Concurrent Programming

Gavin Lowe and Bernard Sufrin

Hilary Term 2016-17

9: Clients and Servers

Reading: Andrews, Section 7.3.

Introduction

A common pattern in concurrent systems is that of clients and servers. A *server* is a process that repeatedly handles requests from *client*s.

Why study client/server patterns in the context of a shared-address-space implementation?

- ⋆ The communication patterns are independent of the means of communication, and so we can use the *same* patterns when we have inter-address-space and cross-host means of communication.
- ⋆ Implementation of reliable inter-host communication can be treated (to some extent) orthogonally to process architectures.
- ⋆ This is a pattern that can be used in the implementation of microkernel-based operating systems.

Requests and responses

⋆ A typical pattern is that the server will receive some request, say of type `Request`, on some channel `req`. This request channel will be shared by all clients – in CSO a `ManyOne` or `N2N` channel.

⋆ The server will handle the request, and normally send back a response of some type `Response`. Since the reply has to go to the correct client, it is best for the server to have a different reply channel for each client.

⋆ There are several ways of achieving this, broadly

- Client-server "session"[1] is stateless: include a `![Response]` in each request
- Conversational continuity is needed:
  (client allocated channels) include a `?[Request]` and `![Response]` in *opening* request
  (server allocated channels) include a `!(![Request],?[Response])` in *opening* request

The type of requests

Typically, the server will be able to handle several different types of request. The natural way to do this in Scala is using *case classes*[2]:

In the case of "stateless" encounters with the server each request includes a response port

```
abstract class Request
case class ReqKind1 (reply: ![Response], param1 : Type1 , ····) extends Request
case class ReqKind2 (reply: ![Response], param2 : Type2 , ····) extends Request
···
```

When clients have "sessions" with servers, they can be opened by clients sending request and response ports it has set up:

```
case class Open(reply: ![Response], request: ?[Request], ···)

abstract class Request
case class ReqKind1 (param1 : Type1 , ····) extends Request
case class ReqKind2 (param2 : Type2 , ····) extends Request
···
```

The type `Response` might also be defined using case classes.

[2] See Chapter 7 of *Scala by Example*

Servicing the request – stateless servers

⋆ The server can identify and act on the kind of a request using the read-then-apply-function notation (or a **match**):

⋆ Stateless servers repeatedly read and decode requests – in effect:

```
def server(request: ?[Request]) = proc
{
      ··· server initialization ···
      repeat {
        request ?
           {  case ReqKind1 (reply, bv1 ) ⇒ { ··· ; reply!response1 ; ··· }
              case ReqKind2 (reply, bv2 ) ⇒ { ··· ; reply!response2 ; ··· }
              ···
           }
      }
      ··· server termination ···
 }
```

⋆ Notice that any client can close its (output) end of the req channel.

The effect of this depends on the kind of channel

Clients of Stateless Servers

⋆ Clients of stateless servers can allocate a single channel for all their replies

```
def client(request: ![Request]) = proc
{ ··· initialization
  val reply = OneOne[Response]
  while (not finished)
  {
    request!ReqKind(reply, ···)
    ···
    ··· reply?() ··· // await then use the reply
  }
  ··· finalization
  request·closeOut
}
```

⋆ Avoiding the client holding up the server by being too slow to read the response:

- Client-side: buffer the reply channel in the client:

  ```
  val reply = OneOneBuf[Response](size)
  ```

- Server-side: reply "offline"

  ```
  ···
  case ReqKind2 (reply, bv2 ) ⇒ { ··· ; fork(proc { reply!response2 ; ···}) }
  ```

System Structures for Stateless Servers

⋆ N clients involved in a system that will need a server only until all the clients terminate:

```
val request = N2N[Request](N, 1)
val System  = Server(request) || || (for (i←0 until N) yield client(request))
...
```

⋆ Dynamic creation of clients: server present until system as a whole terminates

```
val request  = ManyOne[Request]
val terminate = ManyOne[Unit]
// Start the server in the background
Server(request)·fork
// Start client(s) in the background
··· client(request)·fork ···
// wait for some entity to call for termination
terminate ? { () ⇒ request·close }
```

Servicing the request – sessional servers

⋆ Sessional servers use request/reply channels established when the session is opened.

```
def session(reply: ![Reply], request: ?[Request]) = proc
{ // session initializaion ...
  repeat {
     request ? {  case ReqKind1 (bv1 ) ⇒ { ··· ; reply!response1 ; ··· }
                  case ReqKind2 (bv2 ) ⇒ { ··· ; reply!response2 ; ··· }
                  ...
               }
  }
  // session termination ...
}
```

⋆ A session can be forked to serve each new client

```
def server(open: ?[Open]) = proc
{  // server initialization ...
   repeat {
     open ? { case Open(reply, request) ⇒ fork(session(reply, request)) } }
   }
   // server termination ...
}
```

(It is usually appropriate to limit the number of simultaneous sessions)

⋆ Clients of sessional servers

```
def client(server: ![Open]) = proc
{ val request: Chan[Request]  = ···
  val reply:   Chan[Reply]    = ···
  server!Open(reply, request)
  ··· initialization
  while (not finished)
  {
    request!ReqKind(···)
    ···
    ··· reply?() ··· // await then use the reply
  }
  ··· finalization
  request·closeOut
}
```

Multiple subclasses versus alternation

⋆ So far we have had a single request channel `request`, passing data of type `Request` which was split into several subtypes:

```
request ? {
  case ReqKind1 (bv1 ) ⇒ { ··· }
  case ReqKind2 (bv2 ) ⇒ { ··· }
  ...
}
```

⋆ An alternative is to use a different channel for each subtype, and for the server to use alternation to choose between them (usually in a serve loop)

```
serve( req1 =?⇒ { bv1 ⇒ ··· }
     | req2 =?⇒ { bv2 ⇒ ··· }
     | ···
     )
```

⋆ This is particularly useful when some requests can only be serviced in certain states.

Problems with sessional servers

⋆ Suppose many clients open sessions with servers, but fail to close them. This could either be:

- ◦ Deliberate, as the result of a denial-of-service attack (more likely for a net-accessible service); or
- ◦ Accidental, as the result of bugs in client programs.

⋆ When there is a bound on the number of session server processes, eventually new clients won't be served.

⋆ When there is no such bound more and more system resources will be taken up by those session server processes.

⋆ Garbage collection won't fix this problem, but timeouts might.

Using timeouts

⋆ If a session has been waiting to communicate with a client for more than a reasonable time, then it should timeout and terminate.

```
def session(reply: ![Reply], request: ?[Request]) = proc
{ // session initializaion ...
  def deliver(response: Reply) = {
     if (!reply·writeBefore(replyDeadline))(response) { stop }
  }
  serve (
     request =?⇒
             {  case ReqKind_1 (bv_1 ) ⇒ { ··· ; deliver(response_1 ); ··· }
                case ReqKind_2 (bv_2 ) ⇒ { ··· ; deliver(response_2 ); ··· }
                ···
             }
  |  after(requestDeadline) ⟹ stop
  )
  // session termination ...
}
```

⋆ This works better in settings where it is easy to estimate an upper bound on time between requests from a client, and on time for replies to be collected by the client.

Descriptors, Cookies, Persistent Sessions

- ⋆ Server generates a new descriptor/cookie per new session opening
- ⋆ Server keeps a map from cookies to session data
- ⋆ Requests are handled by a pool of request-handling workers
- ⋆ Client opens session with server: receives back a descriptor/cookie
- ⋆ Client presents descriptor/cookie with each request
- ⋆ Server uses a free request-handling worker to handle each request in the context of its current session data

```
case class Open(credential: Credential; reply: ![Descriptor])
case class Request_i(descriptor: Descriptor, reply: Reply, ···) extends Request
```

- ⋆ If a session "times out" then its data can be discarded, or stored persistently on behalf of a client if "reopening" sessions is permitted.

⋆ This form of server shares address space with handlers, and trusts the descriptors provided by clients. It only inputs a request when a handler is ready to process it

⋆ Handlers repeatedly read requests from a shared channel.

```
...
def server(open: ?[Open], request: ?[Request], toHandler: ![(SessionData,Request)])=
proc {
    val sessionMap: Map[Descriptor, SessionData] =
    ··· initialization ···
    serve (
       open =?⇒ {
         case Open(cred, reply) ⇒
              if (iBelieve(cred)) {
                 val descriptor = generateNewDescriptor
                 val session    = new SessionData(descriptor)
                 sessionMap(descriptor) = session
                 if (!reply·writeBefore(timeout)(descriptor)) sessionmap -= descriptor
              }
         }
       | toHandler =!⇒ {                        // a handler is ready
           val req = request?()                 // read a request
           (sessionMap(req·descriptor), req)    // forward it with session-data
         }
     )
     ...
```

⋆ Handler

- owns mutable session-data object while handling a request
- signals readiness to do work by reading

```
...
def handler(fromServer: ?[(SessionData,Request)]) = proc {
 serve (
   fromServer =?⇒ {
     case (sessiondata, request) ⇒
       // service request & send the reply, updating session data if necessary
       ...
   }
 )
}
...
```

⋆ During initialization the server is "primed" with an appropriate number of running handlers

```
val open      = ManyOne[Open]
val request   = ManyOne[Request]
val toHandler = N2N[(SessionData,Request)](writers=1, readers=HandlerCount)
for (i←0 until HandlerCount) fork(handler(toHandler))
...
fork(server(open, request, toHandler))
...
```

Case Study: Memory Allocation Server

⋆ Requests and Replies

```
abstract class Reply
case class  Acquired(pages: Set[Int]) extends Reply
case object Released                  extends Reply

abstract class Request
case class Acquire(reply: ![Reply], count: Int)      extends Request
case class Release(reply: ![Reply], pages: Set[Int]) extends Request
```

⋆ Server strategy is to queue unsatisfiable requests in increasing order of size

```
def server(admin: PageAdmin, request: ?[Request]) = proc
{ val orderByCount = new scala.math.Ordering[Acquire]
      { def compare(l: Acquire, r: Acquire): Int = r.count - l.count }
  val waiting = scala.collection.mutable.PriorityQueue.empty[Acquire](orderByCount)

   serve (
     request =?⇒
           { case req @ Acquire(reply, count) ⇒
                if (admin.hasAvailable(count))
                   reply!Acquired(admin.request(count))
                else
                   waiting.enqueue(req)
                ()

             case Release(reply, pages) ⇒
                admin.release(pages)
                while (!waiting.isEmpty && admin.hasAvailable(waiting.head.count))
                      waiting.head.reply!Acquired(admin.request(waiting.dequeue.count))
                reply!Released
                ()
           }
   )
}
```

Encapsulating Servers

⋆ In our earlier examples of client/server interactions, the client code has embedded within it the protocol for interacting with the server.

But it is often appropriate to hide knowledge of the details of a server protocol behind a "facade" class that provides methods that can be used by a client.

⋆ The class needs to implement methods that implement the protocol for interacting with the server. When an object of the class is created, it may need to fork a new server process. It is sometimes said to be a "proxy object" for the server it uses.

⋆ Methods of a (single) proxy object can be used by many concurrent client processes because synchronization is done automagically by the channel communication primitives.

⋆ Simple example: a suitable facade for the just-defined memory allocation server

```
class AllocatorServer(admin: PageAdmin) extends Allocator
{ import MemoryAllocationServer._

  private val toServer = ManyOne[Request]

  def request(count: Int): Set[Int] =
  { val reply = OneOne[Reply]
    toServer!Acquire(reply, count)
    reply ? { case Acquired(pages) ⇒ reply.close; pages }
  }

  def release(pages: Set[Int])
  { val reply = OneOne[Reply]
    toServer!Release(reply, pages)
    reply ? { case Released ⇒ reply.close; () }
  }

  // now start a memory allocation server running as a background process
  private val handle = fork(server(admin, toServer))
}
```

Monitors reimplemented as encapsulated Servers

⋆ Claim: any class implemented as a monitor can be implemented by encapsulating a server.

Contents

Note 1: Shared Channels v. Alternation 3 ☞

In this section we will focus on providing service for a number of clients that is unknown to the server; hence the use of shared channels. In an alternative model each client could have its own request channel, and the server could be defined using an **alt**. The disadvantage of the latter (when there are many clients) is that the computational cost of an alt is linear in its number of events.

Note 2: 4 ☞

Cf. the Haskell enumerated datatype

```
data Request = ReqKind1 Type1 .... | ReqKind2 Type2 .... | ....
```

You could also use a variant record in Pascal-like languages, or a union type in C++.

The parameters are set when the object is created, and cannot be changed, i.e., they're constant. That means they can be treated declaratively.

Also, we can do pattern matching on the type.

Or the request could be encoded as raw bytes, with one field showing the subtype.

Note 3: Concurrent actions between request and response 6 ☞

Notice that between a client sending a request and reading the corresponding reply there is room for it to do other computtions that don't depend on the reply.

Note 4: Coping with slow clients 6 ☞

Now the server isn't held up by a client that's slow to read its response, but sending the response ties up a whole thread until the response is received. This can be ameliorated with timeouts or by deputising a "delivery service" .

Note 5: Stateless Clients and concurrent requests 6 ☞

⋆ Alternatively clients may allocate one channel per request: this permits concurrent requests

```
def client(request: ![Request]) =
{ ··· initialization
  val R1 = proc { val rep=OneOne[Response]; request!ReqKind1 (rep, ···) ··· rep?() ···}
  val R2 = proc { val rep=OneOne[Response]; request!ReqKind2 (rep, ···) ··· rep?() ···}

  while (not finished)
  {
     ··· (R1 || R2 )() ···
  }
  ··· finalization
  request·closeOut
}
```

Note 6: Server-allocated request/response channels 8 ☞

If it is more effective for the *server* to allocate the request and response channels, then the client sends a port in its opening request to which the server can reply with the appropriate channel ends.

```
def server(open: ![?[Reply], ![Request]]) = proc
{  // server initialization ...
   serve (
     open =?⇒ { case open ⇒
                  { val reply:   Chan[Reply]   =
                    val request: Chan[Request] =
                    fork(session(reply, request))
                    open!((reply, request))
                  }
              }
   )
   // server termination ...
}
```

Note 7: Deadlined reads and writes in timeouts 12 ☞

An alternative way of writing the deadlined request read loop is:

```
request·readBefore(requestDeadline) match {
   case Some(request) ⇒
      {  case ReqKind1 (bv1 ) ⇒ { ··· ; deliver(response1 ); ··· }
         case ReqKind2 (bv2 ) ⇒ { ··· ; deliver(response2 ); ··· }
         ...
      }
   case None ⇒  stop
}
```

Deadlined reads and writes are somewhat more efficient than those embedded in alternations; moreover they have the advantage that they don't prevent the complementary ports of the channels appearing in alternations.

Note 8: Thread safety and the session-data mapping 14 ☞

The implementation of the session-data mapping doesn't need to be thread-safe given the current structure, for nothing is ever deleted from the mapping, and there is only one update per generated descriptor. A more realistic server might have a "close" `Request` whose outcome is the removal of session data for the given descriptor. Such requests can be handled by the server itself, not farmed out to handlers, and in this case all the mapping methods that are needed are run from the server process itself, so there are no races.

Note 9: Slow clients and invalid credentials 14 ☞

Notice that an opening request is simply ignored if the credential supplied is not "believed" by the server; and that if the opening client is too slow to pick up the descriptor, it is removed from the mapping.

Note 10: Stale descriptors 14 ☞

We have omitted the detailed treatment of session data that has become "stale" because the owning client hasn't used it recently. An obvious time to deal with removing stale session data from the session mapping is when the serve-loop finds nothing to do. An `after(···)` event can be added to the serve loop that triggers a scan of the mapping for such data.

Note 11: Assumptions about clients 14 ☞

Here we assume that clients will use descriptors only sequentially: no more than one request using a descriptor will be in progress at a time. We don't *enforce* this assumption dynamically. Doing so would require a shared "termination" channel (of type `Chan[Descriptor]`) back from the handlers to the server, that would enable the server to keep track of the session data objects currently "owned" by handlers.

Note 12: Ownership of the session-data object 15 ☞

In our sketch of the server we have assumed that the handlers and server(s) share address-space, and that the session-data objects passed to handlers are mutable and can be updated "in place". Our convention for managing session data is that a session data object that is read, with a request, by a handler is deemed to "belong" to that handler (and can be mutated by it) until the handler reads another request.

Note 13: Alternative server strategies 17 ☞

As in our earlier chapter, keeping a single request queue in increasing order of size can lead to *starvation* for (clients with) large requests because they might keep getting overtaken by small requests. This can be ameliorated by keeping several queues, each associated with a *priority*. After a release, the queues can be scanned (for satisfiable requests) in decreasing order of priority, and queued requests that remain unsatisfied can be moved to higher priority queues.

An additional feature would be to permit requests to have deadlines, and to queue unsatisfiable requests only until their deadline expires – responding `Fail` in that circumstance.

An alternative to queuing unsatisfiable requests would be to respond to them with a `Fail` reply.

Note 14: Reply channels 19 ☞

Notice that the reply channels are used only once, before being closed by the client.

Although we have not used them here, it is possible to design very lightweight and efficient forms of channel to be used this way.

Note 15: Fork 19 ☞

If P is a process definition, then **`fork(P)`** starts ("forks") a new process running P, and returns its "handle".

This process runs concurrently with the process that invoked **`fork(P)`**, and the invoking process can (if necessary) store the handle, and use it to wait for termination of the fork, or (in exceptional circumstances) to force its termination.

`fork(P)` can also be written P·**`fork`**.