16 Anonymous Network Channels: A Print Server

A print server is probably the simplest service used by users of a networked service. It provides a means whereby a user can send a file for printing using a printer shared among the users of the network. In this implementation a print service will be constructed that accepts print lines, a line at a time, from a user. The print service will accept print lines from a number of users, in parallel, up to some limit set when the print server is installed. Once the user has sent all the lines of text to be printed; the print server will then output those lines in a single printed output. The printed output will be preceded by a job number that can be recognised by the user of the service. The user of the service will be informed both when their job has been accepted and when it has completed. The user will be unaware that the print service is dealing with other user requests. The print service should run in the background and always be ready to accept requests from a user, that is, the user processes should start asynchronously with the print service process. The order in which the respective processes start should have no bearing on the operation of the system.

From the foregoing it is obvious that users need to request that their lines of output are sent to the print service and subsequently on completion of their output the user needs to indicate that the lines of text can be printed. To this end the print service provides two named channels by which the user can request and subsequently release their use of the print service.

In addition, if the print service is going to manage print operations from more than one user in parallel then some means of telling the user which of the services to use will be required. The user also needs to be able to send lines to be printed to the print service. These connections will change with each print job and thus the corresponding network channels will be created dynamically and anonymously.

The architecture of the system is shown in Figure 16-1. The `PrintSpooler` process provides the print service using two named `Net2One` channels called request and release. The network connections are indicated by the dashed lines, one for each channel. Each `PrintUser` process can dynamically connect to the request and release channels by defining them as `Any2Net` when their node is created. In order to avoid multiple communications on the request and release channels only one communication will be permitted on each channel for each print job.

The diagram shows the state when the `PrintSpooler` is willing to accept print lines from up to two `PrintUser` processes in parallel. These have been given names for clarity but in reality are anonymous. The `useChannel` is used by `PrintSpooler` to tell the `PrintUser` the location of the `printChannel` it is to use to send the lines to the `PrintSpooler`. The `printChannel` is used to send the lines to be printed to the `PrintSpooler`.

Pre-publication version

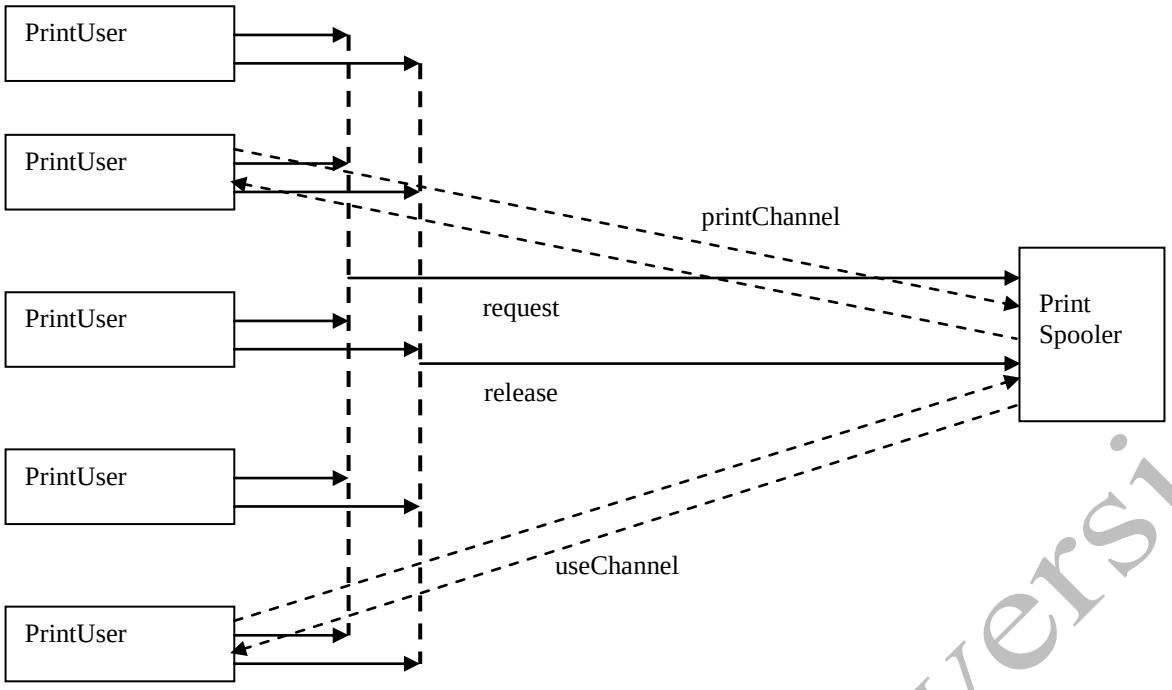

Figure 16-1 The Print Service Architecture

16.1 Print Spooler Data Objects

Two data objects are required, which both are used to transfer information from `PrintUser` processes to the `PrintSpooler` process. The first, `PrintJob`, shown in Listing 16-1 is used to make an initial request for service. It comprises two properties, the identity of the user {2} and the net channel location to be used by `PrintSpooler` as the useChannel {3}. The manner of its creation and its use will be described later.

```
01    class PrintJob  implements Serializable {

02      def int userId
03      def NetChannelLocation useLocation
04    }
```

Listing 16-1 The PrintJob Data Object

The other data object, `PrintLine`, shown in Listing 16-2, is used to transfer lines to be printed from a `PrintUser` process to the `PrintSpooler` process. It is written by the `PrintUser` to a `printChannel`. The property `printKey` {6} indicates which, of the possibly several internal concurrent spoolers within `PrintSpooler`, this line of text is intended. The `String line` {7} is the text to be added to the output.

```
05    class Printline implements Serializable {

06      def int printKey
07      def String line
08    }
```

Listing 16-2 The PrintLine Data Object

16.2 The PrintUser Process

Listing 16-3 shows the coding of the `PrintUser` process. This process has three properties; `printerRequest` {10}, the network channel used to make requests to the `PrintSpooler`, `printerRelease` {11}, the network channel used to release the `PrintSpooler` at the end of printing and the identity of the user, `userId`{12}.

Pre-publication version

```
09    class PrintUser implements CSProcess {

10      def ChannelOutput printerRequest
11      def ChannelOutput printerRelease
12      def int userId

13      void run() {
14        def timer = new CSTimer()
15        def printList = [ "line 1 for user " + userId,
16                          "line 2 for user " + userId,
17                          "last line for user " + userId
18                        ]

19        def useChannel = NetChannelEnd.createNet2One()
20        printerRequest.write(new PrintJob ( userId: userId,
21                                  useLocation: useChannel.getChannelLocation() ) )

22        def printChannelLocation = useChannel.read()
23        def useKey = useChannel.read()
24        println "Print for user ${userId} accepted using Spooler $useKey"

25        def printerChannel = NetChannelEnd.createOne2Net ( printChannelLocation)
26        printList.each { printerChannel.write (new Printline ( printKey: useKey,
27                                                               line: it) )
28                         timer.sleep(10) }

29        printerRelease.write ( useKey )
30        println "Print for user ${userId} completed"
31      }
32    }
```

Listing 16-3 The PrintUser Process Definition

A `timer` {14} is defined, which is used to create a short delay between lines sent to `PrintSpooler`. The `List printList` {15-18} holds the lines of text that are to be printed. Each user prints the same output, only differentiated by their `userId`.

`NetChannelEnd` is a network channel factory that can be used to create network channels and in this case is used to create an anonymous `Net2One` or `NetInputChannel` {19}. This network channel is assigned to variable `useChannel`. A `PrintJob` object is constructed from `userId` and the location of `useChannel` {18-21}. The location of a network channel is obtained by calling the method `getChannelLocation()` {21}, which returns the IP address, port and unique channel number of the channel. The `PrintJob` object is then written to the `printRequest` network channel. This `write` {20} may be delayed if the `PrintSpooler` is already dealing with the maximum number of print requests, but the user will be unaware of this, in the sense that they have to undertake any additional processing. This is also the first part of a client behaviour with its corresponding server response being the `read` of a `printerChannelLocation` on the `useChannel` {22}. The `useChannel` is also used to `read useKey` from the `PrintSpooler` {23}. The `useKey` is the means by which the `PrintUser` process identifies which of the concurrent spoolers maintained by `PrintSpooler` it is to use. A message is then printed indicating that the request has been accepted {24}.

The process now creates the channel `printerChannel` on which it is to send lines to the `PrintSpooler`. The location of this channel has been `read` as `printerChannelLocation`. A network channel can be created from this location by a call to the `NetChannelEnd` factory to create a `One2Net` channel with the location specified in `printerChannelLocation` {25}. This channel is then used to write `each` of the `printList` elements to `printerChannel` using a constructed `PrintLine` object for each element {26-27} This is then followed by a short delay of 10 milliseconds {28}. The Groovy operator `each` iterates through the elements of a `List` and the associated `closure` can refer to the specific element using the `it` keyword.

Once all the elements of `PrintList` have been written to the `PrintSpooler` it can be released and this is simply achieved by writing the `useKey` to the `printerRelease` channel {29}, after which a message can be printed.

Pre-publication version

16.3 The PrintSpooler Process

Listings 16-4 and 16-5 show the coding of the `PrintSpooler` process. Properties `printerRequest` {35} and `printerRelease` {36} are the named request and release networked channels `PrintUsers` use to indicate their desire to access the `PrintSpooler`. The `PrintSpooler` will create two concurrent `spoolers` by default {37} but this can be changed when the process is invoked.

```
33    class PrintSpooler implements CSProcess {
34
35      def ChannelInput printerRequest
36      def ChannelInput printerRelease
37      def int spoolers = 2
38
39      void run() {
40        def spooling = 0
41        def spoolChannels = []
42        def spoolChannelLocations = [:]
43        def unusedSpoolers = []
44        def printMap = [:]
45        def jobMap = [:]

46        def preCon = new boolean[spoolers + 2 ]
47
48        0.upto(spoolers - 1){i ->
49                             def c = NetChannelEnd.createNet2One()
50                             spoolChannels << c
51                             spoolChannelLocations.put(i, c.getChannelLocation())
52                             unusedSpoolers << i
53                             preCon[i+2] = false
54                             }
55        def altChans = [ printerRelease, printerRequest ]
56        altChans = altChans + spoolChannels
57        preCon[0] = true
58        def psAlt = new ALT ( altChans )
```

Listing 16-4 PrintSpooler Initialisation

The variable `spooling` {40} is used to count how many concurrent users are sending output to the `PrintSpooler` and initially this is none. `SpoolChannels` will hold a `List` of the channels that will become the printChannels of Figure 16-1. These will be created at the outset rather than recreating them on each occasion individually for each print request. The `Map spoolChannelLocations` {42} will hold the `NetChannelLocation` of each spool channel used by each spooler; its key is the index of the spooler. The `List unusedSpoolers` {43} holds the index of the spoolers that are currently not being used. The `Map printMap` {44} is used to hold the lines for each concurrent print request and its key is the associated spooler index. The `Map jobMap` {45} is used to maintain the connection between spooler index and user requesting the print job.

The prcoess is going to use an alternative to determine which input channels it will receive input from. These are going to be further managed using a precondition array and this is defined as `preCon` {46}. There is an input channel for each spooler plus the two named request and release channels, giving the number of elements in the array.

Lines {48-54} initialise these data structures as follows by iterating over the number of spoolers {48}, with `i` indexed from 0. A `NetChannelInput c` is created as an anonymous `NetChannelEnd` {49} and this will be subsequently used as the `printChannel`. It is then appended to `spoolChannels` {50}. Its net channel location is then `put` in the `i`'th element of `spoolChannelLocations` {51}. The value of `i` is then appended to the `List` of `unusedSpoolers` {52}. Finally, `preCon[i+2]` {53} is set `false` indicating that the processes cannot accept input on any of its spool channels. The `List altChans` {55} is initialised with the `printerRelease` and `printerRequest` channels to which is added the `List` of `spoolChannels` {56}. The process is always willing to accept an input on its release channel and so `preCon[0]` is set `true` {57}. The reason for the offset of `i+2` {53} is to take account of the fact that the printer release and request channels appear first in the alternative `psAlt` {58}.

Pre-publication version

```
59          while (true) {
60            preCon[1] = (spooling < spoolers)
61            def index = psAlt.select(preCon)
62            switch (index) {
63              case 0: // release
64                def usedKey = printerRelease.read()
65                unusedSpoolers.add(usedKey)
66                preCon[usedKey + 2] = false
67                spooling = spooling - 1
68                def lines = printMap.get(usedKey)
69                print "\n\nOutputFor User ${jobMap.get(usedKey)}\n"
70                println "Produced using spooler $usedKey \n\n"
71                lines.each{ println "${it}" }
72                println "\n\n================================\n\n"
73                printMap.remove(usedKey)
74                jobMap.remove(usedKey)
75                break

76              case 1:  // request
77                def job = printerRequest.read()
78                def useChannelLocation = job.useLocation
79                def userId = job.userId
80                def useChannel = NetChannelEnd.createOne2Net(useChannelLocation)
81                spooling = spooling + 1
82                def useKey = unusedSpoolers.pop()
83                preCon[useKey+2] = true
84                printMap[useKey] = []
85                jobMap[useKey] = userId
86                useChannel.write(spoolChannelLocations.get(useKey) )
87                useChannel.write( useKey )
88                break

89              default :  // printline being received from a user
90                def pLine = spoolChannels[ index - 2].read()
91                printMap[pLine.printKey] << pLine.line
92            }
93          }
94        }
95      }
```

Listing 16-5 The PrintSpooler Process Loop

At the start of each iteration a test is made to determine the state of `preCon[1]` {60} which ensures that a request for service will only be accepted if at least one of the available spoolers is free. The `index` of the enabled alternative is `selected` {61} and used to determine which `case` is processed.

`Case 0` represents an input on the release channel, which means that the lines can be printed and the assocoiated spooler released for another user. The input `usedKey` from the `printerRelease` channel identifies the spooler allocated to the user {64}. This spooler can then be added to the List of `unusedSpoolers` {65}. The process is now unwilling to accept any more inputs from the user and thus sets the corresponding `preCon` element `false` {66}. Similalrly, the number of `spooling` spoolers can be decremented {67}. The `List lines` comprises the `Map` entry for `usedKey` {68} obtained using the `Map get()` method. The printed output banner lines can now be printed {69-70}, after which the `lines` can themselves be printed {71} followed by a terminating banner {72}. The `Map` entry for `usedKey` can now be `removed` {73}. Similarly the entry relating job and user from `jobMap` can be `removed` {74}. This code sequence recovers the printing resources, prints the lines and ensures that the associated data structure have been udated accordingly.

`Case 1` pertains to a request for printing by a user process and will only be accepted if at least one of the spoolers is available. The print `job` details are read from the `printerRequest` channel {77} and the `PrintJob` properties are extracted into variables `useChannelLocation` {78} and `userId` {79}. The channel by which the `PrintSpooler` process sends data to the `PrintUser` process, `useChannel`, is created by taking the value of `useChannelLocation` {80} as a parameter to a call of `NetChannelEnd.CreateOne2Net()`. In the `PrintSpooler` process we are creating the output end of the channel to be connected to the input end that was created in the `PrintUser` process {19}. The number of spoolers that are `spooling` can be incremented {81} and the index of an unused spooler can be `pop`'ed

Pre-Publication version

from the List of `unusedSpoolers` {82} and assigned to `useKey`. The pre condition element of the array `preCon` associated with this spooler can be set true because the process is now willing to accept inputs on the related spool channel {83}. A `Map` entry that uses `useKey` as its `key` can be initialised to an empty `List` {84}. An entry can be placed in `jobMap` that relates `useKey` to the `userId` of the job being processed by this spooler {85}.

The `PrintSpooler` process acts as a server to the `PrintUser` processes and a request for service {18-21} expects a response in finite time. The generated response is the location of the spool channel that the `PrintUser` process is to use for writing lines of text to the `PrintSpooler` {86}. Secondly, the value of `useKey` which is used by the `PrintUser` process to identify which spooler is being used to form the lines of text to be output {87}.

The `default` case {89} is perhaps the simplest and simply reflects the input of a line of text from a `PrintUser` process to the `PrintSpooler`. The line of text is `read` into `pLine` from the element of `spoolChannels` indexed by `index - 2` {90}. The value of `index` is obtained from the `select` method call on the alternative `psAlt` {61}. However, `psAlt` precedes that `spoolChannels` with the `printerRelease` and `printerRequest` channels and thus it is necessary to subtract 2 from the `index` value to access the correct element of `spoolChannels`. The variable `pLine` is of type `PrintLine` and the properties `printKey` and `line` are used to add the `line` to the `printMap` entry for the `printKey` {91}. In this manner each of the active `PrintUser` processes can send lines to the `PrintSpooler` adding lines to the `List` contained in the `printMap`. All that is required by a `PrintUser` process is the key of the `printMap` used to add lines to the List of lines. Thus, each spooler is in fact just represented by an entry in the `printMap` structure.

16.4 Summary

This chapter has introduced the concept of anonymous network channels and demonstrated how they can be used, in conjunction with two named channels, to provide a flexible network based service. In the next chapter we shall show how this concept can be extended to permit the transfer of processes from one node to another enabling the transferred process to access data on the original node.

Pre-publication version