12 Dining Philosophers: A Classic Problem

This problem first formulated by Dijkstra is cited by Hoare in his original paper on Communicating Sequential Processes []. Tantalisingly, Hoare presents the problem and a partial solution leaving it up to the reader to finish the solution. The problem was formulated at a time in the mid-1970's when computer manufacturers were having a great deal of difficulty in building operating systems that were correct and could withstand continued use. Typical problems that had to be overcome were deadlock between different tasks and other tasks being starved of resources; exactly the same problems that the client-server design pattern solves.

The problem has the following statement. Five philosophers spend their lives thinking and eating. They share a common dining room in their college where there is a circular table surrounded by five chairs, each is assigned to one of the philosophers. In the centre of the table there is a large bowl of spaghetti. The table is set with five forks each one assigned to a specific philosopher. On feeling hungry the philosopher enters the room, sits in his own chair and picks up his fork, which is to his left hand. The spaghetti is so tangled that he needs to use the fork to his right hand side as well. When he has finished eating he replaces both forks and leaves the room. The college has provided a butler who ensures that the bowl of spaghetti is always full and can carry out other duties as necessary such as washing-up and guiding philosophers to their own seat.

It is apparent that the critical aspect of this problem is in the management of the forks. If a philosopher is never able to pick up the fork to their right then they will never be able to eat and will thus exhibit starvation or as we have termed it, livelock. Similarly, if all the philosophers enter the room at the same time and each picks up their own left fork none of them will be able to pick up their neighbour's fork to their right and thus deadlock will ensue as none of the philosophers will ever be able to eat.

12.1 Naïve Management

The behaviour of a philosopher is relatively simple and is captured in Listing 12-1. A `Philosopher` can access their own `leftFork` {3}, and their neighbour's as their `rightFork` {4} they can also `enter` {5} into or `exit` {6} from the room. A set of output channels is provided for each Philosopher so they can indicate their intentions. A philosopher is identified by a property `id` {7}. The behaviour of each philosopher will be governed by a `timer` {9}. A method, `action`, has been provided {11-14} that prints the current action of a philosopher and also makes them wait for a specified period. A Philosopher is initially thinking for 1 second {18}, after which they enter the room {19}. They then indicate they are picking up their left fork by means of a signal {21} and similarly for their right fork {23}.

```
01     class Philosopher implements CSProcess {
02
03       def ChannelOutput leftFork
04       def ChannelOutput rightFork
05       def ChannelOutput enter
06       def ChannelOutput exit
07       def int id
08
09       def timer = new CSTimer()
10
11       def void action ( id, type, delay ) {
12         println "${type} : ${id} "
13         timer.sleep(delay)
14       }
15
16       def void run() {
17          while (true) {
18           action (id, "           thinking", 1000 )
19           enter.write(1)
20           println "${id}: entered"
21           leftFork.write(1)
22           println "${id}: got left fork"
23           rightFork.write(1)
24           println "${id}: got right fork"
25           action (id, "           eating", 2000 )
26           leftFork.write(1)
27           println "${id}: put down left"
28           rightFork.write(1)
29           println "${id}: put down right"
30           exit.write(1)
31           println "${id}: exited"
32         }
33       }
34     }
```

Listing 12-1The Behaviour of a Philosopher

They are then eating for 2 seconds {25}, after which they put down their left fork {26}, then their right fork {28} and then they leave the room {30} to resume thinking {18}.

A Fork, Listing 12-2, can either be picked up from the right or the left depending upon which Philosopher has sat down. These are indicated by a signal on the appropriate channel, `left` {37}, or `right` {38}.

```
35     class Fork implements CSProcess {
36
37       def ChannelInput left
38       def ChannelInput right
39
40       def void run () {
41         def fromPhilosopher = [left, right]
42         def forkAlt = new ALT ( fromPhilosopher )
43         while (true) {
44           def i = forkAlt.select()
45           fromPhilosopher[i].read()      //pick up fork i
46           fromPhilosopher[i].read()      //put down fork i
47         }
48       }
49     }
```

Listing 12-2 The Fork Behaviour

An alternative is constructed, `forkAlt` {41, 42}. Once a fork has been picked up by a philosopher it can only be put down by that philosopher, thus all we have to do is process the signal indicating the picking up of the fork {45} and then wait for the signal indicating that it has been put down {46}.

The college has employed a lazy butler who simply notes the entries and exits to the dining room and does little else apart from washing the forks and replenishing the bowl of spaghetti. The latter actions are of no concern. The behaviour of the `LazyButler` is shown in Listing 12-3.

```
50     class LazyButler implements CSProcess {
51
52       def ChannelInputList enters
53       def ChannelInputList exits
54
55       def void run() {
56         def seats = enters.size()
57         def allChans = []
58
59         for ( i in 0 ..< seats ) { allChans << exits[i]  }
60         for ( i in 0 ..< seats ) { allChans << enters[i] }
61
62         def eitherAlt = new ALT ( allChans )
63
64         while (true) {
65           def i = eitherAlt.select()
66           allChans[i].read()
67         }
68       }
69     }
```

Listing 12-3 The Lazy Butler's Behaviour

The channels used to signal the entry and exit from the room are passed to the `LazyButler` as `ChannelInputLists` `enters` {52} and `exits` {53}. The number of `seats` in the dining room can be determined by the size of the list `enters` {56}. A list of all the channels, `allChans` {57} is defined to which each of the elements of the `exits` and `enters` lists are appended {59, 60}. An alternative, `eitherAlt` is defined over `allChans` {62} and as signals are received {65} on any of the channels they are read {66} and ignored by the lazy butler.

The college, believing this to be a sufficient solution, implements it as shown in Listing 12-4.

```
70     def PHILOSOPHERS = 5
71
72     One2OneChannel[] lefts = Channel.createOne2One(PHILOSOPHERS)
73     One2OneChannel[] rights = Channel.createOne2One(PHILOSOPHERS)
74     One2OneChannel[] enters = Channel.createOne2One(PHILOSOPHERS)
75     One2OneChannel[] exits = Channel.createOne2One(PHILOSOPHERS)
76
77     def entersList = new ChannelInputList(enters)
78     def exitsList = new ChannelInputList(exits)
79
80     def butler = new LazyButler ( enters: entersList, exits: exitsList )
81
82     def philosophers = ( 0 ..< PHILOSOPHERS).collect { i ->
83              return new Philosopher ( leftFork: lefts[i].out(),
84                                        rightFork: rights[i].out(),
85                                        enter: enters[i].out(),
86                                        exit: exits[i].out(), id:i ) }
87
88     def forks = ( 0 ..< PHILOSOPHERS).collect { i ->
89                    return new Fork ( left: lefts[i].in(),
90                                       right: rights[(i+1)%PHILOSOPHERS].in() ) }
91
92
93     def processList = philosophers + forks + butler
94
95     new PAR ( processList ).run()
```

Listing 12-4 The College's Lazy Implementation

The number of `PHILOSOPHERS` is defined {70} and then each of the required channel arrays {72-75} and corresponding channel lists {77, 78} are defined. The `butler` and collection of `philosophers` are defined, passing channel parameters as required {82 – 86}. The collection of `forks` is then defined {88-90} noting that the same fork can be accessed as the left fork of the i'th philosopher and the right fork of the i+1'th philosopher {90}, using modulo arithmetic to ensure the subscripts stay in range. Execution of this scheme produces the output shown in Output 12-1.

```
            thinking : 1
            thinking : 2
            thinking : 3
            thinking : 4
            thinking : 0
1: entered
2: entered
3: entered
4: entered
0: entered
2: got left fork
3: got left fork
1: got left fork
4: got left fork
0: got left fork
```

Output 12-1Operation Of The Lazy College

As can be observed, all the philosophers think, then enter the dining room and then they each pick up their left fork after which no further progress is possible. Faced with this situation the college reflects on their operation and decides that the butler has to be more proactive in managing the dining room.

12.2 Proactive Management

The butler is now required to ensure that no more than four of the philosophers are in the room at any one time. This guarantees that at least one of the philosophers will be able to pick the single spare fork on their right hand side. The required behaviour of the butler is shown in Listing 12-5.

The first part of the behaviour up to {109} is identical to that of the `LazyButler` except that a variable seated has been defined {103}, which counts the number of philosophers already sitting. In addition, an extra alternative, `exitAlt` {110} is defined over the exits only. Initially, the butler determines whether there are at least two spare seats in the room {113}, in which case there is `space` for another philosopher to enter and start eating. In this case we can accept an input on any of the channels, `allChans`, managed by the butler. If there is no space then we can only accept inputs from philosophers wishing to exit the room. The alternative to use is determined based on the value of `space` {114}. An enabled input is then `selected` {115} and `read` {116}. It is important to note that `allChans` contains the `exits` channels first so that we can read exit signals from `allChans`; regardless of which alternative is used. We can determine whether or not this instance results from a philosopher exiting or entering the room by testing the index of the read channel, `i`, against the number of `seats` {118} and updating the number of philosophers seated accordingly.

The college is very relieved to discover that this simple change of butler behaviour is sufficient to remedy the situation provided they replace the invocation of the `LazyButler` by the `Butler` on line {80} of Listing 12-4.

```
class Butler implements CSProcess {

  def ChannelInputList enters
  def ChannelInputList exits

  def void run() {
    def seats = enters.size()
    def seated = 0

    def allChans = []
    for ( i in 0 ..< seats ) { allChans << exits[i]  }
    for ( i in 0 ..< seats ) { allChans << enters[i] }

    def eitherAlt = new ALT ( allChans )
    def exitAlt = new ALT ( exits )

    while (true) {
      def space = seated < ( seats - 1 )
      def usedAlt = space ? eitherAlt : exitAlt
      def i = usedAlt.select()
      allChans[i].read()
      def exiting = i < seats
      seated = exiting ? seated - 1 : seated + 1
    }
  }
}
```

Listing 12-5 The Modified Butler Behaviour

Output from the modified butler behaviour is shown in Output 12-2. It can be seen that all bar Philosopher 0 enter the room and that means that Philosopher 1 and 2 can eat at the same time. When Philosopher 1 finishes eating and leaves the room to resume thinking, Philosopher 3 is now able to eat. Further analysis shows that there are two Philosophers eating most of the time as should be expected. Thinking appears to be a solitary activity!

```
            thinking : 0
            thinking : 1
            thinking : 2
            thinking : 3
            thinking : 4
1: entered
2: entered
3: entered
4: entered
1: got left fork
2: got left fork
3: got left fork
4: got left fork
1: got right fork
            eating : 1
2: got right fork
            eating : 2
1: put down left
1: put down right
0: entered
0: got left fork
1: exited
            thinking : 1
2: put down left
3: got right fork
            eating : 3
```

Output 12-2 Modified Behaviour

12.3 A More Sophisticated Canteen

In an effort to provide a better service the college decides that, rather than having a single dinning room with its somewhat limited eating facilities, it is going to invest in a canteen style food facility. Philosophers will be allowed to enter the canteen, go to a serving hatch, pick up their food, in the form of a chicken, without having to wait, in fact waiting will not be allowed and then go into the canteen to find a place to sit. The college authorities guarantee that there will be sufficient places for every one to sit and that nothing else can go wrong. They are so confident that they allow any number of philosophers to enter the canteen. To this end they have decided that a visual display will be provided showing the state of the kitchen, in which the chef cooks the chickens, the state at the serving hatch and they have also installed monitoring devices that shows the action each philosopher is currently undertaking.

The chef is capable of cooking four chickens at a time but it does take time for them to cook and also to take them to the serving hatch. This is shown in Listing 12-6.

```
122    class Chef implements CSProcess {

123      def ChannelOutput supply
124      def ChannelOutput toConsole

125      def void run () {
126        def tim = new CSTimer()
127        def CHICKENS = 4
128        toConsole.write( "Starting ... \n")

129        while(true){
130          toConsole.write( "Cooking ... \n")
131          tim.after (tim.read () + 2000)
132          toConsole.write( "${CHICKENS} chickens ready ... \n")
133          supply.write (CHICKENS)
134          toConsole.write( "Taking chickens to Canteen ... \n")
135          supply.write (0)
136        }
137      }
138
139    }
```

Listing 12-6 The Chef's behaviour

The `supply` channel {123} is used to indicate to the canteen how many chickens are about to arrive. The `toConsole` channel {124} is used to write information on the display. It takes 2 seconds to cook the chickens {131} with appropriate messages output to the console. The number of chickens is sent on the `supply` channel to the canteen {133}. The write to the supply channel {135} is used to represent the point at which the chickens have been transferred to the serving hatch as can be seen in Listing 12-7.

The canteen receives requests for a chicken from a philosopher on the `service` channel {141} and notification of its availability is given on the `deliver` channel {142}. The `Chef` process uses the `supply` channel to indicate that chickens are ready for serving {143}. The `toConsole` channel is used to display the current availability of chickens on the display {144}. The canteen alternates over the `supply` and `service` channels {146}. A `timer` {149} is required reflect the time it takes to set down the chickens by the `Chef`. The enabled alternative is `select`ed using the `fair` option {153}.

In the case of `SUPPLY`, when more chickens become available, the `value` is read from `supply` {155} and a message written to the console {156}. A delay of 3 seconds is created {157} representing the time taken to transfer chickens from the kitchen to the canteen. After this the number of chickens available is incremented {158} by `value`. The canteen console is updated {159} and the signal written by the `Chef` {135} is read {160} and this permits the `Chef` to return to the `Kitchen` to cook more chickens.

```
140    class InstantCanteen implements CSProcess {

141      def ChannelInput service
142      def ChannelOutput deliver
143      def ChannelInput supply
144      def ChannelOutput toConsole

145      def void run () {

146        def canteenAlt = new ALT ([supply, service])
147        def SUPPLY = 0
148        def SERVICE = 1
149        def timer = new CSTimer()
150        def chickens = 0
151        toConsole.write( "Canteen : starting ... \n")

152        while (true) {
153          switch (canteenAlt.fairSelect ()) {

154            case SUPPLY:
155              def value = supply.read()
156              toConsole.write( "Chickens on the way ...\n")
157              timer.after (timer.read() + 3000)
158              chickens = chickens + value
159              toConsole.write( "${chickens} chickens now available ...\n")
160              supply.read()
161            break

162            case SERVICE:
163              def id = service.read()
164              if ( chickens > 0 ) {
165                chickens = chickens - 1
166                toConsole.write ("chicken ready for Philosoper ${id} ...
                                                      chickens left \n")
167                deliver.write(1)
168              }
169              else {
170                toConsole.write( " NO chickens left ... \n")
171                deliver.write(0)
172              }
173            break
174          }
175        }
176      }
177    }
```

Listing 12-7 The Canteen Behaviour

When a philosopher requires `SERVICE`, their `id` is `read` from the `service` channel {163}. The Canteen at this point recognises that there may be no chickens available but is sure that this will not happen. Thus a test is undertaken on the number of available `chickens` {164} and if there is a chicken available the number of `chickens` is decremented {165} and a message to that effect output {166}. The philosopher is informed by the writing of a 1 on the `deliver` channel {167}. If no chickens are available, a message is displayed {170} and a zero is written to the deliver channel {171}.

The behaviour of the Philosophers is now somewhat different; they still think and eat forever, in rotation. However the philosophers are somewhat sanguine about the College authorities' capabilities and use a behaviour in which they try to cover every eventuality as shown in Listing 12-8. A philosopher has an `id` {179}, a channel upon which a `service` request is made {180} and one upon which a chicken `delivery` is made {181} plus a channel to write messages on a console {182}. A `timer` {184} is required to time the philosopher's actions and an initial message is written `toConsole` {185}.

```
178    class PhilosopherBehaviour implements CSProcess {

179      def int id  = -1
180      def ChannelOutput service
181      def ChannelInput deliver
182      def ChannelOutput toConsole

183      def void run() {
184        def timer = new CSTimer()
185        toConsole.write( "Starting ... \n")

186        while (true) {
187          toConsole.write( "Thinking ... \n")
188          if (id > 0) {
189            timer.sleep (3000)
190          }
191          else {
192            timer.sleep (100)
193          }

194          toConsole.write( "Need a chicken ...\n")
195          service.write(id)
196          def gotOne = deliver.read()

197          if ( gotOne > 0 ) {
198            toConsole.write( "Eating ... \n")
199            timer.sleep (2000)
200            toConsole.write( "Brrrp ... \n")
201          }
202          else {
203            toConsole.write( "                   Oh dear No chickens left \n")
204          }
205        }
206      }
207    }
```

Listing 12-8 The Philosopher Behaviour

Initially, a philosopher thinks for 3 seconds {189}, unless they are philosopher 0 who only thinks for 0.1 seconds {192}. At this point the behaviour is common and starts by indicating on the console that the philosopher needs a chicken {194}, and is followed by a signal request on the service channel with the philosopher's id {195}. At this point we note that the philosopher is behaving like a client and thus immediately follows the service request with the input of the chicken on the `deliver` channel {196} containing the server response from the canteen. The philosopher now tests the value of `gotOne` {197} to see if they have been given a chicken. If this is the case, then a message is output and the philosopher takes 2 seconds to eat the chicken, after which he burps {200}. If no chicken is available a sad message appears {203}.

The above processes are formed into a further process each with a `GEclipseConsole`, upon which console messages can be displayed. The script that invokes the system is shown in Listing 12-9. The channels that implement the `service` and `deliver` connections between the philosophers and the canteen are shared {208, 209}, `Any2One` and `One2Any` channels respectively, enabling any of the philosophers to access the canteen. A list of five philosophers is then created with each connected to `service` and `deliver` {212, 216}. The other processes, `InstantServery` comprising the canteen and its console and the `Kitchen` comprising the `Chef` and its console are added to `processList` {218-222}. The processes are then run. This can be observed by running the script `InstantCollege` in `c14.examples.canteen`. Needless to say we observe that some philosophers do not get a chicken and more importantly miss their turn!

```
Any2OneChannel service = Channel.createAny2One ()
One2AnyChannel deliver = Channel.createOne2Any ()
One2OneChannel supply = Channel.createOne2One ()

def philosopherList = (0 .. 4).collect{
         i -> return new Philosopher( philosopherId: i,
                                      service: service.out(),
                                      deliver: deliver.in())
         }

def processList = [ new InstantServery ( service:service.in(),
                                         deliver:deliver.out(),
                                         supply:supply.in()),
                    new Kitchen (supply:supply.out())
                  ]

processList = processList +  philosopherList

new PAR ( processList ).run()
```

Listing 12-9 The Instant Canteen Script

It is obvious that the behaviour of the canteen is at fault as it did not stop philosophers making requests for service when there were no chickens available. The revised behaviour is shown in Listing 12-10, which has been augmented by the use of pre-conditions.

The precondition array is initialised {234} so that chickens can always be supplied from the kitchen. Initially, there are no chickens available so the `service` precondition is `false`. At the start of the process' main loop the state of the service precondition is re-evaluated {241}. If no chickens are available a message to that effect is displayed {243}. Now, of course, we enter each case in the `switch` associated with the enabled alternative knowing the precise state of the canteen and thus the coding is much simpler. In particular, we only permit `service` requests when we are assured that chickens are available {254-258}.

This version of the system can be executed using the script `QueuingCollege` and another version that shows clock ticks in the canteen console is also available, `ClockedQueuingCollege`. It can be observed from an execution of the system, which allows numbers other than five philosophers, that every philosopher gets a chicken whenever they are hungry, they may have to wait.

```
class QueuingCanteen implements CSProcess {

  def ChannelInput service
  def ChannelOutput deliver
  def ChannelInput supply
  def ChannelOutput toConsole

  def void run () {
    def canteenAlt = new ALT ([supply, service])
    def boolean [] precondition = [true, false ]
    def SUPPLY = 0
    def SERVICE = 1
    def tim = new CSTimer()
    def chickens = 0
    toConsole.write ("Canteen : starting ... \n")

    while (true) {
      precondition[SERVICE] = (chickens > 0)

      if (chickens == 0 ){
        toConsole.write ("Waiting for chickens ...\n")
      }

      switch (canteenAlt.fairSelect (precondition)) {

        case SUPPLY:
          def value = supply.read()
          toConsole.write ("Chickens on the way ...\n")
          tim.after (tim.read() + 3000)
          chickens = chickens + value
          toConsole.write ("${chickens} chickens now available ...\n")
          supply.read()
        Break

        case SERVICE:
          def id = service.read()
          chickens = chickens - 1
          toConsole.write ("chicken ready for Philosoper ${id} ...
                                           ${chickens} chickens left \n")
          deliver.write(1)
        break
      }
    }
  }
}
```

Listing 12-10 The Revised Canteen With a Queue

12.4 Summary

This chapter has presented solutions to the classical dining philosophers' problem using two different formulations. The second solution, using a canteen is also an instance of the client-server design pattern with the canteen acting as a pure server and the chef and philosophers acting as pure clients. This perhaps demonstrates that even though the coding in both cases followed the client–server pattern it was still possible to create an erroneous solution. The client-server design pattern is not a panacea for all occasions; it has to be applied sensibly and with understanding. Even if the communication patterns are correct it is still possible to create incorrect systems if insufficient thought is given to the problem solution.