

Somno, The Barber of Clapham Junction, Introduces GParS

Or how I learned to love Actor Model, Communicating Sequential Processes (CSP), and Dataflow Model, and treat threads as the assembly language of concurrency and parallelism.

Dr Russel Winder

2010-12-06

1 Introduction

The Java Virtual Machine (JVM) has been around since before Java became popular – remember the programming language and virtual machine were called Oak as part of Project Green, before it all got rebranded Java in 1995. From the outset it has supported threads. Initially this meant user space threads in a uniprocessor process, but as operating systems started to support kernel threads, JVM threads migrated to being kernel threads. As operating systems harnessed kernel threads for managing multiple processors, this meant that the JVM got parallelism for free: as long as the operating system scheduled kernel threads across all processors available then the JVM threads could potentially execute in parallel.

The Multicore Revolution over the last three or four years has terminated the era of the ever increasing clock speed of uniprocessors, and replaced it with the ever increasing count of cores on a processor chip. The era of the dual core processors is long gone, quad core is now the norm, 12-core is on the horizon, and 48-core will soon be going into production – at least for server chips, if not workstation and laptop chips.

Now whilst C++0x may (or may not) soon be ratified, this just brings a standard thread model, albeit with good things such as futures and asynchronous function calls. But this is low level infrastructure. Moreover, it still assumes shared-memory multithreading, and hence the need for locks, semaphores, monitors, and the ability to program all this concurrency technology. Go (<http://go-lang.org>) and D (<http://d-programming-language.org>) have both chosen to use lightweight processes and message passing as the way forward for managing concurrency and parallelism. Is this process and message passing technology something there should be C++ support for? Almost certainly yes if C++ is to remain relevant in the coming times. C++, Go and D though are in native code territory, what about the JVM?

The main thrust of Java development has been to accept that explicit shared-memory multithreading is not the right approach for application programming in a multi-processor context: the facilities in `java.util.concurrent` have become the proper tools for handling parallelism in Java code. This replaces explicit thread management with the use of “executors” – abstracted ways of working with thread and process pools – along with futures, asynchronous function calls, and parallel arrays. Underneath it remains shared-memory multithreading requiring locks, semaphores, monitors, atomics, etc., but the programmer works with a façade, with a few extra facilities, to make it more usable and less prone to error.

Scala (<http://www.scala-lang.org/>) a language that, like Java, targets the JVM, took the direction of realizing Actor Model as the principle tool of concurrent and parallel programming. People can still use threads and shared-memory multithreading if they really, really have to, but they are considered as low-level tools for realizing a higher level model of computation (actors) more suitable for day-to-day programming of concurrent and parallel systems.

The Actor Model was formulated in the early 1970s, but seems to have been studiously ignored by the mainstream programming languages for reasons that only the language designers involved will ever be able to shed light on.

Likely though, they probably won't be telling – as they probably don't know themselves.

JCSP (<http://www.cs.kent.ac.uk/projects/ofa/jcsp/>) has been around since 1997 and has been keeping the candle burning for process-based computation in Java since before Scala was around (c.2003): JCSP is a realization of Communicating Sequential Processes (CSP, <http://www.usingcsp.com/>) for Java. Sadly, most Java programmers have never heard of CSP, let alone JCSP.

CSP was initially expressed by Tony Hoare around 1978, but it only really hit the headlines in 1984 when his book (*Communicating Sequential Processes*, Prentice-Hall, 1985) was published. As with Actor Model, the mainstream computer languages studiously ignored this model, again for reasons which are unlikely to come to light. Conspiracy theories possibly lead to prejudice as the reason – CSP was perceived as mathematics not programming.

Another process-based model is “Dataflow Model”. Many people associate “dataflow” as a term with either:

- “dataflow diagrams” and the software analysis and design techniques popularized by Tom DeMarco (*Structured Analysis and System Specification*, Prentice-Hall, 1979) and Ed Yourdon and Larry Constantine (*Structured Design: Fundamentals of a Discipline of Computer Program and System Design*, Prentice-Hall, 1979); or
- the attempt to create “dataflow computers”, i.e. hardware.

Whilst there were some sterling efforts to create dataflow computers, dataflow never really took off as a computational model for hardware. However, the abstract architecture makes an excellent software architecture. The underlying “computational model” in both these variants of “dataflow” is in fact fundamentally the same, data flowing along channels between operations. The fact that it works for software whereas it probably doesn't in hardware is a definite opportunity.

2 The Models

The Actor Model, CSP, and Dataflow Model rely on using processes and message passing. Processes are distinct namespaces and/or address spaces. This does not necessarily mean separate processors, processes can be realized by partitioning a global address space – Erlang has been using this approach very successfully for many years. This approach is obviously very appropriate for today's multicore processors with multiple processors sharing a single memory. Processes can of course be separate address spaces on separate processor. This leads to an interesting issue: where does multicore end and distributed begin? This is really a question of communications. There are many levels of communications speed:

1. Bus on chip.
2. Bus on motherboard.
3. Local board cluster.
4. Local machine cluster.
5. Wide area network.

One of the difficulties of the moment for these process-based models is that they have just a single notion of communication: it is assumed that all processes are at just one of the communications levels. Knowing the cost of communication between processes is something that is critically important to the programmer of an algorithm. It will therefore soon have to be the case that weights will have to be given to the communications of the various processes so as to take into account where a target process is relative to the sending process. Parallel and distributed computing will have to merge into a single area of study.

Caveat communications cost, however any of these three models is realized, as far as the programmer is concerned, each process is completely independent and can only pass messages to other processes. There is (or, at least, should not be) any notion of shared state. Clearly though in a single memory realization of processes it can be tempting to utilize shared memory, such temptation must be fought against and eschewed.

So the commonality is that the three models are all process based. The differences between them lies in the way in which messages are passed and the synchronization behaviour that is integral to the processing of messages.

2.1 The Actor Model

Processes in the Actor Model each have one, and only one, incoming message queue, often called a mailbox. An actor can send a message to any other actor for which it has a reference to the target actor's message queue. How a sending actor gets a reference to the message queue of the receiving actor is not predetermined, it could be by being given a reference during construction or a reference might be sent as part of a message.

Messages are sent asynchronously in the Actor Model: the sender adds a message to the message queue of the receiver and continues execution. Each actor is responsible for processing messages in its queue. In effect an actor is an event processing system with the message queue being the event list.

2.2 CSP

CSP uses the idea of processes being connected by *channels*. Channels can be one-to-one, one-to-many, many-to-one, or many-to-many. Each process has zero or more input channels and zero or more output channels. As with Actor Model, there is an element of being event driven: a process is responsible for taking messages from its input channels, doing computation, and putting messages out on its output channels.

Apart from the fact that each process can have many rather than one input message queue, the other crucial difference between actors and CSP is that message passing is synchronous in CSP: message passing is actually a rendezvous between two processes. This makes it relatively straightforward to create systems that deadlock, but if it happens, it is surprisingly easy to discover what the deadlock is and how to fix it. Also because the processes are sequential and the channel properties can be modelled exactly with mathematics, it makes it (relatively) easy to write systems that reflect on code and determine whether or not that code will deadlock. The beauty of CSP is that you can find out with certainty whether your code will exhibit deadlock or not. If the tool for determining deadlock says your code can't deadlock, then it won't, this is not something that can be done for shared memory multithreading or Actor Model.

2.3 Dataflow Model

In the Dataflow Model the processes are usually referred to as operators – a hang over from the days of dataflow computer hardware most likely. Like CSP processes, Dataflow Model operators can have zero or more inputs and zero or more outputs. Unlike CSP and like Actor Model, Dataflow Model has asynchronous message sending. However whereas Actor Model and CSP require an explicit activity in the process for receiving messages, Dataflow Model pins the execution and synchronization behaviour of an operator to the reception of messages on its inputs. The simplest of the algorithms is for the operator to be idle until there is valid input on all the incoming channels. Having valid inputs on all channels is the event that triggers execution in the operator, which then does something and puts data out on the output channels. Alternative algorithms are possible, for example trigger execution on reception of the first message on any input.

2.4 Which Model When

It may not be obvious at this stage but it is possible to implement any of the three process-based models in any of the others. However to do so would likely lead to inefficient systems. It is far better to implement all three independently using the low-level thread and process management of the underlying platform. This means treating the three as distinct. The different message receive and execution trigger properties of the models become the factors for determining what facilities of what packages to use for a given solution to a given problem: some solutions to some problems will more naturally be solved using one model rather than another. So what is a Java programmer to do? `java.util.concurrent` has some great tools but... Groovy and GParc provide massively useful augmentations.

3 GPars

GPars is a Groovy attempt to bring Actor Model, CSP, and Dataflow Model to the Java and Groovy communities – Scala people can also use it, but this seems less likely to happen for various reasons, some technical, most psychological and/or social. The idea of using Groovy as a platform for creating a framework for managing concurrency and parallelism on the JVM had been mooted a number of times on the Groovy mailing lists in 2008 and early 2009. Václav Pech decided to act rather than talk, and created the GParallelizer project. This was a personal project to get things moving. Mid-2009 various people (including myself) joined Václav on the project. After a little debate, we moved the project to Codehaus (<http://www.codehaus.org>), the home of Groovy (<http://groovy.codehaus.org>), and rebranded it GPars (<http://gpars.codehaus.org>). Since then Václav has been tireless in moving things forward supported to a greater or lesser extent by the other members of the team.

Why use Groovy rather than just Java, or Scala? Well the Scala concurrency and parallelism support is ploughing its own furrow based on Actor Model, and supporting frameworks such as Akka and Scalaz. The Groovy effort, driven via GPars, is about providing high-level, low-overhead augmentation of what is available in Java. GPars harnesses the meta-object protocol of Groovy to provide a very easy way of expressing concurrent and parallel computations.

Of course all this is very general and vague, what is needed is something less abstract, something more concrete. Basically we need some examples.

4 Concurrency (and Parallelism), The Classic Problems

Since multiprogramming was first invented, operating systems programming has been spawning problems in concurrency and share memory management. The 1960s and 1970s saw an entire industry in educating people how to manage resources using semaphores, monitors and locks. Sadly these problems, and techniques for solution, were foisted on applications programmers as well as systems programmers, but that rant is wholly, but not completely, inappropriate for this article.

The classic problem in concurrency and resource management that every programmer is forced to study is the “Dining Philosophers Problem”. The underlying issue is one of resources being managed by operating systems – and the origins of “Dining Philosophers” is just such a problem set out by Edsger Dijkstra, but Tony Hoare reformulated it in real-world terms so as to make it more comprehensible. Thus began two industries: a) providing multiple solutions in multiple languages to the “Dining Philosophers Problem”; and b) finding new and novel model problems for concurrency problems in operating systems couched as real-world vignettes.

The “Sleeping Barber Problem” (also attributed to Edsger Dijkstra) is just such a problem. Like “Dining Philosophers Problem” it is a model of a process synchronization issue in operating systems. But as Tony Hoare showed, these things are far more interesting, and take on much more life, when couched in real-world terms. Enter Somno, The Barber of Clapham Junction.

Despite rumours, it is known that Somno does not have brother called Figaro living in Seville. Moreover Somno does not sing, not even in a barber’s shop quartet.

5 The Sleeping Barber Problem

Somno owns a barber shop. He has one hair-cutting chair and four waiting chairs. Music is provided via Somno’s MP3 player, an amplifier and speakers, there is no barber’s shop quartet, Somno does not sing. If there are no customers Somno snoozes in the hair-cutting chair. If there are any customers, Somno cuts hair of customers, one at a time. Potential customers enter the shop and if they see Somno asleep in the chair, they wake him up, take the seat themselves and get a hair cut. If Somno is cutting another customer’s hair then the potential new customer checks the four waiting chairs to see if one is vacant. If there is a free chair the customer sits and waits their turn. If there are no free chairs, the person exits – somewhat less than chuffed at not being able to transform from potential customer to customer. Potential customers arrive at random intervals, and each customer hair cut takes a random amount of time.

At some point Somno is thinking of getting a second, or even a third hair-cutting chair, allowing for there to be more

than one barber in action concurrently. For now though cash flow doesn't allow for this to be implemented. Somno is saving up for a new pair of scissors.

5.1 Using Threads

The Wikipedia article (http://en.wikipedia.org/wiki/Sleeping_barber_problem) on the Sleeping Barber Problem describes things much more according to the operating system problem that inspired the description above. The customers are processes or threads as is the barber: the problem is all about synchronizing and queuing processes or threads. The article presents a pseudocode solution based on using semaphores. This can be coded up very easily in Java or Groovy using threads and the `java.util.concurrent.Semaphore` class, so as to avoid either writing an implementation of semaphores or using the Java wait/notify technology explicitly – which generally results in incomprehensibly deadlocked code. Here is a Groovy version, the Java version would be fundamentally the same, just more verbose:

```
1  #!/usr/bin/env groovy
2
3  // This is a model of the "The Sleeping Barber" problem using Groovy (http://groovy.codehaus.org),
4  // cf. http://en.wikipedia.org/wiki/Sleeping_barber_problem.
5  //
6  // Copyright (c) 2010 Russel Winder
7
8  import java.util.concurrent.Semaphore
9
10 def runSimulation ( final int numberOfCustomers , final int numberOfWaitingSeats ,
11                    final Closure hairTrimTime , final Closure nextCustomerWaitTime ) {
12     final customerSemaphore = new Semaphore ( 1 )
13     final barberSemaphore = new Semaphore ( 1 )
14     final accessSeatsSemaphore = new Semaphore ( 1 )
15     def customersTurnedAway = 0
16     def customersTrimmed = 0
17     def numberOfFreeSeats = numberOfWaitingSeats
18     final barber = new Runnable () {
19         private working = true
20         public void stopWork () { working = false }
21         @Override public void run () {
22             while ( working ) {
23                 customerSemaphore.acquire ()
24                 accessSeatsSemaphore.acquire ()
25                 ++numberOfFreeSeats
26                 barberSemaphore.release ()
27                 accessSeatsSemaphore.release ()
28                 println ( 'Barber : Starting Customer.' )
29                 Thread.sleep ( hairTrimTime () )
30                 println ( 'Barber : Finished Customer.' )
31             }
32         }
33     }
34     final barberThread = new Thread ( barber )
35     barberThread.start ()
36     final customerThreads = []
37     for ( number in 0 ..< numberOfCustomers ) {
38         println ( "World : Customer ${number} enters the shop." )
39         final customerThread = new Thread ( new Runnable () {
40             public void run () {
41                 accessSeatsSemaphore.acquire ()
42                 if ( numberOfFreeSeats > 0 ) {
43                     println ( "Shop : Customer ${number} takes a seat. ${numberOfWaitingSeats - numberOfFreeSeats} in use." )
44                     --numberOfFreeSeats
45                     customerSemaphore.release ()
46                     accessSeatsSemaphore.release ()
47                     barberSemaphore.acquire ()
48                     println ( "Shop : Customer ${number} leaving trimmed." )
49                     ++customersTrimmed
50                 }
51                 else {
52                     accessSeatsSemaphore.release ()
53                     println ( "Shop : Customer ${number} turned away." )
54                     ++customersTurnedAway
55                 }
56             }
57         } )
58         customerThreads.add ( customerThread )
59         customerThread.start ()
60     }
61 }
```

```

56         }
57     })
58     customerThreads << customerThread
59     customerThread.start ()
60     Thread.sleep ( nextCustomerWaitTime () )
61 }
62 customerThreads*.join ()
63 barber.stopWork ()
64 barberThread.join ()
65 println ( "\nTrimmed ${customersTrimmed} and turned away ${customersTurnedAway} today." )
66 }
67
68 runSimulation ( 20 , 4 , { ( Math.random () * 60 + 10 ) as int } , { ( Math.random () * 20 + 10 ) as int } )

```

A few thoughts and points on this code:

- 1 Groovy realizes the #! magic number start word specified by Posix for executables. This means Groovy scripts can be commands on Posix-compliant systems. Those addicted to Windows use will just have to remain green with envy. Either that or join the ranks of those using a proper operating system.
- 36 Groovy allows for list literals, e.g. [], the resultant object is of type ArrayList by default.
- 58 Groovy allows for overloading of operators – like C++ and completely unlike Java. The << in this context is appending a value to a list.
- 62 The operator *. here is a “spread apply” operation. The left-hand operand is a list and the right-hand operand is the method call to be applied to each element in the list.
- 68 Groovy has the facility to create anonymous functions that can be passed as parameters: code in curly brackets in a context other than one of “code block” create a function object. Groovy gives these things the type Closure which is actually a bit of a misnomer, they are not closures (a closure is a function with an environment providing bindings for all the free variables in the function) they are functions potentially with free variables. In this case it happens not to be the case, so we can’t tell that anonymous functions are not actually closures. Which is good.

There are undoubtedly other interesting, or at least quite interesting, points about the code, but let us leave it there for now. If there are questions that are not answered by the end of the article, feel free to email me, or perhaps raise the issue on the ACCU General email list.

Running the above program will result in something such as the following. Since there is an element or two of randomness in the code, no two runs of the code are guaranteed to be the same.

```

World : Customer enters shop.
Barber : Starting Customer.
Shop : Customer takes a seat. -1 in use.
Shop : Customer leaving trimmed.
World : Customer enters shop.
Shop : Customer takes a seat. 0 in use.
Shop : Customer leaving trimmed.
World : Customer enters shop.
Shop : Customer takes a seat. 1 in use.
Barber : Finished Customer.
Barber : Starting Customer.
Shop : Customer leaving trimmed.
World : Customer enters shop.
Shop : Customer takes a seat. 1 in use.
Barber : Finished Customer.
Barber : Starting Customer.
Shop : Customer leaving trimmed.
World : Customer enters shop.
Shop : Customer takes a seat. 2 in use.
World : Customer enters shop.
Shop : Customer takes a seat. 3 in use.
Barber : Finished Customer.
Barber : Starting Customer.
Shop : Customer leaving trimmed.
World : Customer enters shop.
Shop : Customer takes a seat. 3 in use.
World : Customer enters shop.
Shop : Customer turned away.
Barber : Finished Customer.

```

```

Barber : Starting Customer.
Shop : Customer leaving trimmed.
World : Customer enters shop.
Shop : Customer takes a seat. 3 in use.
World : Customer enters shop.
Shop : Customer turned away.
Barber : Finished Customer.
Barber : Starting Customer.
Shop : Customer leaving trimmed.
World : Customer enters shop.
Shop : Customer takes a seat. 3 in use.
World : Customer enters shop.
Shop : Customer turned away.
World : Customer enters shop.
Shop : Customer turned away.
Barber : Finished Customer.
Barber : Starting Customer.
Shop : Customer leaving trimmed.
World : Customer enters shop.
Shop : Customer takes a seat. 3 in use.
World : Customer enters shop.
Shop : Customer turned away.
World : Customer enters shop.
Shop : Customer turned away.
Barber : Finished Customer.
Barber : Starting Customer.
Shop : Customer leaving trimmed.
World : Customer enters shop.
Shop : Customer takes a seat. 3 in use.
World : Customer enters shop.
Shop : Customer turned away.
World : Customer enters shop.
Shop : Customer turned away.
Barber : Finished Customer.
Barber : Starting Customer.
Shop : Customer leaving trimmed.
World : Customer enters shop.
Shop : Customer takes a seat. 3 in use.
World : Customer enters shop.
Shop : Customer turned away.
World : Customer enters shop.
Shop : Customer turned away.
Barber : Finished Customer.
Barber : Starting Customer.
Shop : Customer leaving trimmed.
Barber : Finished Customer.
Barber : Starting Customer.
Shop : Customer leaving trimmed.
Barber : Finished Customer.
Barber : Starting Customer.
Shop : Customer leaving trimmed.
Barber : Finished Customer.
Barber : Starting Customer.
Shop : Customer leaving trimmed.
Barber : Finished Customer.

```

Trimmed 12 and turned away 8 today.

Which is (quite) interesting, not least because Somno appears to start cutting the hair of the first customer before that customer has arrived in the shop!

One of the problems with the code above is that it is trying to (minimally) solve the thread/process scheduling problem, annotated with pointers to the analogy, rather than being a model of Somno's barber shop. There are therefore some inconsistencies between the activity and the analogy used to describe the problem. The solution is to forget the operating system problem that inspired the analogy and to model and realize the analogy more directly, i.e. take a much more simulation-oriented view of the whole problem.

Also of course, the above is really a lesson in how not to do things unless you are writing an operating system – and who would want to write an operating system in Groovy, or even Java.

There are rumours that a number of universities are using Java as the programming language for exercises in operating systems courses – bizarre.

So what we should do is move much more towards a simulation modelling approach. The core change is that customers become represented as data rather than being labels on processes. This models far more literally what happens in Somno's shop.

```

1  #! /usr/bin/env groovy
2
3  // This is a model of the "The Sleeping Barber" problem using Groovy (http://groovy.codehaus.org) only,
4  // cf. http://en.wikipedia.org/wiki/Sleeping\_barber\_problem.
5  //
6  // Copyright (c) 2010 Russel Winder
7
8  import java.util.concurrent.ArrayBlockingQueue
9
10 import groovy.transform.Immutable
11
12 @Immutable class Customer { Integer id }
13

```

```

14 def runSimulation ( final int numberOfCustomers , final int numberOfWaitingSeats ,
15                   final Closure hairTrimTime , final Closure nextCustomerWaitTime ) {
16   final waitingChairs = new ArrayBlockingQueue<Customer> ( numberOfWaitingSeats )
17   final customersTurnedAway = 0
18   final customersTrimmed = 0
19   final barber = new Runnable () {
20     private working = true
21     public void stopWork () { working = false }
22     @Override public void run () {
23       while ( working || ( waitingChairs.size () > 0 ) ) {
24         def customer = waitingChairs.take ()
25         assert customer instanceof Customer
26         println ( "Barber : Starting Customer ${customer.id}." )
27         Thread.sleep ( hairTrimTime () )
28         println ( "Barber : Finished Customer ${customer.id}." )
29         ++customersTrimmed
30         println ( "Shop : Customer ${customer.id} leaving trimmed." )
31       }
32     }
33   }
34   final barberThread = new Thread ( barber )
35   barberThread.start ()
36   for ( number in 0 ..< numberOfCustomers ) {
37     Thread.sleep ( nextCustomerWaitTime () )
38     println ( "World : Customer ${number} enters the shop." )
39     final customer = new Customer ( number )
40     if ( waitingChairs.offer ( customer ) ) {
41       println ( "Shop : Customer ${customer.id} takes a seat. ${waitingChairs.size ()} in use." )
42     }
43     else {
44       ++customersTurnedAway
45       println ( "Shop : Customer ${customer.id} turned away." )
46     }
47   }
48   barber.stopWork ()
49   barberThread.join ()
50   println ( "\nTrimmed ${customersTrimmed} and turned away ${customersTurnedAway} today." )
51 }
52
53 runSimulation ( 20 , 4 , { ( Math.random () * 60 + 10 ) as int } , { ( Math.random () * 20 + 10 ) as int } )

```

Note that we have to allow Somno to finish all the waiting customers (cf. 48, 23, 49) before actually shutting up shop, even after it is closed. It would be somewhat uncivilized to allow customers to seat themselves expecting a hair cut only to be ushered out of the door, sans trim, just because Somno wants to knock off and go home.

An alternative might be for Somno to start trying to sing when he wants to go home. The appalling noise would almost certainly clear the shop in a huge hurry.

Now whilst the above is a (relatively) simple, two-thread solution with the waiting chairs being the thread-safe shared state, there is a lot of data coupling. For example notification about customers leaving the shop are handled in the barber thread. Just because Somno has finished cutting the customers hair doesn't imply the customers has actually paid up and left the shop. Also the world and shop are being modelled with the same thread. In effect we haven't modelled separately the idea of world and shop, which would properly decouple things.

So perhaps we should write a new version with a barber object, a shop object, perhaps even a world object, each having a Singleton instance, and animated with many threads.

Mentioning Singleton here is of course offering the proverbial red rag to the proverbial bull, especially as the readership here are ACCU members. I have no doubt there will be a re-emergence of the traditional anti-Singleton thread on ACCU General email list if anyone actually reads this bit of this article.

6 An Analytic Intervention

Is it obvious? We are rapidly moving towards creating a number of objects (world, shop, barber) each animated with a thread with each object communicating to other objects by passing customers around. This sounds like processes and message passing à la Actor Model, CSP, or Dataflow Model!

It should be no surprise to anyone that trying to enforce encapsulation and decoupling in this simulation modelling approach leads, especially in an object-oriented programming context, to processes and message passing. After all, object-orientation is all about processes and message passing: the object-oriented model resulted from investigating simulation, cf. Simula-67.

Using shared-memory multithreading explicitly at the same time as claiming to be following an object-oriented approach seems to be an act of blatant doublethink.

Let's short-circuit showing the whole sequence of solutions between above and below and jump straight to GPar-based solutions – thereby avoiding reimplementing GPar, which is effectively what would happen.

7 Actor Model

This section presents a version of the Sleeping Barber Problem in which each entity is modelled with an actor. There are actors for barber, shop and world: well at least barber and shop, the world is modelled using the initial thread that executed the program.

Before reading the following code, it is probably worth noting that the @Grab annotation is one specific to Groovy (it is part of the Grapes system in Groovy) that uses Ivy under the covers to ensure that the named artifact is in the classpath for execution of the code, downloading the artifact if necessary. In the case below the Grab is ensuring that the GPar artifact version 0.11-beta-2 is used which is the latest version of GPar in the Maven repository.

Grapes has to be deemed extremely cool.

```
1  #! /usr/bin/env groovy
2
3  // This is a model of the "The Sleeping Barber" problem using Groovy (http://groovy.codehaus.org) and GPar
4  // (http://gpar.codehaus.org) actors, cf. http://en.wikipedia.org/wiki/Sleeping\_barber\_problem.
5  //
6  // Copyright (c) 2009-10 Russel Winder
7
8  @Grab ( 'org.codehaus.gpars:gpars:0.11' )
9
10 import groovy.transform.Immutable
11
12 import groovyxx.gpars.group.DefaultPGroup
13
14 @Immutable class Customer { Integer id }
15 @Immutable class SuccessfulCustomer { Customer customer }
16
17 def runSimulation ( final int numberOfCustomers , final int numberOfWaitingSeats ,
18                   final Closure hairTrimTime , final Closure nextCustomerWaitTime ) {
19     def group = new DefaultPGroup ( )
20     def barber = group.reactor { customer ->
21         assert customer instanceof Customer
22         println ( "Barber : Starting Customer ${customer.id}." )
23         Thread.sleep ( hairTrimTime ( ) )
24         println ( "Barber : Finished Customer ${customer.id}." )
25         new SuccessfulCustomer ( customer )
26     }
27     def shop = group.actor {
28         def seatsTaken = 0
29         def isOpen = true
30         def customersTurnedAway = 0
31         def customersTrimmed = 0
32         loop {
33             react { message ->
34                 switch ( message ) {
35                     case Customer :
36                         if ( seatsTaken <= numberOfWaitingSeats ) {
37                             ++seatsTaken
38                             println ( "Shop : Customer ${message.id} takes a seat. ${seatsTaken} in use." )
39                             barber.send ( message )
40                         }
41                     else {
42                         println ( "Shop : Customer ${message.id} turned away." )
43                         ++customersTurnedAway
44                     }
45                 }
46             }
47         }
48     }
49 }
```

```

44     }
45     break
46     case SuccessfulCustomer :
47         --seatsTaken
48         ++customersTrimmed
49         println ( "Shop : Customer ${message.customer.id} leaving trimmed." )
50         if ( ! isOpen && ( seatsTaken == 0 ) ) {
51             println ( "\nTrimmed ${customersTrimmed} and turned away ${customersTurnedAway} today." )
52             stop ( )
53         }
54         break
55     case "" : isOpen = false ; break
56     default : throw new RuntimeException ( "Shop got a message of unexpected type ${message.class}" )
57 }
58 }
59 }
60 }
61 for ( number in 0 ..< numberOfCustomers ) {
62     Thread.sleep ( nextCustomerWaitTime ( ) )
63     println ( "World : Customer ${number} enters the shop." )
64     shop.send ( new Customer ( number ) )
65 }
66 shop.send ( "" )
67 shop.join ( )
68 }
69
70 runSimulation ( 20 , 4 , { ( Math.random ( ) * 60 + 10 ) as int } , { ( Math.random ( ) * 20 + 10 ) as int } )

```

Various notes about the code:

- 15 Message passing in the Actor Model invariably requires the use of case classes. Since an actor has but a single message queue, the type of a message becomes very important metadata that drives the computation. The shop has to process messages coming from both the world and the barber and there has to be a way of distinguishing which message came from where. The barber therefore (25) returns a customer wrapped as a SuccessfulCustomer so that the shop can make the distinction. cf. 34–57.
- 19 Thread pools are used to animate the actors, hence the notion of a group: all actors in a given group will be animated by the threads in the associated thread pool. In this case we have just the one group.
- 20 Somno is realized as a reactor. A reactor is an actor that has no persistent state, it simply processes received message and returns a message to the sender of the received message. A nice abstraction for Somno the barber. Somno sleeps if there are no customers (modelled by being blocked waiting for an item on the message queue: the message queue models the waiting chairs in the shop) and reacts to a new customer by giving them a hair cut and sending them back to the shop wrapped as a SuccessfulCustomer
- 61–67 The world is not an actor just the initial thread sending in customers to the shop. Note that we use an empty string as a marker to close the shop. Messages can be of any type, the receiving actor must deal with this. so in the **switch** statement (34–57) we have 55 which deals with this signal to close the shop, sent in 66.

The complexity in the shop actor is due to the need to managing persistent state. It is an actor, but not a simple reactor, so therefore it needs a react loop. So there is a loop construct (32–59) and a react construct (33–58), which defines the function to execute on reception of a message. loop is just a function that takes a “closure” parameter and executes that “closure” according to the loop constraints – in this case loop infinitely. react is a function that takes a one-parameter “closure” that gets called for each message received by the actor. The shop receives customers from the outside world and from the barber. Hence the need for the switch and case classes in the react function to distinguish where the customer has come from. The shop keeps track of how many customers there are in Somno’s waiting queue, and is responsible for generating statistics for the day’s business.

Note that we use the sending in of an empty string instead of a customer instance as the marker to close the shop. As previously though we must let the barber process all the waiting customers before it is home time. This way of closing the shop – processing a fixed number of customers and then sending in a special message – is clearly not a good “physical” simulation, but we’ll live with it for now.

8 Groovy CSP

Groovy CSP is a Groovy layer over JCSP – all the power of CSP as implemented in JCSP, with all the ease and simplicity of Groovy as a language for creating internal, domain specific languages. Sadly at the time of writing Groovy CSP hasn't had the work done that it deserves, so the code is a little bit ugly. The hope and intention is to evolve the Groovy CSP system to make much of what is in fact boilerplate code, expressible in a neater and shorter way.

```
1  #! /usr/bin/env groovy
2
3  // This is a model of the "The Sleeping Barber" problem using Groovy (http://groovy.codehaus.org) and
4  // Groovy CSP (a part of GPars, http://gpars.codehaus.org), cf. http://en.wikipedia.org/wiki/Sleeping\_barber\_problem.
5  //
6  // Copyright (c) 2010 Russel Winder
7
8  @Grab ( 'org.codehaus.jcsp:jcsp:1.1-rc5' )
9  @Grab ( 'org.codehaus.gpars:gpars:0.11' )
10
11 import groovy.transform.Immutable
12
13 import org.jcsp.util.Buffer
14 import org.jcsp.lang.Channel
15 import org.jcsp.lang.CSProcess
16
17 import groovyx.gpars.csp.PAR
18 import groovyx.gpars.csp.ALT
19
20 @Immutable class Customer { Integer id }
21
22 def runSimulation ( final int numberOfCustomers , final int numberOfWaitingSeats ,
23                   final Closure hairTrimTime , final Closure nextCustomerWaitTime ) {
24     final worldToShopChannel = Channel.oneZone ( )
25     final shopToBarberChannel = Channel.oneZone ( new Buffer ( numberOfWaitingSeats ) )
26     final barberToShopChannel = Channel.oneZone ( )
27     final barber = new CSProcess ( ) {
28         @Override public void run ( ) {
29             final fromShopChannel = shopToBarberChannel.in ( )
30             final toShopChannel = barberToShopChannel.out ( )
31             while ( true ) {
32                 final customer = fromShopChannel.read ( )
33                 if ( customer == "" ) { break }
34                 assert customer instanceof Customer
35                 println ( "Barber : Starting Customer ${customer.id}." )
36                 Thread.sleep ( hairTrimTime ( ) )
37                 println ( "Barber : Finished Customer ${customer.id}." )
38                 toShopChannel.write ( customer )
39             }
40         }
41     }
42     final shop = new CSProcess ( ) {
43         @Override public void run ( ) {
44             final fromBarberChannel = barberToShopChannel.in ( )
45             final fromWorldChannel = worldToShopChannel.in ( )
46             final toBarberChannel = shopToBarberChannel.out ( )
47             final selector = new ALT ( [ fromBarberChannel , fromWorldChannel ] )
48             def seatsTaken = 0
49             def customersTurnedAway = 0
50             def customersTrimmed = 0
51             def isOpen = true
52             mainloop:
53             while ( true ) {
54                 switch ( selector.select ( ) ) {
55                     case 0 : //////// From the Barber //////////
56                     def customer = fromBarberChannel.read ( )
57                     assert customer instanceof Customer
58                     --seatsTaken
59                     ++customersTrimmed
60                     println ( "Shop : Customer ${customer.id} leaving trimmed." )
61                     if ( ! isOpen && ( seatsTaken == 0 ) ) {
62                         println ( "\nTrimmed ${customersTrimmed} and turned away ${customersTurnedAway} today." )
63                         toBarberChannel.write ( "" )
64                         break mainloop
65                     }
66                 }
67             }
68         }
69     }
70     PAR ( barber , shop )
71 }
```

```

66     break
67     case 1 : ////////// From the World //////////
68     def customer = fromWorldChannel.read ()
69     if ( customer == "" ) { isOpen = false }
70     else {
71         assert customer instanceof Customer
72         if ( seatsTaken < numberOfWaitingSeats ) {
73             ++seatsTaken
74             println ( "Shop : Customer ${customer.id} takes a seat. ${seatsTaken} in use." )
75             toBarberChannel.write ( customer )
76         }
77         else {
78             println ( "Shop : Customer ${customer.id} turned away." )
79             ++customersTurnedAway
80         }
81     }
82     break
83     default :
84         throw new RuntimeException ( 'Shop : Selected a non-existent channel.' )
85     }
86 }
87 }
88 }
89 final world = new CSPProcess () {
90     @Override public void run () {
91         def toShopChannel = worldToShopChannel.out ()
92         for ( number in 0 ..< numberOfCustomers ) {
93             Thread.sleep ( nextCustomerWaitTime () )
94             println ( "World : Customer ${number} enters the shop." )
95             toShopChannel.write ( new Customer ( number ) )
96         }
97         toShopChannel.write ( "" )
98     }
99 }
100 new PAR ( [ barber , shop , world ] ).run ()
101 }
102
103 runSimulation ( 20 , 4 , { ( Math.random () * 60 + 10 ) as int } , { ( Math.random () * 20 + 10 ) as int } )

```

Some commentary:

- 21 No extra “case class”. Because CSP allows for multiple input channels, the shop can distinguish messages from the world and from the barber by having them to be sent on different channels. If we had a many-to-one channel for both the world and the barber to send messages to the shop, then we would need case classes – but why bother when we can just use multiple one-to-one channels.
- 24–26 We must explicitly create the channels. The channel for the world to communicate to the shop and the channel for the barber to communicate to the shop are both simple one-to-one channels that require rendezvous between the processes. The channel for the shop to send messages to the barber has a buffer. It is this that introduces the element of asynchronous behaviour in the communication between shop and barber. This just models the waiting seats of course.
- 27, 42, 89 Barber, shop and world are processes: the way CSP works everything has to be a process, we cannot “get away with”, as we did in the actor version in the previous section, using the initial thread to represent the world.
- 100 Once all the processes are set up, and all the channels connected to the processes, we have to create a process that determines how the other processes execute. In this case we start all the processes in parallel and let the communication of messages between the processes determine what happens.
- 47, 54–85 Because message passing is synchronous in CSP and the shop has multiple input channels, it has to have a way of choosing between the channels. So in 48 we create an “alting” object. The select method of this object blocks until there is a message available on one of the channels and return the index in the original list of the channel that is ready to read. There are therefore two “branches” of control flow, one for a barber channel message and one for a world channel message.
- 33, 51, 61–65, 69, 97 As with the Actor Model version in the previous section, an “out of band” message, i.e. a message with an unusual type that constitutes a case class, is used to signal termination. The world sends an empty string message to the shop just prior to terminating. The shop uses this empty string message to set the shop

state to closed. When the barber has cut the last customer's hair, the shop sends an empty string message to the barber telling it to terminate, and then terminates itself.

- 52, 64 Crikey, a label, and what effectively amounts to a goto. Should Somno declare “*shock, horror, . . . , probe*”? Not really. The problem here is that **break** on its own is relevant to the **switch** and we need to have a break out of the **while**. The option of introducing a Boolean to handle this seems overcomplicated compared to using the “break out of a labelled statement” feature of Java and Groovy.

There are of course many other ways of structuring this code. For example we could have explicitly defined classes for barber, shop and world with constructors. Somno and I will leave this as an “exercise for the student”.

9 Dataflow Model

The following solution to The Sleeping Barber Problem not only uses Dataflow Model, it uses a lot of “short cut” features that GParS provides. Structurally the code is not dissimilar to the Groovy CSP version in the previous section. This is entirely reasonable: the algorithm is fundamentally similar except that we are using asynchronous submission of values to dataflow queues instead of synchronous communication via CSP channels. Another aspect of the the reason this code looks cleaner and simpler than the Groovy CSP code is that more work has already gone into the “dataflow DSL” compared to Groovy CSP.

```
1  #! /usr/bin/env groovy
2
3  // This is a model of the "The Sleeping Barber" problem using Groovy (http://groovy.codehaus.org) and GParS
4  // (http://gpars.codehaus.org) dataflow, cf. http://en.wikipedia.org/wiki/Sleeping\_barber\_problem.
5  //
6  // Copyright (c) 2010 Russel Winder
7
8  @Grab ( 'org.codehaus.gpars:gpars:0.11' )
9
10 import groovy.transform.Immutable
11
12 import groovyx.gpars.dataflow.DataFlow
13 import groovyx.gpars.dataflow.DataFlowQueue
14
15 @Immutable class Customer { Integer id }
16
17 def runSimulation ( final int numberOfCustomers , final int numberOfWaitingSeats ,
18                   final Closure hairTrimTime , final Closure nextCustomerWaitTime ) {
19     def worldToShop = new DataFlowQueue ()
20     def shopToBarber = new DataFlowQueue ()
21     def barberToShop = new DataFlowQueue ()
22     final barber = DataFlow.task {
23         while ( true ) {
24             def customer = shopToBarber.val
25             if ( customer == "" ) { break }
26             assert customer instanceof Customer
27             println ( "Barber : Starting Customer ${customer.id}." )
28             Thread.sleep ( hairTrimTime () )
29             println ( "Barber : Finished Customer ${customer.id}." )
30             barberToShop << customer
31         }
32     }
33     final shop = DataFlow.task {
34         def seatsTaken = 0
35         def customersTurnedAway = 0
36         def customersTrimmed = 0
37         def isOpen = true
38         mainloop:
39         while ( true ) {
40             def selector = DataFlow.select ( barberToShop , worldToShop )
41             def item = selector.select ()
42             switch ( item.index ) {
43                 case 0 : ////////// From the Barber //////////
44                     assert item.value instanceof Customer
45                     --seatsTaken
46                     ++customersTrimmed
47                     println ( "Shop : Customer ${item.value.id} leaving trimmed." )
```

```

48     if ( ! isOpen && ( seatsTaken == 0 ) ) {
49         println ( "\nTrimmed ${customersTrimmed} and turned away ${customersTurnedAway} today." )
50         shopToBarber << ""
51         break mainloop
52     }
53     break
54     case 1 : ////////// From the World //////////
55     if ( item.value == "" ) { isOpen = false }
56     else {
57         assert item.value instanceof Customer
58         if ( seatsTaken < numberOfWaitingSeats ) {
59             ++seatsTaken
60             println ( "Shop : Customer ${item.value.id} takes a seat. ${seatsTaken} in use." )
61             shopToBarber << item.value
62         }
63         else {
64             println ( "Shop : Customer ${item.value.id} turned away." )
65             ++customersTurnedAway
66         }
67     }
68     break
69     default :
70     throw new RuntimeException ( 'Shop : Selected an non-existent queue.' )
71 }
72 }
73 }
74 for ( number in 0 ..< numberOfCustomers ) {
75     Thread.sleep ( nextCustomerWaitTime ( ) )
76     println ( "World : Customer ${number} enters the shop." )
77     worldToShop << new Customer ( number )
78 }
79 worldToShop << ""
80 // Make sure all computation is over before terminating.
81 [ barber , shop ]*.join ( )
82 }
83
84 runSimulation ( 20 , 4 , { ( Math.random ( ) * 60 + 10 ) as int } , { ( Math.random ( ) * 20 + 10 ) as int } )

```

Some commentary:

24 Attempting to obtain the next value from the DataFlowQueue (remember Groovy has properties, in this case shopToBarber.val means shopToBarber.getVal ()) causes a block pending availability of a value – Somno sleeps until there is a customer of whom to cut the hair.

30, 50, 61, 77, 79 The << operator targeting a DataFlowQueue is an operator overload and means insert the item in the queue. Probably obvious to C++ people, less so to Java people.

25, 50, 55, 79 An empty string message remains the tool for signalling the end of computation.

40–41 The shop has two input queues. Rather than waiting for valid input on both, we need to ensure we process valid values as soon as they are available. Hence a selector on which we can select. This is directly analogous to the select mechanism in CSP, except that we are dealing with queues and not channels. In GPar, the select method returns an object that packages the available value and the index of the selector from which the value was retrieved.

As always there are many, many variants that could be written using the same underlying infrastructure. In this case (Dataflow Model), we could use explicit operators. Somno (and I) are of the opinion that this is an excellent “exercise for the student”. Of course it may be that we could be convinced to write a follow up article presenting these alternate solutions.

10 Reflections

Ignoring the first version, and looking at the other four – threads, actors, CSP, dataflow – it is clear that the threads version has fewer lines of code. In this case, I suggest that more is more. Although the threads version is shorter, I claim the other three are actually easier to comprehend. Of course the trouble is I wrote them. I guess I need to not read the code for six months and then see.

Despite the similarities between the actors, CSP and dataflow versions, a different mental model of execution is required – the different message passing semantics have to be handled explicitly.

A very CSP oriented viewpoint has been used to write the dataflow version, and so the dataflow version looks more like the CSP version than perhaps it should – at least in trying to present actors, CSP and dataflow as three separately useful tools.

Anecdotal experience indicates that the actor, CSP and dataflow versions are much easier not to get wrong compared to the threads version.

11 Summary

Of course these solution are unlikely to be useful as solution to the operating system resource management problem per se, not least because it is highly unlikely that anyone will ever write an operating system in a mix of Groovy and Java. There is though the distinct possibility that D and/or Go will eventually be used to write an operating system. Since D implements Actor Model and Go implements CSP (more or less), some of the ideas in this article may eventually become part of operating system orthodoxy.

It might even be the case that C++ gets Actor Model and Dataflow Model infrastructure based on the new C++0x standard. There is already a CSP implementation, C++CSP2, but it is not C++0x, it is just C++99.

Who said dynamic languages had no place in performance code – despite being extraordinarily slow in comparison to Java, huge tracts of applications are not performance critical, and Groovy is easily fast enough for those bits.

Acknowledgements

Thanks to Václav Pech and Paul King, fellow committers in the Groovy and GParc projects for various conversations and code examples that didn't make it explicitly into this article but helped shape what did get in. Also they gave splendidly valuable feedback on a draft of this article.

Postscript

The code in this article, and many other not dissimilar variants in many other languages, can be found in browseable form at <http://www.russel.org.uk:8080/SleepingBarber>. This is a Bazaar branch being rendered by Loggerhead. If you want to branch the branch using Bazaar then use the url <http://www.russel.org.uk/Bazaar/SleepingBarber>.

Endnote

If you want to get involved in testing or developing GParc, just get stuck in and/or ask on the mailing lists. If you want to do things surreptitiously, then just clone the GParc Git repository which is at <git://git.codehaus.org/gpars.git> and research from there. The browseable version is at <http://git.codehaus.org/gitweb.cgi?p=gpars.git>.