Guide To Communicating Sequential Processes

Russell Winder

Version 1.0, 2015-10-01

Table of Contents

The **CSP** (Communicating Sequential Processes) abstraction builds on independent composable processes, which exchange messages in a synchronous manner. GPars leverages the **JCSP** library developed at the University of Kent, UK.

Jon Kerridge, the author of the **CSP** implementation in **GPars**, provides exhaustive examples on of **GroovyCSP** use at www.soc.napier.ac.uk

Purpose

The **GroovyCSP** implementation leverages **JCSP**, a Java-based **CSP** library, which is licensed under LGPL. There are some differences between the Apache 2 license, which **GPars** uses, and LGPL. Please make sure your application conforms to the LGPL rules before enabling the use of **JCSP** in your code.

If the LGPL license is not adequate for your use, you might consider checking out the **Dataflow Concurrency** chapter of this **User Guide** to learn about *tasks* , *selectors* and *operators* , which may help you resolve concurrency issues in ways similar to the **CSP** approach. In fact, the dataflow and **CSP** concepts, as implemented in **GPars**, are very close to each other.

Apache 2 License

By default, without actively adding an explicit dependency on **JCSP** in your build file or downloading and including the **JCSP** jar file in your project, the standard commercial-software-friendly *Apache 2 License* terms apply to your project. **GPars** directly only depends on software licensed under licenses compatible with the *Apache 2 License*.

The CSP Model Principles

In essence, the **CSP** model builds on independent concurrent processes, which mutually communicate through channels using synchronous (i.e. rendezvous) message passing. Unlike actors or dataflow operators, which revolve around the event-processing pattern, **CSP** processes place the focus of their activities (aka sequences of steps) around the use of communications to remain mutually in sync along the way.

Since the addressing is indirect through channels, the processes do not need to know about one another. They typically consist of a set of input and output channels and a body. Once a **CSP** process is started, it obtains a thread from a thread pool and starts processing its body, pausing only when reading from a channel or writing into a channel. Some implementations (e.g. **GoLang**) can also detach the thread from the **CSP** process when blocked on a channel.

CSP programs are deterministic. The same data on the program's input will always generate the same output, irrespective of the actual thread-scheduling scheme used. This helps a lot when debugging **CSP** programs as well as analyzing deadlocks.

Determinism combined with indirect addressing results in a great level of composability of **CSP** processes. You can combine small **CSP** processes into bigger ones just by connecting their input and output channels and then wrapping them by another, bigger containing process.

The **CSP** model introduces non-determinism using *Alternatives*. A process can attempt to read a value from multiple channels at the same time through a construct called *Alternative* or *Select*. The first value that becomes available in any of the channels involved in the *Select* will be read and consumed by the process. Since the order of messages received through a *Select* depends on unpredictable conditions during program run-time, the value that will be read is non-deterministic.

CSP with GPars Dataflow

GPars provides all the necessary building blocks to create **CSP** processes.

- **CSP** Processes can be modelled through **GPars** tasks using a *Closure*, a *Runnable* or a *Callable* to hold the actual implementation of the process
- **CSP Channels** should be modelled with *SyncDataflowQueue* and *SyncDataflowBroadcast* classes
- **CSP Alternative** is provided through the *Select* class with its *select* and _ prioritySelect_ methods

---

Processes

To start a process, simply use the *task* factory method.

Start A Process

```
import groovyx.gpars.group.DefaultPGroup
import groovyx.gpars.scheduler.ResizeablePool

group = new DefaultPGroup(new ResizeablePool(true))

def t = group.task {
    println "I am a process"
}

t.join()
```

Since each process consumes a thread for its lifetime, it is advisable to use resizeable thread pools as in the example above.

A process can also be created from a **Runnable** or **Callable** object:

A Runnable Sample

```
import groovyx.gpars.group.DefaultPGroup
import groovyx.gpars.scheduler.ResizeablePool

group = new DefaultPGroup(new ResizeablePool(true))

class MyProcess implements Runnable {

    @Override
    void run() {
        println "I am a process"
    }
}
def t = group.task new MyProcess()

t.join()
```

Using **Callable** allows values to be returned through the *get()* method:

A Callable Sample

```
import groovyx.gpars.group.DefaultPGroup
import groovyx.gpars.scheduler.ResizeablePool

import java.util.concurrent.Callable

group = new DefaultPGroup(new ResizeablePool(true))

class MyProcess implements Callable<String> {

    @Override
    String call() {
        println "I am a process"
        return "CSP is great!"
    }
}

def t = group.task new MyProcess()

println t.get()
```

Channels

Processes typically need channels to communicate with their companion processes as well as with the outside world:

A Channel Sample

```
import groovy.transform.TupleConstructor
import groovyx.gpars.dataflow.DataflowReadChannel
import groovyx.gpars.dataflow.DataflowWriteChannel
import groovyx.gpars.group.DefaultPGroup
import groovyx.gpars.scheduler.ResizeablePool

import java.util.concurrent.Callable
import groovyx.gpars.dataflow.SyncDataflowQueue

group = new DefaultPGroup(new ResizeablePool(true))

@TupleConstructor
class Greeter implements Callable<String> {
    DataflowReadChannel names
    DataflowWriteChannel greetings

    @Override
    String call() {
        while(!Thread.currentThread().isInterrupted()) {
            String name = names.val
            greetings << "Hello " + name
        }
        return "CSP is great!"
    }
}

def a = new SyncDataflowQueue()
def b = new SyncDataflowQueue()

group.task new Greeter(a, b)

a << "Joe"
a << "Dave"
println b.val
println b.val
```

Which Delivery Technique To Use for Messages ?

The **CSP** model uses synchronous messaging, however, in GPars you may consider using asynchronous channels as well as synchronous ones.

You can also combine these two types of channels within the same process.

Composition

Grouping processes simply becomes a matter of connecting them with channels:

A Grouping Sample

```
group = new DefaultPGroup(new ResizeablePool(true))

@TupleConstructor
class Formatter implements Callable<String> {
    DataflowReadChannel rawNames
    DataflowWriteChannel formattedNames

    @Override
    String call() {
        while(!Thread.currentThread().isInterrupted()) {
            String name = rawNames.val
            formattedNames << name.toUpperCase()
        }
    }
}

@TupleConstructor
class Greeter implements Callable<String> {
    DataflowReadChannel names
    DataflowWriteChannel greetings

    @Override
    String call() {
        while(!Thread.currentThread().isInterrupted()) {
            String name = names.val
            greetings << "Hello " + name
        }
    }
}

def a = new SyncDataflowQueue()
def b = new SyncDataflowQueue()
def c = new SyncDataflowQueue()

group.task new Formatter(a, b)
group.task new Greeter(b, c)

a << "Joe"
a << "Dave"
println c.val
println c.val
```

Alternatives

To introduce non-determinist, **GPars** offers the *Select* class with its *select* and *prioritySelect* methods:

A Select Sample

```
import groovy.transform.TupleConstructor
import groovyx.gpars.dataflow.SyncDataflowQueue
import groovyx.gpars.dataflow.DataflowReadChannel
import groovyx.gpars.dataflow.DataflowWriteChannel
import groovyx.gpars.dataflow.Select
import groovyx.gpars.group.DefaultPGroup
import groovyx.gpars.scheduler.ResizeablePool

import static groovyx.gpars.dataflow.Dataflow.select

group = new DefaultPGroup(new ResizeablePool(true))

@TupleConstructor
class Receptionist implements Runnable {
    DataflowReadChannel emails
    DataflowReadChannel phoneCalls
    DataflowReadChannel tweets
    DataflowWriteChannel forwardedMessages

    private final Select incomingRequests = select([phoneCalls, emails, tweets])
//prioritySelect() would give highest precedence to phone calls

    @Override
    void run() {
        while(!Thread.currentThread().isInterrupted()) {
            String msg = incomingRequests.select()
            forwardedMessages << msg.toUpperCase()
        }
    }
}

def a = new SyncDataflowQueue()
def b = new SyncDataflowQueue()
def c = new SyncDataflowQueue()
def d = new SyncDataflowQueue()

group.task new Receptionist(a, b, c, d)

a << "my email"
b << "my phone call"
c << "my tweet"
```

```
//The values come in random order since the process uses a Select to read its input
3.times{
    println d.val.value
}
```

Components

CSP processes can be composed into larger entities. Suppose you already have a set of **CSP** processes (aka Runnable/Callable classes), you can compose them into a larger process:

A Larger Sample

```
final class Prefix implements Callable {
    private final DataflowChannel inChannel
    private final DataflowChannel outChannel
    private final def prefix

    def Prefix(final inChannel, final outChannel, final prefix) {
        this.inChannel = inChannel;
        this.outChannel = outChannel;
        this.prefix = prefix
    }

    public def call() {
        outChannel << prefix
        while (true) {
            sleep 200
            outChannel << inChannel.val
        }
    }
}
```

Another Building Block

```
final class Copy implements Callable {
    private final DataflowChannel inChannel
    private final DataflowChannel outChannel1
    private final DataflowChannel outChannel2

    def Copy(final inChannel, final outChannel1, final outChannel2) {
        this.inChannel = inChannel;
        this.outChannel1 = outChannel1;
        this.outChannel2 = outChannel2;
    }

    public def call() {
        final PGroup group = Dataflow.retrieveCurrentDFPGroup()
        while (true) {
            def i = inChannel.val
            group.task {
                outChannel1 << i
                outChannel2 << i
            }.join()
        }
    }
}
```

A Sample

```
import groovyx.gpars.dataflow.DataflowChannel
import groovyx.gpars.dataflow.SyncDataflowQueue
import groovyx.gpars.group.DefaultPGroup

group = new DefaultPGroup(6)

def fib(DataflowChannel out) {
    group.task {
        def a = new SyncDataflowQueue()
        def b = new SyncDataflowQueue()
        def c = new SyncDataflowQueue()
        def d = new SyncDataflowQueue()
        [new Prefix(d, a, 0L), new Prefix(c, d, 1L), new Copy(a, b, out), new StatePairs
(b, c)].each { group.task it}
    }
}

final SyncDataflowQueue ch = new SyncDataflowQueue()
group.task new Print('Fibonacci numbers', ch)
fib(ch)

sleep 10000
```