: maxTime ),
26           new GPrint ( inChannel: udd2prn.in(),
27                        heading : "Event Output",
28                        delay: 0)
29           ]
30     new PAR (pList).run()
31   }
32 }
```

Listing 9-11 The EventProcessing Process Definition

The process `EventProcessing` receives inputs from each of the event streams on the `ChannelInputList eventStreams` {12}. The manner in which each event stream is selected is determined by the multiplexer {20–21}. The version shown in Listing 9-11 uses a `FairMultiplex` process, that ensures each stream is allocated an equal amount of the available output bandwidth. Other multiplexers can be used that have different properties, see `org.jcsp.groovy.util`. As in the single stream version the delay taken by processing the event is simulated by a `UniformlyDistributedDelay` process {22–25} before the events are printed using a `GPrint` process {26–28}.

The script that executes the multiStream version is shown in Listing 9-12.

```
10 def sources = Ask.Int ("Number of event sources between 1 and 9 ? ", 1, 9)
11
12 minTimes = [ 10, 20, 30, 40, 50, 10, 20, 30, 40 ]
13 maxTimes = [ 100, 150, 200, 50, 60, 30, 60, 100, 80 ]
14
15 def es2ep = Channel.one2oneArray(sources)
```

```
16
17 ChannelInputList eventsList = new ChannelInputList (es2ep)
18
19 def sourcesList = ( 0 ..< sources).collect { i ->
20             new EventSource ( source: i+1,
21             outChannel: es2ep[i].out(),
22             minTime: minTimes[i],
23             maxTime: maxTimes[i] )
24  }
15
26 def eventProcess = new EventProcessing ( eventStreams: eventsList,
27                                          minTime: 10,
28                                          maxTime: 400 )
29
30 new PAR( sourcesList + eventProcess).run()
```

Listing 9-12 The Run MultiStream Script

Initially the number of sources is determined by user interaction {10}. Then two lists {12, 13} are defined that specify the minimum and maximum time to be allocated to the `UniformlyDistributedDelay` process in each `EventSource`.

I joined MITAS because
I wanted real responsibility
The Graduate Programme
for Engineers and Geoscientists
www.discovermitas.com
Month 16
I was a construction supervisor in the North Sea advising and helping foremen solve problems
Real work
International opportunities
Three work placements
MAERSK

9.6 Summary

This chapter has shown that the adoption of parallel processing design techniques and implementation can shed a new light on age old computing problems. In particular, it allows designers to reason about both a system's behaviour and its performance when subjected to a large number of randomly occurring events.

9.7 Exercises

Exercise 9-1

Using the suggestion (Section 9.4.4) made earlier in the chapter, construct an additional process for the event handling system that ensures that the number of missed events is correct. The additional process should be added to the network of processes. You may need to modify the EventData class (Section 9.2.4) to facilitate this.

Exercise 9-2

The accompanying exercise package contains a version of the event handling system, `RunMultiStream`, which allows the creation of 1 to 9 event streams. By modifying the times associated with each event generation stream and also of the processing system explore the performance of the system. What do you conclude?

Exercise 9-3

The process `EventProcessing` has three versions of multiplexer defined within it, two of which are commented out. By choosing each of the options in turn, comment upon the effect that each multiplexer variation has on overall system performance.

Exercise 9-4

A manufacturing process utilises hoppers and a blender. The hoppers are used to hold raw materials and the blender is used to mix the contents from one or more hoppers. The collection of hoppers and the blender is managed by a controller. The hoppers indicate when they are ready to be used. The blender indicates when it is ready and also when mixing is to stop. The hoppers and the blender are clients to the server manager of the controller. The hoppers make a request to the manager to determine when they should stop processing raw materials. The aim of this exercise is to create three different control regimes as follows:

i. The hoppers and the blender indicate they are ready to start but mixing only commences when all three hoppers are ready after which the ready signal from the blender is ready.
ii. As in (i) above but mixing commences as soon as two hoppers and the blender are ready. If three hoppers are ready before the blender is ready then the last hopper is not used and will only be used during the following mixing cycle.

iii. As in (ii) above but mixing commences as soon as just one hopper and the blender are ready. If more than one hopper is ready before the blender becomes ready then these are retained until the following mixing cycle(s).

The accompanying exercises package has definitions for Hopper and Blender processes that utilise the GConsole to enable user interaction. These processes are complete and implement a client style behaviour. The user inputs an 'r' into the input area of the GConsole of the required Hopper or Blender to signify that it is ready. The Hopper or Blender process then outputs a '1' to signal to the Manager process that it is ready. The Manager then implements the required control regime as described above. A Hopper process then sends a '2' signal to the Manager indicating that it is ready to be stopped. The Blender process waits for the user to input an 'f' to indicate that blending can stop. The Blender then sends a '2' to the Manager process. The Manager then completes the client-server interactions. The web site contains scripts to execute each of the above control regimes. There are also outline process definitions for each of the control regimes that need to be completed. Initially, you are advised to produce a process network diagram to enable a better understanding of the interactions and process architecture.