UNIVERSITÀ DI PADOVA  FACOLTÀ DI INGEGNERIA

TESI DI LAUREA

GPARS: un ambiente evoluto per la programmazione concorrente in Java/Groovy

Laureando: Limena Claudio

Relatore: ing. Moro Michele

Corso di laurea Vecchio Ordinamento in Ingegneria Informatica

17/04/2012

A.A. 2011-2012

Indice generale

Abstract

La programmazione parallela è comunemente considerata un argomento difficile da affrontare, GPars mette a disposizione del programmatore Java o Groovy un insieme di astrazioni ad alto livello per la gestione della concorrenza in un modo che le rende facili da utilizzare.
GPars è un ambiente strutturato in modo tale da essere sufficientemente robusto per l'implementazione di sistemi estremamente complessi, prestandosi allo stesso tempo alla realizzazione di semplici sistemi software destinati al mercato mainstream.

Introduzione

L'avvento dei chip multicore ha reso onnipresenti i sistemi multiprocessore, il numero di thread per chip e il numero di chip multicore per sistema multiprocessore continua ad aumentare; tutti noi eseguiamo il nostro software su sistemi che sono, in definitiva, multiprocessore, probabilmente il codice scritto oggi o domani non verrà mai eseguito su un sistema a singolo processore.

Al giorno d'oggi il concetto di core, come entità a se stante, sta perdendo di significato grazie all'apparizione sul mercato mainstream di processori multicore con strutture ibride[1] e di sistemi per l'accelerazione del calcolo parallelo[2].
Mentre l'hardware parallelo è diventato di uso comune, questo non è ancora il caso del software; la comunità dei programmatori crea ancora programmi a singolo thread che non saranno mai in grado di sfruttare appieno l'hardware odierno o futuro.

Applicazioni scalabili, cioè applicazioni in grado di sfruttare efficientemente un numero crescente thread hardware mano a mano che questi si rendono disponibili, sono difficili da costruire a causa della necessità di fare dei compromessi tra performance, scalabilità e complessità.
Tradizionali metodi di programmazione basati su lock sono troppo difficili da utilizzare e troppo propensi a generare errori per supportare lo sviluppo su larga scala di applicazioni scalabili; è diventato ovvio che metodologie basate su multithreading con memoria condivisa generano più problemi di quelli che risolvono.

Con un'evoluzione così radicale dell'hardware il software deve evolversi di conseguenza; concetti astratti di alto livello per sistemi concorrenti, come map/reduce, fork/join, attori, agenti, dataflow o STM, permettono di risolvere elegantemente problemi complessi sfruttando l'hardware parallelo disponibile in modo trasparente per il programmatore.
Si tratta di concetti che esistono da diverso tempo, dato che sistemi paralleli sono diventati di uso comune nell'industria e che ancora non lo sono nel mercato mainstream.
É giunta l'ora di adottare questi strumenti anche nella realizzazione del software mainstream.

1 Pensiamo ad architetture tipo i core bulldozer di amd o a chip che integrano un core “classico” e una gpu.
2 Ad esempio, Nvidia Tesla.

1) Groovy

Groovy[1] è un linguaggio di programmazione orientato ad oggetti compatibile con la piattaforma Java; presenta funzionalità simili a quelle di Pyton, Ruby e Smalltalk.
Si tratta di un linguaggio di programmazione agile e dinamico; viene compilato in bytecode per la Java Virtual Machine (JVM) e può interoperare con il codice e le librerie proprie di Java; usa una sintassi simile a quella usata in Java per cui la maggior parte del codice scritto in Java è sintatticamente corretta in Groovy.
Per quanto Groovy e Java siano piuttosto simili, Groovy permette di produrre codice che risulta molto più compatto rispetto allo stesso codice scritto in Java ed è per questo considerato da alcuni un linguaggio di scripting.

Ora qualcuno si chiederà: ma esistono già svariati linguaggi compatibili con la JVM, compreso lo stesso Java, perché bisogna crearne un altro ?
Innanzitutto si può dire che Java è un linguaggio di programmazione logorroico e di non facile utilizzo, che si compila in un bytecode che viene a sua volta interpretato dalla JVM; da qui nasce la proliferazione di linguaggi alternativi: affinché un linguaggio sia alternativo a Java, è sufficiente che si possa compilare in Java bytecode.
Tra le varie possibili alternative a Java, viene in mente un linguaggio di programmazione dinamico ed estremamente agile chiamato Scala.

Groovy nacque nel 2003; il suo creatore, James Strachan, scrisse nel 2009 nel suo blog, parlando dell'evoluzione (o meglio, assenza della medesima) di Java:

> "So whats gonna be the long term replacement for javac? Certainly the dynamic languages like Ruby, Groovy, Python, JavaScript have been getting very popular the last few years - lots of folks like them.
>
> Though my tip though for the long term replacement of javac is Scala. I'm very impressed with it! I can honestly say if someone had shown me the Programming in Scala book by by Martin Odersky, Lex Spoon & Bill Venners back in 2003 I'd probably have never created Groovy. "[3]

Premesso questo, qualcuno potrebbe chiedersi: ma allora, perché esiste ancora Groovy ?

Le risposte possono essere le più svariate ma, per quanto Groovy e Scala siano entrambi linguaggi di programmazione orientati agli oggetti e siano entrambi interoperabili con Java, tra essi esistono diverse differenze, vediamone alcune:

Groovy è un linguaggio dinamico, non solo nel senso che supporta i tipi di dati dinamici ma nel senso che supporta anche la metaprogrammazione dinamica.
Scala è un linguaggio di programmazione statico, nel senso che usa tipi statici di dati e che non permette la metaprogrammazione dinamica a parte quella maldestramente permessa da Java.

Groovy è sintatticamente influenzato da Java ma semanticamente più influenzato da altri linguaggi come Ruby.
Scala è sintatticamente influenzato sia da Ruby che da Java, è semanticamente influenzato da Java, SML, Haskell e da un linguaggio orientato ad oggetti poco conosciuto chiamato gBeta.

3 Estratto da [2]

Groovy supporta il multiple dispatch[4] accidentalmente a causa di come viene gestito l'overloading di Java.
Scala supporta solo il dispatch singolo, ma implementa un pattern matching ispirato a SML che permette di gestire la stessa categoria di problemi gestiti dal multiple dispatch; tuttavia, mentre il multiple dispatch può basarsi solo sui tipi di dati a runtime, il pattern matching di Scala può basarsi sui tipi o sui valori dei dati o su entrambi.

Scala supporta nativamente la chiamata parziale di un metodo[5] sia il currying[6]; Groovy supporta goffamente la chiamata parziale di una funzione mediante l'uso di un metodo specifico (curry).

Scala usa la direct tail call recursion optimization[7]. Groovy non ne forza l'uso, tuttavia questa metodologia è molto più importante in functional programming che in imperative programming (intesi come paradigmi di programmazione) perché il secondo si basa sul cambiamento di stato degli oggetti e il primo no.

Sia in Scala che in Groovy, quando ad un'espressione viene assegnato un dato, l'espressione viene valutata subito. Tuttavia, Scala supporta parametri call-by-name[8] e Groovy no, call-by-name deve essere emulata usando le chiusure.

Scala non ha alcun concetto di campi statici, metodi interni ad una classe etc, al posto di questo usa degli oggetti particolari chiamati singleton.
Groovy implementa il concetto di staticità.

Scala non implementa una selezione di operatori aritmetici come fa Groovy.

---

4 Il multiple dispatch[3] è un meccanismo in cui, in presenza di più metodi definiti con lo stesso nome ma con argomenti diversi, il metodo corretto da invocare è scelto a runtime in base al tipo assunto, sempre a runtime, dagli argomenti facenti parte della chiamata al metodo. Questo è possibile nei linguaggi dinamici.

5 Chiamata parziale di un metodo[5]: dato un metodo che richiede in ingresso una n-pla di argomenti, consiste nel fissare i valori di solo alcuni di questi (ad esempio x argomenti), trattando il metodo così ottenuto come una funzione dei restanti m=n-x argomenti.

6 Currying[4]: si tratta di una tecnica che permette la trasformazione di una funzione (o un metodo in programmazione) che richiede una n-pla di argomenti in modo che possa essere chiamata come una catena di n metodi che richiedono in ingresso un argomento ciascuno.

7 Intraducibile in modo comprensibile.
La ricorsione si dice diretta quando una funzione chiama ricorsivamente se stessa, l'esempio tipico è la funzione che calcola il fattoriale di un numero come f(x)=x*f(x-1).
Quando si chiama un metodo, il computer deve ricordare il punto da cui il metodo viene chiamato, il punto (o l'indirizzo) di ritorno, in modo da poter riprendere l'esecuzione della procedura chiamante tornando a quel punto con il risultato del metodo chiamato. Questa informazione è tipicamente memorizzata in uno stack.
Una tail call[6] consiste nella chiamata ad un metodo A che il metodo chiamante B esegue subito prima del suo completamento. Se il metodo B è stato a sua volta creato da una chiamata eseguita da un terzo metodo C, questo permette di non salvare nello stack il punto di ritorno da A in B, ma di riciclare il punto di ritorno in C risparmiando spazio nello stack (perché B terminerebbe subito dopo e si tornerebbe in C in ogni caso).
Nel caso si utilizzino algoritmi che implementano una ricorsione profonda, l'applicazione di questa tecnica permette un risparmio di tempo di esecuzione e di memoria considerevole.

8 Call-by-name: è una tecnica di valutazione degli argomenti di una funzione, consiste nel non valutare gli argomenti di un metodo prima che questo sia chiamato ma nel sostituirli direttamente nel corpo della funzione valutandoli mano a mano che vengono incontrati, questo fa si che se un argomento non viene incontrato nel computo di un metodo, l'argomento non venga mai valutato; tuttavia se lo stesso argomento viene incontrato più volte verrà valutato più volte; per maggiori informazioni, si veda [7].
Dato che la tecnica call-by-value prevede la valutazione di tutti gli argomenti di un metodo indipendentemente dal fatto che siano poi utilizzati o meno, la tecnica call-by-value, nonostante richieda generalmente un tempo di computazione maggiore, può permettere un notevole risparmio in tempo se una buona parte degli argomenti di un metodo solitamente non vengono valutati.

Si può scegliere con molta flessibilità i nomi dei metodi in Scala.
Groovy usa l'operatore ?. per gestire i riferimenti a null. I programmatori che usano Scala preferiscono usare il tipo option, ma è molto semplice creare in Scala un operatore come quello che usa Groovy.

1.1) Approdare a Groovy salpando da Java

Nel corso degli anni è diventata prassi comune scrivere, come primo codice creato in un nuovo linguaggio di programmazione, un programma che stampi a monitor la stringa “hello world”.
Vediamo come sia possibile scrivere tale programma in Groovy.

```
//HelloWorld in Java
public class HelloWorld {
   String name;

   public void setName(String name)
   { this.name = name; }
   public String getName(){ return name; }

   public String greet()
   { return "Hello "+ name; }

   public static void main(String args[]){
      HelloWorld helloWorld = new HelloWorld();
      helloWorld.setName("Java");
      System.out.println( helloWorld.greet() );
   }
}

//HelloWorld in Groovy
public class HelloWorld {
   String name;

   public void setName(String name)
   { this.name = name; }
   public String getName(){ return name; }

   public String greet()
   { return "Hello "+ name; }

   public static void main(String args[]){
      HelloWorld helloWorld = new HelloWorld();
      helloWorld.setName("Groovy");
      System.out.println( helloWorld.greet() );
   }
}
```

E no, non si tratta di uno scherzo.
Groovy e Java sono parenti stretti, la loro sintassi è molto simile e può essere potenzialmente identica cosicché la maggioranza dei programmi scritti in Java sono validi programmi scritti in Groovy: nella stragrande maggioranza dei casi per trasformare del codice da Java a Groovy è sufficiente rinominare il file *.java in cui è contenuto in *.groovy.
Come si può intuire, la curva di apprendimento minima necessaria per passare da Java a Groovy ricorda molto l'universo di Flatlandia[9].

9 Si veda [8]

Giunti a questo punto, potrebbe sorgere spontanea la domanda: ma allora, perché esiste Groovy ?
Perché Java permette la stessa agilità permessa da un elefante; vediamo come è possibile "groovizzare" il codice di hello world.

Primo passo: eliminiamo il rumore

- In Groovy è tutto pubblico se non diversamente specificato.
- Il punto e virgola alla fine di una sentenza è opzionale.

```
class HelloWorld {
   String name

   void setName(String name)
   { this.name = name }
   String getName(){ return name }

   String greet()
   { return "Hello "+ name }

   static void main(String args[]){
      HelloWorld helloWorld = new HelloWorld()
      helloWorld.setName("Groovy")
      System.out.println( helloWorld.greet() )
   }
}
```

In Groovy è presente un solo caso in cui è obbligatorio l'uso di un ; come terminazione di una sentenza:

```
class Trial {
  private final thing = new Thing ( )
  { thing.doSomething ( ) }
}
```

Per quanto questo possa sembrare assurdo, questo codice lancerà una `MissingMethodException` perché verrà interpretato come tentativo di passare una chiusura al costruttore di `Thing`[10].

Versione corretta:

```
class Trial {
  private final thing = new Thing ( ) ;
  { thing.doSomething ( ) }
}
```

Secondo passo: eliminiamo le parti prolisse

- Creare un JavaBean richiede fornire una coppia get/set per ogni attributo, questo è noto a tutti, compreso Groovy: permettiamogli di scriverla per noi.
- Il main() richiede sempre un string[] come parametro, sostituiamolo nella definizione del metodo con un parametro opzionale.
- Stampare a consolle è un'attività così comune, non è possibile averne una versione abbreviata ?

```
class HelloWorld {
```

10 Esistono altri problemi in cui si può incorrere convertendo con troppa leggerezza del codice Java in Groovy, un elenco è riportato in [9]

```
   String name

   String greet()
   { return "Hello "+ name }

   static void main( args ){
      HelloWorld helloWorld = new HelloWorld()
      helloWorld.setName("Groovy")
      println( helloWorld.greet() )
   }
}
```

Terzo passo: introduciamo i tipi di dati dinamici

- Si utilizzi la parola chiave **def** quando non si vuole specificare il tipo di una variabile.
- Groovy ne identificherà correttamente il tipo; questo è chiamato duck typing[11]

```
class HelloWorld {
   String name

   def greet()
   { return "Hello "+ name }

   static def main( args ){
      def helloWorld = new HelloWorld()
      helloWorld.setName("Groovy")
      println( helloWorld.greet() )
   }
}
```

Quarto passo: uso di Variable Interpolation

- Groovy supporta il Variable Interpolation[12] attraverso le Gstring.
- Funziona come in altri linguaggi (Perl, PHP, Ruby etc).
- Permette di inserire in una stringa una qualsiasi espressione scritta in Groovy utilizzando il costrutto ${}.

```
class HelloWorld {
   String name

   def greet(){ return "Hello ${name}" }

   static def main( args ){
      def helloWorld = new HelloWorld()
      helloWorld.setName("Groovy")
      println( helloWorld.greet() )
   }
}
```

Quinto passo: eliminiamo delle altre parole chiavi ridondanti

- La parola chiave **return** è opzionale[13], il valore di ritorno sarà quello dell'ultima espressione valutata.

---

11 Si veda [10]

12 Si veda [11]

13 Per quanto l'uso della parola chiave return sia opzionale, questo può comportare una riduzione nella leggibilità del codice.
Sta al singolo programmatore, o al gruppo di lavoro di cui fa parte, la scelta di quali, tra i costrutti opzionali offerti da Groovy, utilizzare nella realizzazione di un'applicazione.

- Non è necessario utilizzare la parola chiave **def** nei metodi statici

```
class HelloWorld {
   String name

   def greet(){ "Hello ${name}" }

   static main( args ){
      def helloWorld = new HelloWorld()
      helloWorld.setName("Groovy")
      println( helloWorld.greet() )
   }
}
```

Sesto passo: POJOs sotto steroidi

- I POJO[14] (POGO in Groovy) non solo forniscono i metodi per l'accesso alle loro proprietà ma forniscono anche un costruttore di default con parametri nominativi.
- I POGO supportano l'accesso ai propri parametri con le notazioni bean[prop] e bean.prop.

```
class HelloWorld {
   String name

   def greet(){ "Hello ${name}" }

   static main( args ){
      def helloWorld = new
               HelloWorld(name:"Groovy")
      helloWorld.name = "Groovy"          //inizializzazione alternativa nome
      helloWorld["name"] = "Groovy"       //inizializzazione alternativa nome
      println( helloWorld.greet() )
   }
}
```

Settimo passo: Groovy supporta gli script

- Per quanto codice Groovy venga compilato in byte code Java, Groovy supporta gli script che, a loro volta, verranno compilati in byte code Java.
- É possibile definire una classe ovunque in uno script.
- I package sono supportati negli script, dopotutto il contenuto di un package può essere importato come una qualsiasi altra classe Java.

```
class HelloWorld {
   String name
   def greet() { "Hello $name" }
}

def helloWorld = new HelloWorld(name:"Groovy")
println helloWorld.greet()
```

Questo conclude la groovizzazione del codice di hello world.
Si ricorda che siamo partiti da qui:

```
public class HelloWorld {
   String name;

   public void setName(String name)
```

14 Si veda [12]

```
   { this.name = name; }
   public String getName(){ return name; }

   public String greet()
   { return "Hello "+ name; }

   public static void main(String args[]){
      HelloWorld helloWorld = new HelloWorld()
      helloWorld.setName("Groovy")
      System.err.println( helloWorld.greet() )
   }
}
```

e che in tutto il processo di conversione sono stati utilizzati esclusivamente costrutti opzionali.

1.2) Concetti fondamentali[15]

In Groovy, tutti i tipi di dati sono oggetti

Non esistono tipi primitivi; chiamate del genere sono perfettamente legittime:

```
3.times {
   println "hello"
}
```

3 è un intero, ma in Groovy è un oggetto, su cui viene chiamato un metodo che a sua volta riceve come parametro una chiusura; il risultato finale viene interpretato come "3 volte stampa hello".

Chiusure

Le chiusure possono essere viste come blocchi di codice riutilizzabili, simili a metodi, che sostituiscono le classi interne (o private) nella stragrande maggioranza dei casi.
Una chiusura, a differenza di un metodo, può essere dichiarata ovunque: non deve necessariamente essere dichiarata all'interno di una classe; una chiusura assomiglia ad un metodo nel senso che può accettare dei parametri (passati con l'operatore `->`) che possono essere a loro volta delle chiusure e che può ritornare valori.
Una chiusura avrà un parametro di default chiamato **it** se non ne viene definito uno.
Una chiusura è un oggetto di tipo `groovy.lang.Closure`, è costituito da un blocco di codice racchiuso tra parentesi graffe che può essere restituito come risultato di qualcosa, considerato come variabile e in genere trattato come qualunque altro oggetto.
Una chiusura viene eseguita quando viene chiamata e non quando viene definita.

Creazione:

```
//senza parametri
Closure simpleCloj1 = {
println 'Hello, World!'
}

//con un parametro non tipizzato
def simpleCloj2 = { obj ->
println "Hello, $obj!"
}

//con un parametro di tipo stringa
def simpleCloj3 = { String obj ->
println "Hello, $obj!"
```

15 La documentazione relativa a Groovy è disponibile in [13]; un'introduzione a Groovy è disponibile in [14]

```
}

/*Se la chiusura accetta un solo parametro, questo può essere omesso dalla
 *definizione e l'accesso avverrà tramite la parola chiave it*/
def simpleCloj4 = {
println "Hello, $it!"
}

//accetta più parametri
def twoParamsCloj = { obj1, obj2 ->
println "$obj1, $obj2!"
}
```

Chiamata:
Una chiusura può essere chiamata in tre modi : closure(), closure.call() o closure.doCall().

```
simpleCloj1()
simpleCloj2.call('World')
simpleCloj4.doCall('World')
twoParamsCloj('Hello', 'World')
```

Currying
Il currying è una tecnica di programmazione che trasforma una funzione in un'altra funzione fissando alcuni attributi della funzione originaria (si pensi a costanti).
Utilizzando il currying è possibile anche trasformare una funzione che richiede una n-pla di argomenti in una sequenza di funzioni innestate che richieda in totale una n-pla di argomenti.

```
// Una chiusura con tre parametri, il terzo è opzionale perché definisce un
// valore di default
def getSlope = { x, y, b = 0 ->
   println "x:${x} y:${y} b:${b}"
   (y - b) / x
}

/**
normale chiamata a getSlope(), i valori passata agli attributi sono x=2, y=2 e
b=0 (per default)
**/
assert 1 == getSlope( 2, 2 )

/**
Definizione di una nuova chiusura, getSplopeX, ottenuta assegnando un valore,
nell'esempio 5, al primo parametro della chiusura getSlope().
La chiusura ottenuta è diventata una "funzione" in due parametri: y che deve
essere assegnato dall'esterno e b, con valore di default 0.
**/
def getSlopeX = getSlope.curry(5)
assert 1 == getSlopeX(5)
assert 0 == getSlopeX(2.5,2.5)
// Output
// x:2 y:2 b:0
// x:5 y:5 b:0
// x:5 y:2.5 b:2.5
```

Iteratori
Così come in Ruby, anche in Groovy è possibile utilizzare gli iteratori in praticamente qualunque contesto, sarà Groovy stesso a decidere cosa fare in ogni singolo caso.
Gli iteratori permettono di eliminare completamente il costrutto di ciclo, qualora si debba utilizzare

un ciclo per iterare su una collezione.
Tutti gli iteratori accettano una chiusura come parametro, ciò permette, ad esempio, di applicare una chiusura ad ogni elemento di una collezione.
Gli iteratori vengono anche utilizzati per sostituire costrutti come i cicli for.

```
def printIt = { println it }
// 3 modi per iterare da 1 a 5
[1,2,3,4,5].each printIt
1.upto 5, printIt
(1..5).each printIt

// In confronto ad un normale ciclo
for( i in [1,2,3,4,5] ) printIt(i)
// O, in alternativa
for( i in (1..5) ) printIt(i)

[1,2,3,4,5].eachWithIndex { v, i -> println "list[$i] => $v" }
// list[0] => 1
// list[1] => 2
// list[2] => 3
// list[3] => 4
// list[4] => 5
```

Nuovi operatori

- ?: (elvis) – simile all'operatore ternario di Java
  In questo esempio si vuole utilizzare un chatName, se impostato, altrimenti si vuole impostare chatName al valore “Anonymous”

  ```
  String chatName =       user.chatName ?: 'Anonymous' // In Groovy

  String chatName =       user.chatName != null ?
                          user.chatName : "Anonymous"; // In Java
  ```

- ?. – Safe navigation, permette di controllare l'esistenza di un oggetto prima di accedervi per qualunque ragione.

  ```
  user?.doSomething() // In Groovy

  if(user != null)
     user.doSomething(); // In Java
  ```

- <=> (spaceship) – confronta due valori
- * (spread) – “esplode” il contenuto di una lista o di un array
- *. (spread-dot) – applica un metodo su ogni elemento di una lista o di un array

Metaprogrammazione
Permette di aggiungere metodi e attributi ad ogni oggetto a runtime.
Permette di intercettare chiamate a metodi e/o accessi ad attributi.
Permette di modificare in genere il comportamento di un oggetto.
Questo significa che Groovy offre una struttura simile al concetto di classe aperta di Ruby.

Un esempio usando le categories

```
class Pouncer {
   static pounce( Integer self ){
      def s = "Boing!"
```

```
      1.upto(self-1) { s += " boing!" }
      s + "!"
   }
}

use( Pouncer ){
   assert 3.pounce() == "Boing! boing! boing!"
}
```

Stesso esempio, usando le metaclassi

```
Integer.metaClass.pounce << { ->
   def s = "Boing!"
   delegate.upto(delegate-1) { s += " boing!" }
   s + "!"
}

assert 3.pounce() == "Boing! boing! boing!"
```

Groovy Truth

Allo scopo di valutare una condizione, Java impone al programmatore di fornire un'espressione booleana che esprima la condizione (magari contenuta in un costrutto if); Groovy è più dinamico ed utilizza una sintassi più espressiva: a seconda del contesto Groovy valuterà un'espressione del tipo null, stringa vuota, "", e zero come falso.

Si supponga di voler eseguire una chiusura se una stringa `str` sconosciuta contiene dei caratteri in modo da poterla utilizzare senza problemi all'interno della chiusura:

```
String str = ... // Sconosciuta
/* Esegui la chiusura se Str contiene dei caratteri */
if( str ) {...} // In Groovy
if( str != null && !str.isEmpty() ) {...} // In Java
```

Certamente, i due approcci sono equivalenti, ma la versione scritta in Groovy è più leggibile e più facile da comprendere.

Tabella che esprime la Groovy truth:

Espressione da valutare	**Condizione da verificare**
Valore booleano	VERO
Collezione	Non vuota
Carattere	Valore non zero
Sequenza di caratteri	Lunghezza maggiore di zero
Enumerazione	Ha altri elementi
Iteratore	Esiste il prossimo elemento
Numero	Double, valore non zero
Map	Non vuota
Matcher	Ha almeno un match
Oggetto	Lunghezza non nulla
Ogni altro tipo	Non nullo

GString[15]
Stringhe contenute tra `"string"` o tra `"""string"""` possono contenere delle espressioni arbitrarie in blocchi del tipo `${espressione}`.
Ogni tipo di espressione valida in Groovy può essere contenuta in un blocco `${espressione}`, chiamate a metodi incluse.
Una GString è definita come una stringa viene definita in Java, ecco un semplice esempio:

```
foxtype = 'quick'
foxcolor = ['b', 'r', 'o', 'w', 'n']
println "The $foxtype ${foxcolor.join()} fox"
// => The quick brown fox
```

Quello che accade è che, quando una stringa contiene un'espressione `${espressione}`, viene creato un oggetto GString contenente il testo e i valori delle espressioni al posto di una normale stringa.
Le espressioni contenute in un oggetto GString vengono valutate quando l'oggetto viene stampato, permettendo la lazy evaluation.

2) GPARS

Tradizionalmente la concorrenza in un linguaggio di programmazione è introdotta con il supporto diretto ai thread.
In questo modello, l'esecuzione di un programma è suddivisa in task in esecuzione concorrente: è come se più copie dello stesso programma venissero avviate contemporaneamente, con la differenza che tutte queste istanze operano su un'area di memoria condivisa.
La condivisione della memoria introduce una serie di problemi di difficile individuazione, i due più comuni sono il lost update e il deadlock.

Supponiamo ora che due processi stiano cercando di incrementare il valore di un oggetto condiviso acc: entrambi leggono il valore dell'oggetto, lo incrementano e aggiornano il valore memorizzato nell'oggetto.
Dato che queste operazioni non sono atomiche, è possibile che i comandi nelle due sequenze vengano eseguiti in modo intervallato, culminando in un aggiornamento errato di acc.
La soluzione a questo problema, noto come lost update, consiste nell'uso di uno strumento chiamato Lock.
Il lock permette la mutua esclusione: in ogni istante temporale, solo un processo può aver acquisito il lock su un determinato oggetto e nessun altro processo potrà accedervi.
L'utilizzo del lock permette ad un processo di acquisire il controllo assoluto su un oggetto, impedendo al lost update di verificarsi; tuttavia il lock introduce tutta una serie di altri problemi, il più noto dei quali è il deadlock.

Il deadlock è una situazione in cui due o più processi stiano tentando di assumere il controllo di un insieme di risorse, almeno in parte comuni ai vari processi, e in cui ogni processo riesca ad assumere il controllo di solamente una parte delle risorse e risulta impossibilitato nel proseguimento della sua esecuzione.
Così facendo, ogni processo attende indefinitamente il liberarsi delle risorse acquisite dal processo o dai processi concorrenti, creando una situazione di stallo che persiste indefinitamente.

Esistono vari metodi per evitare di incorrere in questi e in altri problemi derivanti dall'utilizzo diretto di thread e lock, alcuni sono riportati più avanti; tuttavia il metodo più banale, anche se può sembrare il meno ovvio, è quello di evitare di gestire la concorrenza in modo “artigianale”.

Lost Update Problem

Process 1	Process 2
`a = acc.get()`	
`a = a + 100`	`b = acc.get()`
	`b = b + 50`
	`acc.set(b)`
`acc.set(a)`	

Deadlock Problem

Process 1	Process 2
`lock(A)`	`lock(B)`
`lock(B)`	`lock(A)`
... Deadlock! ...	

GPars[16] è una libreria che permette di introdurre una serie di concetti astratti di alto livello per la gestione della concorrenza in Groovy o in Java; fornisce varie astrazioni a supporto della parallelizzazione del codice, spesso complementari tra loro, che permettono sia di parallelizzare parte del codice esistente, sia di creare codice parallelo ex novo.
Come scegliere, dunque, la o le astrazioni da usare ?

16 Homepage del progetto disponibile in [16]

Ovviamente in base al problema da affrontare:

1) Se ci si trova di fronte ad una collezione da processare in cui l'elaborazione di ciascun elemento possa avvenire indipendentemente dagli altri, è consigliato l'uso dei metodi di GPars relativi alla gestione degli insiemi, come each(), collect(), find(), etc.
2) Se si prevede una computazione di lunga durata, che possa avvenire in sicurezza in background, ci si può appoggiare all'asynchronous invocation support di GPars.
3) Si ha la necessità di parallelizzare un algoritmo già in uso.
   Se si possono identificare dei sub-task che possono essere eseguiti in parallelo e se si può fornire un metodo per lo scambio di dati tra questi in momenti ben definiti usando dei canali di comunicazione con una sintassi sicura, allora si può creare un insieme di task da eseguire concorrentemente usando i dataflow, tasks e streams di GPars.
4) Non si può evitare la condivisione di risorse: più thread accederanno a dei dati condivisi e alcuni di questi li modificheranno.
   L'approccio tradizionale basato su lock e sincronizzazione è troppo rischioso o poco familiare; GPars mette a disposizione gli agenti, che ingloberanno le risorse condivise e si occuperanno di tutta la sincronizzazione necessaria.
5) Si sta costruendo un'applicazione che esige un alto livello di concorrenza.
   Modificare una struttura dati qua o un task là non sarà sicuramente sufficiente, sarà necessario progettare l'architettura dall'inizio tenendo presente la concorrenza.
   La via da seguire potrebbe essere il message-passing.
   a) Groovy CSP fornirà un modello altamente deterministico e modulare per la programmazione concorrente.
   b) Se si sta cercando di risolvere un problema complesso di data-processing, si può considerare l'uso dei dataflow operator per costruire una data flow network.
   c) Gli attori brilleranno se si vuole costruire un'architettura scalabile, altamente concorrente e general-purpose.

Questi sono solo esempi: si possono costruire delle architetture mescolando a piacimento le astrazioni fornite da GPars, senza essere vincolati in nessun modo.

3) Le promesse di cui fidarsi

La decomposizione dei task è uno dei modi più intuitivi per introdurre la concorrenza.
Si possono definire diversi task o processi o thread indipendenti, dividere il lavoro da eseguire acciocché ogni task ne riceva una parte e poi lasciarli semplicemente lavorare concorrentemente.
GPars fornisce diverse tecniche per la creazione di attività asincrone: Dataflow tasks, asynchronous functions e active objects per nominare i concetti che usano maggiormente le promesse.

Ora, quando si hanno delle computazioni in esecuzione, si ha la necessità di coordinarle, di monitorarle e eventualmente anche di recuperare i loro risultati combinandoli.
Questo è quello che il concetto di promessa risolve elegantemente.
Il concetto di promessa è usato in molti linguaggi e framework orientati alla concorrenza, Akka, Clojure e Dart sono solo degli esempi; GPars supporta il concetto di promessa implementandolo tramite le dataflow variables.

Il concetto di promessa introduce la possibilità di disporre di segnaposti invece di variabili: se si sta avviando un task asincrono che eventualmente calcolerà un risultato si avrà la necessità di disporre di qualcosa di tangibile mentre il task sta lavorando in background, qualcosa che permetta di verificare lo stato del task o di recuperare il risultato della computazione, qualora sia terminata: la promessa di un risultato futuro.

La delicata gestione dei risultati

Quando una funzione asincrona, un task o un active object restituisce una promessa al posto di un risultato concreto, la promessa in sé rappresenta un handler alla computazione asincrona in esecuzione.
Un thread utente può ora far valere la promessa ricevuta da un servizio e, chiamando get(), bloccarsi fino a che un valore rappresentante il risultato della computazione sia reso disponibile.

```
Promise bookingPromise = task {
    final data = collectData()
    return broker.makeBooking(data)
}

//...qualche tempo dopo

printAgenda bookingPromise.get()
```

Se il concetto di promessa sembra familiare, probabilmente è perché si è utilizzata la classe java.util.concurrent.Future in precedenza; si tratta di due concetti molto simili ma separati da una fondamentale differenza: una promessa, così com'è intesa in GPars, permette di attendere un risultato futuro senza bloccare il thread corrente, è solo necessario creare un handler che verrà invocato quando il valore promesso si renderà disponibile.
Ovviamente niente impedisce di avere più handlers per la stessa promessa, scatteranno tutti in parallelo quando la promessa assumerà un valore concreto.

```
Promise bookingPromise = task {
    final data = collectData()
    return broker.makeBooking(data)
}
```

```
bookingPromise.whenBound {booking -> printAgenda booking}
bookingPromise.whenBound {booking -> sendMeAnEmail booking}
bookingPromise.whenBound {booking -> updateTheCalendar booking}
```

Tutto questo apre un insieme di nuove possibilità come la concatenazione, composizione e il raggruppamento di promesse.
Più promesse collegate alla stessa computazione asincrona aiutano nella scrittura di applicazioni altamente concorrenti che non richiedono mai più thread di quante siano le computazioni effettivamente in esecuzione.
I thread di sistema vengono raramente bloccati e spostati in uno stato passivo o di parcheggio; vengono invece riutilizzati quando handlers diversi diventano disponibili per l'esecuzione.

La catena delle responsabilità

A partire dalla versione 1.0[17], GPars permette di concatenare i risultati di operazioni asincrone usando il metodo then(); permette, cioè, di creare funzioni utilizzando delle promesse come parametri, gestendo questo senza bloccare il thread in esecuzione mentre si attende la disponibilità dei valori promessi.

```
//Alcuni servizi asincroni da utilizzare
final polish = ...
final transform = ...
final save = ...
final notify = ...
Promise promiseForStuff = task {
    loadStuffFromDB()
}
promiseForStuff.then polish then transform then save then {notify me}
```

Utilizzando il metodo then() è possibile concatenare più servizi sincroni o asincroni permettendone l'esecuzione asincrona dal thread principale; i risultati verranno passati automaticamente lungo la catena di servizi senza consumare threads mentre aspettano il loro input.
In questo modo è possibile raggruppare servizi asincroni senza sforzo.

Quando tutte le promesse diventano realtà

Non è sempre possibile costruire catene di promesse, a volte è necessario disporre di tutti i risultati promesse per poter proseguire nella computazione; in questi casi la funzione whenAllBound() si dimostra estremamente utile.

```
Promise module1 = task {
    compile(module1Sources)
}
Promise module2 = task {
    compile(module2Sources)
}
final jarCompiledModules = {List modules ->  //comprime i moduli in un jar
whenAllBound([module1, module2], jarCompiledModules)
}
```

Ovviamente si può anche utilizzare whenAllBound() per iniziare una catena di operazioni asincrone:

17 Rilasciata in beta-1 il 30 dicembre 2011

```
whenAllBound([module1, module2], jarCompiledModules).then publishToMavenRepo
then {println 'Done'}
```

In alternativa, è possibile rendere asincrona la funzione jarCompileModules() ed invocarla diretta mente sulle promesse.
JarCompileModules(), essendo una funzione asincrona, risolverà le promesse al suo interno senza ricevere alcuna assistenza dal codice chiamante.

```
final jarCompiledModules = {module1, module2 -> ...}.asyncFun()
jarCompiledModules(module1, module2)
```

Concludendo, dato che una funzione asincrona restituisce una promessa, è possibile utilizzare jarCompileModules() per iniziare una catena.

```
jarCompiledModules(module1, module2).then publishToMavenRepo then {println
'Done'}
```

Questo mostra come ci sia molta libertà di scelta sulla composizione dei servizi asincroni, come il completamento di alcuni possa sbloccare il computo di altri e come il programmatore sia completamente schermato dalla gestione del thread pool e dello scheduling.

4) Il parallelismo sui dati

Focalizzarsi sui dati invece che sui processi aiuta notevolmente nella creazione di programmi concorrenti robusti; il programmatore definisce i dati e le funzioni che li devono elaborare e poi lascia che sia l'apparato sottostante a processare il tutto.
Tipicamente saranno creati un insieme di task concorrenti che saranno a loro volta inviati ad un insieme di thread[18] per l'elaborazione vera e propria.

In GPars, le classi GParsPool e GParsExecutorsPool forniscono le tecniche per gestire il parallelismo a basso livello.
Mentre la classe GParsPool si basa sul framework fork/join di jsr-166y e di conseguenza offre molte più funzionalità e migliori prestazioni, la classe GParsExecutorsPool si basa sui classici executors di Java ed è quindi più facilmente gestibile in un ambiente che debba sottostare a delle limitazioni di qualche tipo[19].

Il parallelismo a basso livello di GPars copre fondamentalmente tre problematiche:

1) L'elaborazione concorrente di collezioni
2) L'esecuzione asincrona di funzioni (chiusure)
3) L'esecuzione di algoritmi fork/join (paradigmi divide & conquer)

4.1) Parallel Collections

Manipolare dati spesso significa confrontarsi con collezioni di dati: liste, vettori, insiemi, maps[20], puntatori, stringhe e molti altri tipi di dati che possono essere visti come collezioni di elementi.
L'approccio che classicamente si segue è quello di considerare gli elementi sequenzialmente, uno alla volta, eseguendo un'operazione di qualche tipo su ogni elemento.

Si consideri la funzione min(): questa funzione restituisce il più piccolo valore tra quelli contenuti in una collezione di elementi.
Quando viene invocato il metodo min() su una collezione di numeri, il thread chiamante crea un accumulatore di quello che è “fin'ora il minore” inizializzandolo ad un valore arbitrario, per esempio il massimo possibile; poi quel thread itererà sugli elementi della collezione confrontandoli di volta in volta con il valore memorizzato nell'accumulatore aggiornando quest'ultimo secondo necessità.
Al termine della computazione, il minor valore sarà memorizzato nell'accumulatore.

Questo algoritmo, per quanto semplice, non è in grado di sfruttare un sistema multiprocessore: anche avendo a disposizione solo quattro core fisici si sfrutta solo il 25% della capacità di calcolo, effettivamente sprecando il 75% della capacità potenziale dell'hardware.

Una funzione parallel_min() potrebbe dividere la collezione di partenza in sotto collezioni di valori contigui per esempio creando una struttura ad albero e calcolare parallelamente il min() di ciascuna di questi sottoinsiemi di dati promuovendone il risultato al prossimo ciclo di confronti.
Per ogni ciclo di confronti, ciascuno di questi sottoinsiemi può essere processato da un core diverso

18 Altrimenti noto come thread pool.
19 Un esempio di ambiente limitato è costituito da un ambiente dotato di relativamente poca memoria di sistema, si pensi ad un dispositivo embedded.
20 Può essere inteso come mappa o tabella, per quanto, a seconda dell'implementazione, potrebbe tranquillamente riferirsi ad una hash table (ad esempio); in GPars tipicamente ci si sta riferendo ad una struttura dati chiamata Parallel Array.

in parallelo evitando che i thread che effettivamente si stanno occupando della computazione competano per il controllo delle risorse.

Parallel arrays
La libreria jsr-166y fornisce un'astrazione molto conveniente chiamata Parallel arrays; GPars sfrutta l'applicazione dei Parallel arrays in vari modi.
Le classi GParsPool e GParsExecutorsPool[21] mettono a disposizione varianti parallele dei comuni metodi iterativi di Groovy come each(), collect(), findAll() etc[22].

```
def selfPortraits = images.findAllParallel{it.contains me}.collectParallel
{it.resize()}
```

Permettono anche un approccio più funzionale all'elaborazione di collezioni secondo il paradigma map/reduce.
```
def smallestSelfPortrait = images.parallel.filter{it.contains
me}.map{it.resize()}.min{it.sizeInMB}
```

GParsPool

La classe GParsPool abilita un DSL[23] concorrente basato su Parallel Arrays per collezioni ed oggetti.

Esempi di utilizzo.
```
//Sommatoria concorrente
 GParsPool.withPool {
     final AtomicInteger result = new AtomicInteger(0)
     [1, 2, 3, 4, 5].eachParallel {result.addAndGet(it)}
     assertEquals 15, result
 }
 //Moltiplicazione asincrona
 GParsPool.withPool {
     final List result = [1, 2, 3, 4, 5].collectParallel {it * 2}
     assert ([2, 4, 6, 8, 10].equals(result))
 }
```

La chiusura accetta un'istanza di ForkJoinPool come parametro che può poi essere usato liberamente all'interno della chiusura.
```
//Controlla se tutti gli elementi in una collezione soddisfano una certa
condizione
 GParsPool.withPool(5) {ForkJoinPool pool ->
     assert [1, 2, 3, 4, 5].everyParallel {it > 0}
     assert ![1, 2, 3, 4, 5].everyParallel {it > 1}
 }
```

Il metodo GParsPool.withPool() accetta come parametri opzionali il numero di thread da creare nel thread pool e un handler da lanciare per le eccezioni.
```
withPool(10) {...}
withPool(20, exceptionHandler) {...}
```

Il GParsPool.withExistingPool() accetta come parametro un'istanza di ForkJoinPool da riutilizzare.

21 L'implementazione di GParsExecutorsPool non è basata sulla libreria jsr-166y, quindi non utilizza i parallel arrays.
22 Si faccia riferimento alle API di Groovy relative alla classe collection disponibili in [17] per le versioni sequenziali dei metodi qui descritti.
23 Domain-specific language [18]

Il DSL è valido solo all'interno del blocco di codice a cui è associato e solo per il thread che ha chiamato il metodo withPool() o withExistingPool().

Il metodo withPool() ritorna solo quando tutti i thread al suo interno abbiano completato i compiti a loro assegnati (sotto forma di task) e sono stati distrutti, restituisce il valore di ritorno dell'associato blocco di codice (come farebbe una qualsiasi altra funzione sequenziale).
Il metodo withExistingPool(), non creando un thread pool proprio, restituisce il risultato non appena questo sia disponibile e non attende che i thread utilizzati siano stati distrutti.
Alternativamente, la classe GParsPool può essere importata staticamente
```
import static groovyx.gpars.GParsPool.`*`
```
questo permette di ometterene il nome invocandone i metodi.

```
withPool {
     assert [1, 2, 3, 4, 5].everyParallel {it > 0}
     assert ![1, 2, 3, 4, 5].everyParallel {it > 1}
 }
```

I seguenti metodi sono correntemente applicabili a tutti gli oggetti in Groovy:

- eachParallel()
- eachWithIndexParallel()
- collectParallel()
- findAllParallel()
- findAnyParallel
- findParallel()
- everyParallel()
- anyParallel()
- grepParallel()
- groupByParallel()
- foldParallel()
- minParallel()
- maxParallel()
- sumParallel()
- splitParallel()
- countParallel()
- foldParallel()

Parallelizzazione tramite meta-classe
Esiste la possibilità, data una normale classe sequenziale, di crearne una meta classe o un'istanza arricchendola con i metodi di GparsPool utilizzando la classe ParallelEnhancer.

```
import groovyx.gpars.ParallelEnhancer
def list = [1, 2, 3, 4, 5, 6, 7, 8, 9]
ParallelEnhancer.enhanceInstance(list)
println list.collectParallel {it * 2 }
def animals = ['dog', 'ant', 'cat', 'whale']
ParallelEnhancer.enhanceInstance animals
println (animals.anyParallel {it ==~ /ant/} ? 'Found an ant' : 'No ants found')
println (animals.everyParallel {it.contains('a')} ? 'All animals contain a' :
'Some animals can live without an a')
```

Utilizzando la classe ParallelEnhancer l'uso dei DSL di GParsPool non è costretto all'interno di un

blocco costruito con withPool(); la classe o le istanze rivalutate rimarranno tali fintantoché non saranno eliminate dal garbage collector.

La gestione degli errori

Se, durante l'elaborazione, una qualsiasi istanza creata da un metodo di GParsPool lancia una o più eccezioni, la prima eccezione lanciata viene rilanciata dal metodo xxxParallel() al cui interno è stata creata e l'algoritmo viene terminato il prima possibile.
Questo meccanismo per la gestione degli errori è costruito su quello intrinseco del framework fork/join; dato che un algoritmo in questo framework è per sua natura gerarchico, non si trae alcun beneficio dalla prosecuzione della computazione quando una parte dell'algoritmo fallisce.
L'implementazione di GParsPool non fornisce nessuna garanzia sul suo comportamento dopo che sia stata invocata la prima eccezione non gestita, a parte terminare l'algoritmo e passare al thread chiamante la prima eccezione invocata.

Parallelizzare una collezione in modo trasparente all'utente

Gpars, oltre a fornire i metodi xxxParallel(), permette anche di cambiare la semantica dei metodi originali parallelizzandoli.
Si stia, per esempio, chiamando un metodo di una libreria su una collezione: il metodo in questione sia collect(), ad esempio; utilizzando il metodo makeConcurrent() si parallelizza il modo in cui il metodo collect() viene applicato agli elementi della collezione.
Questo è valido per qualunque metodo sequenziale si voglia applicare agli elementi di una collezione.

```
GParsPool.withPool {
    //Il metodo selectImportantNames() elaborerà la collezione di nomi
parallelamente
    assert ['ALICE', 'JASON'] == selectImportantNames(['Joe', 'Alice', 'Dave',
'Jason'].makeConcurrent())
}
/**
 * Una funzione implementata usando i metodi sequenziali standard collect() e
findAll().
 */
def selectImportantNames(names) {
    names.collect {it.toUpperCase()}.findAll{it.size() > 4}
}
```

Il metodo makeSequential() riporterà la collezione alla semantica sequenziale originaria.

```
import static groovyx.gpars.GParsPool.withPool
def list = [1, 2, 3, 4, 5, 6, 7, 8, 9]
println 'Sequential: '
list.each { print it + ',' }
println()
withPool {
    println 'Sequential: '
    list.each { print it + ',' }
    println()
    list.makeConcurrent()
    println 'Concurrent: '
    list.each { print it + ',' }
    println()
    list.makeSequential()
```

```
    println 'Sequential: '
    list.each { print it + ',' }
    println()
}
println 'Sequential: '
list.each { print it + ',' }
println()
```

Il metodo asConcurrent() permette di specificare blocchi di codice in cui la collezione utilizzerà la semantica concorrente.

```
import static groovyx.gpars.GParsPool.withPool
def list = [1, 2, 3, 4, 5, 6, 7, 8, 9]
println 'Sequential: '
list.each { print it + ',' }
println()
withPool {
    println 'Sequential: '
    list.each { print it + ',' }
    println()
    list.asConcurrent {
        println 'Concurrent: '
        list.each { print it + ',' }
        println()
    }
    println 'Sequential: '
    list.each { print it + ',' }
    println()
}
println 'Sequential: '
list.each { print it + ',' }
println()
```

La parallelizzazione trasparente, ottenuta utilizzando i metodi makeConcurrent(), makeSequential() e asConcurrent() può essere utilizzata anche in concomitanza con ParallelEnhancer.

```
/**
 * Una funzione implementata usando i metodi sequenziali standard collect() e
findAll().
 */
def selectImportantNames(names) {
    names.collect {it.toUpperCase()}.findAll{it.size() > 4}
}
def names = ['Joe', 'Alice', 'Dave', 'Jason']
ParallelEnhancer.enhanceInstance(names)
//Il metodo selectImportantNames() processerà la collezione di nomi
concorrentemente
assert ['ALICE', 'JASON'] == selectImportantNames(names.makeConcurrent())

import groovyx.gpars.ParallelEnhancer
def list = [1, 2, 3, 4, 5, 6, 7, 8, 9]
println 'Sequential: '
list.each { print it + ',' }
println()
ParallelEnhancer.enhanceInstance(list)
println 'Sequential: '
list.each { print it + ',' }
println()
list.asConcurrent {
```

```
    println 'Concurrent: '
    list.each { print it + ',' }
    println()
}
list.makeSequential()
println 'Sequential: '
list.each { print it + ',' }
println()
```

Evitare gli effetti indesiderati
Dato che le chiusure fornite a metodi paralleli come eachParallel() o collectParallel() possono essere eseguite in parallelo, è necessario assicurarsi che le chiusure fornite siano scritte in modo da essere thread safe.
Le chiusure non devono conservare uno stato interno, condividere dati o avere effetti esterni all'elemento su cui sono state invocate.
La violazione di queste regole apre la porta a race conditions e deadlock, i nemici principali della programmazione multi core.

Esempio di cosa non fare:

```
def thumbnails = []
images.eachParallel {thumbnails << it.thumbnail}  //Accesso concorrente ad una
collezione di miniature non thread safe
```

GParsExecutorsPool

La classe GParsExecutorsPool può essere usata come strumento per l'elaborazione parallela di collezioni e oggetti, è basata esclusivamente sul JDK.
La classe GParsExecutorsPool, contrariamente alla classe GParsPool, non si basa sulla libreria jsr-166y ma si basa sullo standard executor service fornito dal JDK per parallelizzare iterativamente il modo in cui una chiusura processa una collezione o un oggetto; bisogna tuttavia sottolineare che, tipicamente, questo viene eseguito molto più efficientemente da GParsPool.
Esempio:

```
//Moltiplicazione asincrona
 GParsExecutorsPool.withPool {
     Collection<Future> result = [1, 2, 3, 4, 5].collectParallel{it * 10}
     assertEquals(new HashSet([10, 20, 30, 40, 50]), new
HashSet((Collection)result*.get()))
 }
//Moltiplicazione asincrona utilizzando una chiusura asincrona
 GParsExecutorsPool.withPool {
     def closure={it * 10}
     def asyncClosure=closure.async()
     Collection<Future> result = [1, 2, 3, 4, 5].collect(asyncClosure)
     assertEquals(new HashSet([10, 20, 30, 40, 50]), new
HashSet((Collection)result*.get()))
 }
```

La chiusura accetta un'istanza di ExecutorService come parametro, istanza che può essere usata liberamente all'interno della chiusura.

```
//Trova un elemento che soddisfa una specifica condizione.
 GParsExecutorsPool.withPool(5) {ExecutorService service ->
     service.submit({performLongCalculation()} as Runnable)
 }
```

Il metodo GparsExecutorsPool.withPool() accetta come parametri opzionali il numero di thread da creare nel pool e la thread factory[24] da utilizzare.

```
withPool(10) {...}
withPool(20, threadFactory) {…}
```

Il metodo GparsExecutorsPool.withExistingPool() utilizza un'istanza già esistente di executor service.
Il DSL è valido solo per il corrispondente blocco di codice e solo per il thread che ha invocato il metodo withPool() o withExistingPool(); come nel caso di GParsPool, il metodo withPool() termina solo quando tutti i thread al suo interno hanno completato i compiti loro assegnati e l'executor service è stato distrutto.
Il metodo withExistingPool() ritorna appena possibile e non attende la terminazione dei thread utilizzati.

Come nel caso di GParsPool, la classe GParsExecutorsPool può essere staticamente importata per utilizzare una sintassi più compatta.

```
import static groovyx.gpars.GParsExecutorsPool.`*`
withPool {
     def result = [1, 2, 3, 4, 5].findParallel{Number number -> number > 2}
     assert result in [3, 4, 5]
 }
```

I metodi seguenti sono disponibili per tutti gli oggetti in Groovy che possono essere considerati thread safe:

- eachParallel()
- eachWithIndexParallel()
- collectParallel()
- findAllParallel()
- findParallel()
- allParallel()
- anyParallel()
- grepParallel()
- groupByParallel()

Parallelizzazione tramite meta-classe
Come alternativa, si può utilizzare la classe GparsExecutorsPoolEnhancer per creare meta-classi di ogni classe o istanza arricchite dai metodi asincroni.

```
import groovyx.gpars.GParsExecutorsPoolEnhancer
def list = [1, 2, 3, 4, 5, 6, 7, 8, 9]
GParsExecutorsPoolEnhancer.enhanceInstance(list)
println list.collectParallel {it * 2 }
def animals = ['dog', 'ant', 'cat', 'whale']
GParsExecutorsPoolEnhancer.enhanceInstance animals
println (animals.anyParallel {it ==~ /ant/} ? 'Found an ant' : 'No ants found')
println (animals.allParallel {it.contains('a')} ? 'All animals contain a' :
'Some animals can live without an a')
```

24 L'uso di una thread factory permette di creare thread con caratteristiche specifiche (sottoclasse, priorità, gruppo etc) senza doverlo specificare ogni volta con una chiamata al costruttore della classe Thread.

Come in GParsPool, utilizzando la classe GparsExecutorsPoolEnhancer, l'uso dei DSL di GParsExecutorsPool non è costretto all'interno di un blocco costruito con withPool(); la classe o le istanze arricchite rimarranno tali fintantoché non saranno eliminate dal garbage collector.

La gestione degli errori
Se viene lanciata un'eccezione durante l'elaborazione di una qualsiasi chiusura, i metodi xxxParallel() lanciano un'istanza di AsyncException inglobante tutte le eccezioni incontrate.

Evitare gli effetti indesiderati
E' necessario evitare di parallelizzare chiusure che influenzino oggetti diversi da quello su cui sono al momento applicate o che mantengano uno stato interno.

Memoize

La funzione gmemoize() permette di associare ad una funzione una cache che ne memorizzerà i risultati; chiamate successive alla funzione con i medesimi parametri non saranno computate, bensì recupereranno il valore calcolato in precedenza e memorizzato in cache senza che questo debba essere specificato dal programmatore.
Usualmente la computazione di una funzione richiede più tempo di quello necessario al recupero di un valore da una cache; la funzione gmemoize permette, dunque di scambiare utilizzo di memoria per prestazioni.
La funzionalità gmemoize di GPars è praticamente identica alla funzione memoize di Groovy, l'unica differenza sta nel fatto che gmemoize esegue ricerche nella cache concorrentemente e quindi può essere in qualche scenario più veloce del suo corrispettivo in Groovy.

Esempio: esplorazione di alcuni siti internet per la ricerca di un contenuto in particolare

```
GParsPool.withPool {
    def urls = ['http://www.dzone.com', 'http://www.theserverside.com',
'http://www.infoq.com']
    Closure download = {url ->
        println "Downloading $url"
        url.toURL().text.toUpperCase()
    }
    Closure cachingDownload = download.gmemoize()
    println 'Groovy sites today: ' + urls.findAllParallel {url ->
cachingDownload(url).contains('GROOVY')}
    println 'Grails sites today: ' + urls.findAllParallel {url ->
cachingDownload(url).contains('GRAILS')}
    println 'Griffon sites today: ' + urls.findAllParallel {url ->
cachingDownload(url).contains('GRIFFON')}
    println 'Gradle sites today: ' + urls.findAllParallel {url ->
cachingDownload(url).contains('GRADLE')}
    println 'Concurrency sites today: ' + urls.findAllParallel {url ->
cachingDownload(url).contains('CONCURRENCY')}
    println 'GPars sites today: ' + urls.findAllParallel {url ->
cachingDownload(url).contains('GPARS')}
}
```

Le chiusure sono contenute in un blocco GParsPool.withPool() a sua volta potenziato con l'aggiunta di una cache grazie alla funzione gmemoize(): si è così creata una nuova chiusura, dotata di cache e contenente la chiusura originaria.
Nell'esempio, la funzione cachingDownload viene chiamata più volte, tuttavia il contenuto corrispondente all'indirizzo fornito viene scaricato solo la prima volta in cui viene richiesto (questo

ovviamente accade per ogni indirizzo univoco), i dati scaricati sono poi disponibili in cache quando avvengono le successive chiamate alla funzione cachingDownload; è importante notare come i valori in cache siano disponibili per tutti i thread all'interno del pool in uso e non solo per quello che abbia effettuato la prima chiamata su un particolare indirizzo dovendo, di conseguenza, gestire il download e la prima computazione.

La funzione gmemoize avvolge il metodo su cui è invocata con una cache contenente i risultati precedenti, tuttavia gmemoize (e la funzione memoize corrispondente in Groovy) può fare di molto di più: a volte aggiungere un po' di memoria ad un algoritmo può significare un drastico cambiamento per quello che riguarda le prestazioni.
Vediamo come con un esempio classico.

La successione di Fibonacci:
Nella successione di Fibonacci, ogni numero è dato dalla somma dei due numeri precedenti.
La complessità computazionale di un'implementazione puramente ricorsiva che segua strettamente la definizione della successione di Fibonacci è esponenziale.

```
Closure fib = {n -> n > 1 ? call(n - 1) + call(n - 2) : n}
```

Con una piccola modifica e l'aggiunta di una cache la complessità computazionale dell'algoritmo diventa magicamente lineare.

```
Closure fib
fib = {n -> n > 1 ? fib(n - 1) + fib(n - 2) : n}.gmemoize()
```

La memoria extra elimina tutte le chiamate ricorsive meno una; tutte le successive chiamate alla stessa funzione fib trarranno beneficio dall'avere per lo meno una parte dei valori già disponibili in cache.

Varianti

Esistono svariate varianti della funzione gmemoize: permettono al programmatore di avere un controllo parziale sull'estensione della cache.

gmemoize
La variante di base, mantiene in cache indefinitamente i valori calcolati dal metodo a cui è associata.
Si tratta della variante che fornisce le migliori prestazioni.

gmemoizeAtMost
Permette all'utente (inteso come codice chiamante) di impostare un limite al numero di oggetti presenti in cache; quando il limite viene superato ogni nuovo oggetto in cache rimpiazza l'oggetto utilizzato meno di recente.
Considerando l'esempio della successione di Fibonacci, è possibile ridurre il numero di numeri in cache a due soli elementi mantenendo la linearità dell'algoritmo.

```
Closure fib
fib = {n -> n > 1 ? fib(n - 1) + fib(n - 2) : n}.gmemoizeAtMost(2)
```

Impostare una dimensione massima per la cache può avere due scopi:

1) Limitare l'impatto della cache sulla memoria di sistema

2) Preservare le prestazioni della funzione: una cache troppo grande può portare ad un degrado prestazionale a causa del tempo necessario alla ricerca un elemento dalla cache, in casi estremi può essere più dispendioso in tempo di computo la ricerca di un elemento in cache che il computo di quello stesso elemento partendo da zero.

gmemoizeAtLeast
Permette alla cache di crescere indefinitamente fin tanto che il garbage collector della JVM non decida di eliminare dalla memoria di sistema le SoftReferences, utilizzate per implementare questo tipo di cache.
Il parametro passato al metodo gmemoizeAtLeast() specifica il minimo numero di elementi della cache che devono essere protetti dalla pulizia eseguita dal garbage collector.
Il numero di oggetti in cache non calerà mai sotto al valore specificato, assicurando che gli oggetti utilizzati più di recente siano preservati.

gmemoizeBetween
Si tratta di un ibrido tra gmemoizeAtLeast e gmemoizeAtMost, permette alla cache di assumere una dimensione variabile all'interno delle due estensioni specificate, a seconda di quanta memoria di sistema sia disponibile e dal livello di attività del garbage collector; tuttavia la dimensione della cache non supera mai il valore massimo impostato per evitarne il degrado prestazionale.

4.2) Map-Reduce

I metodi xxxParallel() devono essere compatibili con i loro corrispettivi sequenziali, quindi, pur processando internamente gli elementi delle collezioni a loro fornite in modo concorrente, devono restituire in uscita una versione sequenziale della specifica collezione ricevuta in ingresso.
Internamente i metodi xxxParallel() costruiscono un'efficiente struttura parallela chiamata parallel array, eseguono la computazione richiesta parallelamente e, una volta terminata la computazione, ricostruiscono la collezione sequenziale da restituire in uscita e distruggono la struttura parallela che hanno internamente utilizzato.
Si può immaginare come, chiamate successive a diversi metodi xxxParallel() effettuate passando sempre la stessa collezione risultino estremamente inefficienti a causa del ripetersi del processo di costruzione e distruzione della struttura parallel array.

Il DSL Map/Reduce risolve questo problema: si comporta in maniera molto simile ai metodi xxxParallel() ma permette di concatenare diverse operazioni costruendo il parallel array all'inizio della catena, distruggendolo solo al termine della medesima; all'interno della catena, I vari metodi del DSL utilizzano come parametro direttamente la struttura parallel array costruita internamente.

```
println 'Number of occurrences of the word GROOVY today: ' + urls.parallel
            .map {it.toURL().text.toUpperCase()}
            .filter {it.contains('GROOVY')}
            .map{it.split()}
            .map{it.findAll{word -> word.contains 'GROOVY'}.size()}
            .sum()
```

In casi come quello riportato nell'esempio, le performance del DSL Map/Reduce sono decisamente migliori di quelle ottenute concatenando i metodi xxxParallel() corrispondenti, tuttavia è possibile utilizzare il DSL Map/Reduce solo all'interno del framework Fork/Join basato su GParsPool e non su quello basato su GParsExecutorsPool.

I metodi che è possibile concatenare sono:

- map()
- reduce()
- filter()
- size()
- sum()
- min()
- max()
- sort()
- groupBy()
- combine()

Al termine della catena è necessario ricostruire una collezione sequenziale, questo si ottiene recuperando la proprietà “collection”.

```
def myNumbers = (1..1000).parallel.filter{it % 2 == 0}.map{Math.sqrt
it}.collection
```

Evitare gli effetti indesiderati
Anche nel DSL Map/Reduce è necessario utilizzare chiusure che non mantengano uno stato interno e che non vadano ad influenzare oggetti esterni a quello su cui stanno lavorando.

Un esempio classico: conta il numero di parole in una stringa.

```
import static groovyx.gpars.GParsPool.withPool
def words = "This is just a plain text to count words in"
print count(words)
def count(arg) {
  withPool {
    return arg.parallel
      .map{[it, 1]}
      .groupBy{it[0]}.getParallel()
      .map {it.value=it.value.size();it}
      .sort{-it.value}.collection
  }
}
```

Lo stesso esempio, implementato utilizzando l'operatore più generale combine:

```
def words = "This is just a plain text to count words in"
print count(words)
def count(arg) {
  withPool {
    return arg.parallel
      .map{[it, 1]}
      .combine(0) {sum, value -> sum + value}.getParallel()
      .sort{-it.value}.collection
  }
}
```

Combine
L'operatore combine si aspetta di ricevere in ingresso una lista di coppie chiave-valore (come, per esempio [chiave 1, valore 1, chiave 2, valore 2, chiave 1, valore 3, chiave 3, valore 4, chiave 1, valore 5, etc]) in cui le chiavi possano essere presenti più volte.
Quando combine riceve in ingresso una lista di questo tipo unisce i valori che condividono la stessa

chiave utilizzando la funzione di accumulazione che gli è stata fornita e produce una mappa costituita da coppie chiave (univoca) – valori accumulati.
Per esempio, la lista [a,b,c,d,a,e,c,f] sarà trasformata nella lista [a: b+e, c: d+f], l'operatore + rappresenta la chiusura accumulatrice che deve essere fornita dall'utente e che si occuperà di fondere assieme i valori corrispondenti alla stessa chiave.
Deve essere fornito anche un valore che inizializzi la chiusura accumulatrice e, dato che il metodo combine elaborerà gli oggetti in parallelo, questo valore iniziale sarà riutilizzato diverse volte, quindi deve trattarsi di qualcosa che permetta ciò; può trattarsi, ad esempio, di una variabile cloneable o immutable o di una chiusura che crei un nuovo accumulatore ogni volta che le sia richiesto.

Ecco alcuni esempi di funzioni accumulatrici e di valori iniziali:

```
accumulator = {List acc, value -> acc << value} initialValue = []
accumulator = {List acc, value -> acc << value} initialValue = {-> []}
accumulator = {int sum, int value -> acc + value} initialValue = 0
accumulator = {int sum, int value -> sum + value} initialValue = {-> 0}
accumulator = {ShoppingCart cart, Item value -> cart.addItem(value)}
initialValue = {-> new ShoppingCart()}
```

Il tipo di dato restituito è map, quindi, partendo da una lista di questo tipo ['he', 1, 'she', 2, 'he', 2, 'me', 1, 'she, 5, 'he', 1] se il valore iniziale fornito è zero, il risultato di combine sarà 'he': 4,'she': 7, 'me': 1.

4.3) Parallel Arrays

In alternativa, è possibile utilizzare la struttura parallel array direttamente: invocando la proprietà parallelArray su una qualsiasi collezione di oggetti si ottiene un'istanza di jsr66y.forkjoin.ParallelArray contente gli elementi della collezione originaria; tale istanza può poi essere elaborata utilizzando i metodi propri della libreria JSR-166Y.

Ad esempio:

```
groovyx.gpars.GParsPool.withPool {
    assert 15 == [1, 2, 3, 4, 5].parallelArray.reduce({a, b -> a + b} as
Reducer, 0)                                          //sommatoria
    assert 55 == [1, 2, 3, 4, 5].parallelArray.withMapping({it ** 2} as
Mapper).reduce({a, b -> a + b} as Reducer, 0)//sommatoria dei numeri al quadrato
    assert 20 == [1, 2, 3, 4, 5].parallelArray.withFilter({it % 2 == 0} as
Predicate)                              //sommatoria dei quadrati dei numeri pari
            .withMapping({it ** 2} as Mapper)
            .reduce({a, b -> a + b} as Reducer, 0)
    assert 'aa:bb:cc:dd:ee' == 'abcde'.parallelArray
//concatenazione di caratteri duplicati divisi da un separatore
            .withMapping({it * 2} as Mapper)
            .reduce({a, b -> "$a:$b"} as Reducer, "")
```

4.4) Computazione asincrona

A volte si ha la necessità di eseguire certe operazioni in background; tipicamente si tratta di operazioni il cui risultato non è immediatamente necessario, ad esempio download di dati, ricerche o altri tipi di elaborazioni.

GParsPool e GParsExecutorsPool, pur basandosi su concetti diversi, forniscono dei servizi praticamente identici per la gestione delle attività in background e per recuperarne i risultati quando saranno richiesti.

Le chiusure all'interno di un blocco Gpars(Executors)Pool.withPool() vengono dotate dei due metodi seguenti:

I. async() - Crea una variante asincrona della chiusura su cui è chiamato. Quando invocata, la chiusura restituirà un "future" che permette il recupero dell'eventuale risultato.
II. callAsync() - Lancia la chiusura su un thread diverso fornendo i necessari argomenti, restituisce in uscita un future per la gestione dell'eventuale risultato.

Esempi:

```
GParsPool.withPool() {
    Closure longLastingCalculation = {calculate()}
    Closure fastCalculation = longLastingCalculation.async()  //si crea una
nuova chiusura che avvia la chiusura originaria in un thread pool
    Future result=fastCalculation()                           //ritorna quasi
istantaneamente
    //fa qualcos'altro durante l'esecuzione di longLastingCalculation
    println result.get()      //recupera il risultato della computazione
asincrona
}

GParsPool.withPool() {
    /**
     * Il metodo callAsync()è una variante asincrona del metodo call() usato per
chiamare una chiusura
     * Restituisce un future che conterrà il risultato della computazione.
     */
    assert 6 == {it * 2}.call(3)
    assert 6 == {it * 2}.callAsync(3).get()
}
```

Timeouts

Il metodo callTimeoutAsync() accetta come parametro un long o un'istanza di Duration e permette all'utente di terminare la computazione asincrona automaticamente al termine del determinato intervallo temporale.

```
{->
    while(true) {
        Thread.sleep 1000  //Simula una computazione utile
        if (Thread.currentThread().isInterrupted()) break;  //Il thread è stato
terminato
    }
}.callTimeoutAsync(2000)
```

Per permettere l'attuale terminazione del thread dall'esterno, il codice asincrono in esecuzione deve monitorare l'attributo interrupted del thread che lo sta eseguendo e cessare la sua esecuzione quando interrupted assuma il valore vero.

Accessori forniti all'executor service

L'operatore << (leftshift) è fornito come accessorio alle classi ExecutorService e alla jsr166y.forkjoin.ForkJoinPool: permette di inviare un task al thread pool ricevendo un Future come risultato.

Esempio:
```
GParsExecutorsPool.withPool {ExecutorService executorService ->
    executorService << {println 'Inside parallel task'}
}
```

Eseguire chiusure in parallelo
Le classi GParsPool e GPArsExecutorsPool forniscono anche i metodi executeAsync() ed executeAsyncAndWait(); questi due metodi permettono di lanciare facilmente diverse chiusure da eseguire in parallelo.

Esempio
```
GParsPool.withPool {
    assertEquals([10, 20], GParsPool.executeAsyncAndWait({calculateA()},
{calculateB()}))         //attende i risultati
    assertEquals([10, 20], GParsPool.executeAsync({calculateA()},
{calculateB()})*.get())  //restituisce un Future e non aspetta che i risultati
siano disponibili
}
```

Composizione di funzioni asincrone

Una funzione altro non è che un blocco di codice dal comportamento ben definito: dato lo stesso input, se la funzione non conserva uno stato interno genererà sempre lo stesso risultato, senza modificare inaspettatamente il proprio comportamento, indipendentemente dal numero di thread che stanno eseguendo contemporaneamente quella particolare funzione.
In Groovy, una chiusura che non acceda ad alcunché situato all'esterno del proprio scopo altro non è che una pura funzione e, in quanto tale, può essere composta.
Per esempio, componendo una funzione che sommi due numeri con una funzione che iteri su tutti gli elementi di una collezione, si può ottenere una funzione composta che esegue la sommatoria di tutti gli elementi della collezione.

```
def sum = (0..100000).inject(0, {a, b -> a + b})
```

Sostituendo la funzione somma con una funzione di confronto, si può rapidamente ottenere una funzione composta che restituisca il massimo elemento presente nella collezione.

```
def max = myNumbers.inject(0, {a, b -> a>b?a:b})
```

Ovviamente questi sono esempi puramente sequenziali.
Quest'altro esempio è composto da quattro funzioni che si occupano di verificare se il contenuto di una particolare pagina web coincide con quello di un file presente in locale.
Si tratta di scaricare la pagina, caricare il file, calcolarne l'hash e confrontare i valori così ottenuti.

```
Closure download = {String url ->
    url.toURL().text
}
Closure loadFile = {String fileName ->
    …  //carica il file
}
Closure hash = {s -> s.hashCode()}.asyncFun()
Closure compare = {int first, int second ->
    first == second
}
def result = compare(hash(download('http://www.gpars.org')),
```

```
hash(loadFile('/coolStuff/gpars/website/index.html')))
println "The result of comparison: " + result
```

Ogni funzione svolge un ruolo particolare, comporle è semplice come concatenarne le chiamate.
Il problema è che non si sta sfruttando l'indipendenza delle due funzioni download() e loadFile() e che i corrispettivi hash non si stanno calcolando concorrentemente perché si sta utilizzando una serie di chiamate sequenziali.
Ovviamente non tutte le funzioni posso essere eseguite in parallelo, alcune funzioni, come compare() e hash() necessitano dei risultati di altre funzioni per poter essere eseguite, è necessario bloccarle fintantoché questi siano disponibili.

I vincoli temporali tra funzioni sono espressi implicitamente nel codice; per esempio se una funzione utilizza come uno dei parametri il risultato di un'altra funzione un vincolo implicito è specificato.
Per sfruttare l'indipendenza esistente tra i vari metodi, è necessario trasformarli in modo che generino promesse[25], quando invocati, ed accettino le promesse generate da altri metodi.

```
withPool {
    def maxPromise = numbers.inject(0, {a, b -> a>b?a:b}.asyncFun())
    println "Guardate! Posso parlare mentre qualcuno sta lavorando per me."
    println maxPromise.get()
}
```

La funzione inject() non si cura di che tipo di oggetto sia restituito dalla funzione di confronto.

Esempio di come si può attendere l'effettiva disponibilità del valore promesso.

```
withPool {
    def sumPromise = (0..100000).inject(0, {a, b -> a + b}.asyncFun())
    println "Are we done yet? " + sumPromise.bound
    sumPromise.whenBound {sum -> println sum}
}
```

Il metodo get() è disponibile in una variante che permette di specificare un timeout onde evitare di aspettare in eterno il risultato di una computazione che sia incappata in un errore.

Nel caso in cui venga generata un'eccezione dal metodo che si dovrebbe occupare di fornire il valore promesso, il metodo get() collegato lancerà a sua volta un'eccezione una volta invocato.

```
try {
    sumPromise.get()
} catch (MyCalculationException e) {
    println "Ops, qualcosa è andato storto"
}
```

Non ci sono limitazioni sul tipo di funzioni che possono essere composte, se è possibile concatenare chiamate a delle funzioni sequenziali allora è possibile comporre anche le loro versioni asincrone.
Tornando all'esempio originario (quello che scarica il contenuto di una pagina web e lo confronta con il contenuto di un file locale) è semplicemente possibile chiamare tutte le funzioni come asincrone utilizzando il metodo asyncFun().

```
Closure download = {String url ->
```

25 Si ricorda che in GPars una promessa altro non è che una variabile dataflow, quindi è possibile verificarne lo stato, collegarla a più handlers o utilizzarla come input di un algoritmo dataflow.

```
        url.toURL().text
    }.asyncFun()
    Closure loadFile = {String fileName ->
        …  //carica il file
    }.asyncFun()
    Closure hash = {s -> s.hashCode()}.asyncFun()
    Closure compare = {int first, int second ->
        first == second
    }.asyncFun()
    def result = compare(hash(download('http://www.gpars.org')),
hash(loadFile('/coolStuff/gpars/website/index.html')))
    println 'Allowed to do something else now'
    println "The result of comparison: " + result.get()
```

Composizione di promesse

Un'altra caratteristica decisamente utile delle funzioni asincrone è che anche le loro promesse possono essere composte.

```
import static groovyx.gpars.GParsPool.withPool
  withPool {
      Closure plus = {Integer a, Integer b ->
          sleep 3000
          println 'Adding numbers'
          a + b
      }.asyncFun()
      Closure multiply = {Integer a, Integer b ->
          sleep 2000
          a * b
      }.asyncFun()
      Closure measureTime = {->
          sleep 3000
          4
      }.asyncFun()
      Closure distance = {Integer initialDistance, Integer velocity, Integer
time ->
          plus(initialDistance, multiply(velocity, time))
      }.asyncFun()
      Closure chattyDistance = {Integer initialDistance, Integer velocity,
Integer time ->
          println 'All parameters are now ready - starting'
          println 'About to call another asynchronous function'
          def innerResultPromise = plus(initialDistance, multiply(velocity,
time))
          println 'Returning the promise for the inner calculation as my own
result'
          return innerResultPromise
      }.asyncFun()
      println "Distance = " + distance(100, 20, measureTime()).get() + ' m'
      println "ChattyDistance = " + chattyDistance(100, 20, measureTime()).get()
+ ' m'
  }
```

Se una funzione asincrona (per esempio la funzione distance) chiama al suo interno una funzione asincrona (plus, ad esempio) e ritorna la promessa del risultato al thread chiamante originario, la promessa fatta dalla funzione interna verrà composta con la promessa fatta dalla funzione esterna che l'ha invocata e il risultato della computazione interna verrà associato al risultato della computazione esterna (distance) una volta che la computazione interna (plus) abbia seguito il suo corso.

L'abilità di comporre promesse fornisce alle funzioni asincrone la possibilità non solo di sospendersi senza bloccare il thread che le sta eseguendo mentre attendono un parametro fornito da un'altra funzione ma anche quando chiamano un'altra funzione asincrona al loro interno.

Come trasformare i metodi di una classe in funzioni asincrone
Si può trasformare un metodo di una classe in una chiusura utilizzando l'operatore .& rendendolo poi asincrono utilizzando asyncFun().

```
class DownloadHelper {
    String download(String url) {
        url.toURL().text
    }
    int scanFor(String word, String text) {
        text.findAll(word).size()
    }
    String lower(s) {
        s.toLowerCase()
    }
}
//Trasformazione dei metodi nelle loro versioni asincrone
withPool {
    final DownloadHelper d = new DownloadHelper()
    Closure download = d.&download.asyncFun()
    Closure scanFor = d.&scanFor.asyncFun()
    Closure lower = d.&lower.asyncFun()
    //computazione asincrona
    def result = scanFor('groovy', lower(download('http://www.infoq.com')))
    println 'Allowed to do something else now'
    println result.get()
}
```

Uso delle annotazioni
Esiste un'alternativa all'uso della funzione asyncFun() per la creazione di una chiusura asincrona: si tratta dell'annotazione @AsyncFun che permette di annotare un blocco chiusura trasformandolo in una chiusura asincrona direttamente all'interno del codice.
I campi della chiusura devono essere inizializzati immediatamente e la classe che la contiene deve essere istanziata in un blocco withPool.

```
import static groovyx.gpars.GParsPool.withPool
import groovyx.gpars.AsyncFun
class DownloadingSearch {
    @AsyncFun Closure download = {String url ->
        url.toURL().text
    }
    @AsyncFun Closure scanFor = {String word, String text ->
        text.findAll(word).size()
    }
    @AsyncFun Closure lower = {s -> s.toLowerCase()}
    void scan() {
        def result = scanFor('groovy', lower(download('http://www.infoq.com')))
//Computazione sincrona
        println 'Allowed to do something else now'
        println result.get()
    }
}
withPool {
    new DownloadingSearch().scan()
}
```

L'annotazione AsyncFun usa di default un'istanza di GParsPool del blocco withPool che la contiene, è tuttavia possibile specificare il tipo di thread pool che si vuole utilizzare.

```
@AsyncFun(GParsExecutorsPoolUtil) def sum6 = {a, b -> a + b }
```

L'annotazione AsyncFun permette all'utente di specificare se la funzione risultante debba bloccare il thread che la sta eseguendo, di default questo non avviene.

.

```
@AsyncFun(blocking = true)
def sum = {a, b -> a + b }
```

4.5) Speculazione parallela

Data l'abbondanza di core disponibili, la risoluzione di alcuni problemi può trarre beneficio dalla duplicazione parallela a forza bruta, cioè dalla possibilità di non decidere in fase di codifica come risolvere il problema, quale algoritmo sia meglio applicare etc ma di eseguire in parallelo tutte le soluzioni potenziali.

Si immagini di dover eseguire un compito che richieda molto tempo, come il computo di una funzione molto onerosa o la lettura di dati da dei file, da un database o da internet; fortunatamente si conoscono vari approcci (equivalenti dal punto di vista del risultato della computazione, ad esempio si conoscono diverse url da cui scaricare i dati) che permettono di eseguire il compito sottoposto, il problema è che non tutti sono equivalenti dal punto di vista dell'impiego di tempo e, sfortunatamente, non garantiscono l'esecuzione priva di errori (ad esempio problemi di rete che impediscono il recupero dei dati da un particolare sito).

Domande tipiche che si possono incontrare sono:

- Nell'ordinamento di una particolare lista, è più efficiente applicare quick sort o merge sort ?
- Da quale url è possibile recuperare più rapidamente i dati (congestione di rete) ?
- Quale url è raggiungibile e quale non lo è ?
- Questo servizio è disponibile nella sua locazione primaria o devo usare un backup ?
- Quale istanza di un database remoto devo accedere per ottenere dei dati consistenti ?[26]

La speculazione parallela in GPars rende possibile l'esplorazione parallela di tutte le alternative possibili (o a cui si è interessati) e l'ottenimento del risultato dalla prima alternativa funzionante che termina ignorando tutte le alternative che hanno terminato in modo improprio o che sono ancora in esecuzione.
Tutto questo si ottiene utilizzando il metodo speculate() presente in GParsExecutorsPool e in GParsPool.

```
def numbers = …
def quickSort = …
def mergeSort = …
def sortedNumbers = speculate(quickSort, mergeSort)
```

L' esempio mostra come eseguire merge sort e quick sort parallelamente ottenendo il risultato voluto dal primo dei due algoritmi che termina, questo senza conoscere in anticipo quale dei due terminerà per primo.
Nel caso in cui le risorse computazionali necessarie all'esecuzione dei due algoritmi (in questo caso

26 Si visiti [19] per un esempio di questa problematica.

possono bastare due core) non siano impegnate da altri task, la speculazione non sovraccarica il sistema e permette di ottenere il risultato come se si stesse eseguendo solo l'algoritmo più veloce; ottenendo in ogni caso il risultato prima di quello fornito dall'algoritmo più lento, senza tuttavia conoscere in anticipo quali siano le prestazioni dei due algoritmi sui nostri dati, da cui la speculazione.

Esempio: download di un documento da diverse sorgenti con diverse caratteristiche di affidabilità e velocità.
Ovviamente il thread pool utilizzato deve contenere un numero di thread sufficiente al computo contemporaneo di tutte le chiusure presenti.

```
import static groovyx.gpars.GParsPool.speculate
import static groovyx.gpars.GParsPool.withPool
def alternative1 = {
    'http://www.dzone.com/links/index.html'.toURL().text
}
def alternative2 = {
    'http://www.dzone.com/'.toURL().text
}
def alternative3 = {
    'http://www.dzzzzzone.com/'.toURL().text  //Url errato
}
def alternative4 = {
    'http://dzone.com/'.toURL().text
}
withPool(4) {
    println speculate([alternative1, alternative2, alternative3,
alternative4]).contains('groovy')
}
```

In alternativa, è possibile utilizzare i dataflow sotto forma di variabili o di streams; in particolare quando non sia necessario terminare i rami della speculazione che hanno incontrato problemi.

```
import groovyx.gpars.dataflow.DataflowQueue
import static groovyx.gpars.dataflow.Dataflow.task
def alternative1 = {
    'http://www.dzone.com/links/index.html'.toURL().text
}
def alternative2 = {
    'http://www.dzone.com/'.toURL().text
}
def alternative3 = {
    'http://www.dzzzzzone.com/'.toURL().text  //Fallirà a causa di un errore
nell'url
}
def alternative4 = {
    'http://dzone.com/'.toURL().text
}
//Scegliere una delle due alternative seguenti, funzioneranno entrambe
final def result = new DataflowQueue()
//  final def result = new DataflowVariable()
[alternative1, alternative2, alternative3, alternative4].each {code ->
    task {
        try {
            result << code()
        } catch (ignore) { }  //Si ignorano deliberatamente gli url errati
    }
}
println result.val.contains('groovy')
```

4.6) Fork-Join

Il paradigma fork/join costituisce uno strumento molto potente per la risoluzione di problemi gerarchici, degli esempi applicativi sono gli algoritmi quick e merge sort, la navigazione in strutture ad albero etc.
Fondamentalmente si tratta di un paradigma applicabile a tutti i problemi che possono essere risolti con un approccio divide and conquer:

- Si divide il problema originario in sotto problemi, applicando ricorsivamente l'algoritmo a ciascun sotto problema.
- Quando il sotto problema in esame è diventato sufficientemente semplice, viene risolto direttamente.
- Le soluzioni di tutti i sotto problemi sono composte per risolvere il sotto problema da cui derivano, questo viene iterato finché si risolve il problema originario.

La libreria JSR-166y fornisce tutti gli strumenti necessari alla gestione di qualunque tipo di algoritmo basato su fork/join ma, essendo una libreria di Java, presenta tutta una serie di problematiche relative alla sincronizzazione e alla gestione di thread e di thread pool.
GPars permette di sfruttare tutte le potenzialità della libreria JSR-166y nascondendone la complessità.
Essendo basato sulla libreria JSR-166y, il paradigma fork/join è disponibile solo utilizzando la classe GParsPool e non è disponibile nella classe GParsExecutorsPool.

Esempio: calcolare il numero di file presente in una directory e in tutte le sue sotto directory

```
import static groovyx.gpars.GParsPool.runForkJoin
import static groovyx.gpars.GParsPool.withPool
withPool() {
    println """Number of files: ${
        runForkJoin(new File("./src")) {file ->
            long count = 0
            file.eachFile {
                if (it.isDirectory()) {
                    println "Forking a child task for $it"
                    forkOffChild(it)           //crea un task figlio
                } else {
                    count++
                }
            }
            return count + (childrenResults.sum(0))
            //Usa i risultati del task figli creati per calcolare il proprio
risultato
        }
    }"""
}
```

Il metodo runForkJoin() utilizzerà il codice e i valori iniziali forniti per costruite la computazione gerarchica; il numero di parametri forniti al metodo runForkJoin() deve corrispondere al numero di parametri attesi dalla chiusura su cui è applicato e al numero di parametri passati ai metodi forkOffChild() e runChildDirectly().

```
def quicksort(numbers) {
    withPool {
        runForkJoin(0, numbers) {index, list ->
            def groups = list.groupBy {it <=> list[list.size().intdiv(2)]}
            if ((list.size() < 2) || (groups.size() == 1)) {
```

```
                return [index: index, list: list.clone()]
            }
            (-1..1).each {forkOffChild(it, groups[it] ?: [])}
            return [index: index, list: childrenResults.sort {it.index}.sum
{it.list}]
        }.list
    }
}
```

Alternativamente è possibile creare dei task worker specifici, in modo da limitare l'overhead generato dalla moltiplicazione incontrollata dei parametri generata dall'uso dei worker generici; è oltretutto possibile implementare i task worker in Java (nell'esempio, si tratta della classe AbstractForkJoinWorker), ottenendo un certo miglioramento prestazionale dell'algoritmo.

```
public final class FileCounter extends AbstractForkJoinWorker<Long> {
    private final File file;
    def FileCounter(final File file) {
        this.file = file
    }
    @Override
    protected Long computeTask() {
        long count = 0;
        file.eachFile {
            if (it.isDirectory()) {
                println "Forking a thread for $it"
                forkOffChild(new FileCounter(it))    //Crea un nuovo task figlio
            } else {
                count++
            }
        }
        return count + ((childrenResults)?.sum() ?: 0)  //utilizza i risultati
dei figli per calcolare il proprio
    }
}
withPool(1) {pool ->  //numero di thread nel pool
    println "Number of files: ${runForkJoin(new FileCounter(new File("..")))}"
}
```

Il computo delle operazioni di Fork/Join può essere eseguito con un insieme di thread molto ridotto grazie all'utilizzo della classe TaskBarrier.
La classe TaskBarrier viene utilizzata per permettere ai vari task creati dal paradigma Fork/Join di restituire al pool il thread che li sta eseguendo, allorché debbano sospendersi per attendere il risultato dei task figli che hanno a loro volta creato; il thread così liberato può, ad esempio, occuparsi dell'esecuzione di uno dei task figli rendendo così possibile l'effettiva esecuzione del paradigma Fork/Join utilizzando un solo thread (per quanto questo sia decisamente poco sensato).

Esempio: Mergesort.

```
import static groovyx.gpars.GParsPool.runForkJoin
import static groovyx.gpars.GParsPool.withPool
/**
 * Smezza una lista di numeri
 */
def split(List<Integer> list) {
    int listSize = list.size()
    int middleIndex = listSize / 2
    def list1 = list[0..<middleIndex]
    def list2 = list[middleIndex..listSize - 1]
```

```
    return [list1, list2]
}
/**
 * Fonde due liste ordinate
 */
List<Integer> merge(List<Integer> a, List<Integer> b) {
    int i = 0, j = 0
    final int newSize = a.size() + b.size()
    List<Integer> result = new ArrayList<Integer>(newSize)
    while ((i < a.size()) && (j < b.size())) {
        if (a[i] <= b[j]) result << a[i++]
        else result << b[j++]
    }
    if (i < a.size()) result.addAll(a[i..-1])
    else result.addAll(b[j..-1])
    return result
}
final def numbers = [1, 5, 2, 4, 3, 8, 6, 7, 3, 4, 5, 2, 2, 9, 8, 7, 6, 7, 8, 1,
4, 1, 7, 5, 8, 2, 3, 9, 5, 7, 4, 3]
withPool(3) {  //feel free to experiment with the number of fork/join threads in
the pool
    println """Sorted numbers: ${
        runForkJoin(numbers) {nums ->
            println "Thread ${Thread.currentThread().name[-1]}: Sorting $nums"
            switch (nums.size()) {
                case 0..1:
                    return nums               //restituisce il proprio risultato
                case 2:
                    //Restituisce una lista ordinata composta da due elementi
                    if (nums[0] <= nums[1]) return nums
                    else return nums[-1..0]
                default:
                    def splitList = split(nums)
                    [splitList[0], splitList[1]].each {forkOffChild it}
                    //Crea due task figli
                    return merge(* childrenResults)
                    //Usa i risultati dei figli per calcolare il proprio
            }
        }
    }"""
}
```

Stesso esempio, in questo si utilizza una classe worker dedicata.

```
public final class SortWorker extends AbstractForkJoinWorker<List<Integer>> {
    private final List numbers
    def SortWorker(final List<Integer> numbers) {
        this.numbers = numbers.asImmutable()
    }
    /**
     * Smezza una lista di numeri
     */
    def split(List<Integer> list) {
        int listSize = list.size()
        int middleIndex = listSize / 2
        def list1 = list[0..<middleIndex]
        def list2 = list[middleIndex..listSize - 1]
        return [list1, list2]
    }
    /**
     * Fonde due liste ordinate
```

```
     */
    List<Integer> merge(List<Integer> a, List<Integer> b) {
        int i = 0, j = 0
        final int newSize = a.size() + b.size()
        List<Integer> result = new ArrayList<Integer>(newSize)
        while ((i < a.size()) && (j < b.size())) {
            if (a[i] <= b[j]) result << a[i++]
            else result << b[j++]
        }
        if (i < a.size()) result.addAll(a[i..-1])
        else result.addAll(b[j..-1])
        return result
    }
    /**
     * Ordina una lista di dimensioni ridotte o delega l'ordinamento a due
figli, se la lista contiene più di due elementi.
     */
    @Override
    protected List<Integer> computeTask() {
        println "Thread ${Thread.currentThread().name[-1]}: Sorting $numbers"
        switch (numbers.size()) {
            case 0..1:
                return numbers                //restituisce il proprio risultato
            case 2:
                //Restituisce una lista ordinata composta da due elementi
                if (numbers[0] <= numbers[1]) return numbers
                else return numbers[-1..0]
            default:
                def splitList = split(numbers)
                [new SortWorker(splitList[0]), new
SortWorker(splitList[1])].each{forkOffChild it}  //Crea due task figli
                return merge(* childrenResults)
                //Usa i risultati dei figli per calcolare il proprio
        }
    }
}
final def numbers = [1, 5, 2, 4, 3, 8, 6, 7, 3, 4, 5, 2, 2, 9, 8, 7, 6, 7, 8, 1,
4, 1, 7, 5, 8, 2, 3, 9, 5, 7, 4, 3]
withPool(1) {  //numero di thread nel pool
    println "Sorted numbers: ${runForkJoin(new SortWorker(numbers))}"
}
```

Il metodo forkOffChild() ha un fratello: il metodo runChildDirectly() che si occupa di riutilizzare direttamente il thread del task padre invece di inviare il task figlio allo scheduler in attesa di un thread che lo possa eseguire.
Tipicamente si invoca forkOffChild() per tutti i sub task tranne l'ultimo, che si esegue direttamente per ridurre l'overhead tipico di questo tipo di computazioni.

Esempio:

```
Closure fib = {number ->
            if (number <= 2) {
                return 1
            }
            //  Questo task verrà probabilmente eseguito da un altro thread
            forkOffChild(number - 1)
            //  Questo task viene eseguito direttamente dal thread corrente
            final def result = runChildDirectly(number - 2)
            return (Integer) getChildrenResults().sum() + result
```

```
}
withPool {
    assert 55 == runForkJoin(10, fib)
}
```

Questo conclude la parte relativa al parallelismo sulle collezioni di dati.

5) CSP

Communicating Sequential Processes, si tratta di un linguaggio formale per la descrizione di modelli di iterazione in sistemi concorrenti.
Per quanto, trattandosi di un formalismo matematico, sia stato utilizzato per la specificazione e la verifica degli aspetti legati alla concorrenza in una varietà di sistemi come ad esempio il T9000 Transputer, sistemi di ecommerce sicuri etc, è anche stato implementato in una serie di linguaggi di programmazione.
CSP permette la descrizione di sistemi in termini dei processi componenti che operano indipendentemente e che comunicano gli uni con gli altri esclusivamente attraverso lo scambio di messaggi.
La parte "sequenziale" nel nome CSP è tuttavia diventata fuorviante; nella sua incarnazione moderna[27] il CSP permette la descrizione di processi sia come processi sequenziali sia come composizione parallela di processi. Le relazioni tra i vari processi e il modo in cui ogni processo comunica con l'ambiente in cui viene eseguito sono descritte utilizzando vari operatori algebrici; utilizzando l'approccio algebrico è possibile descrivere in modo compatto iterazioni anche molto complesse.

L'implementazione di CSP in GPars sfrutta la libreria JCSP[28], si tratta di una libreria CSP basata su Java e licenziata sotto la Lesser General Public License (LGPL)[29].
Bisogna tenere presente come, per quanto la LGPL permetta l'utilizzo per fini commerciali del codice a cui è associata, presenta comunque alcune limitazioni che potrebbero indirizzare il programmatore verso l'utilizzo di un altro meccanismo per la gestione della concorrenza, per esempio quello basato sui dataflow, concettualmente molto simile al CSP.
Con l'eccezione costituita da CSP, GPars utilizza la liberal Apache 2 license, decisamente più benigna nei confronti del software commerciale.

5.1) Concetti di base

I concetti fondamentali in CSP sono:

1) Processi
2) Canali
3) Timers
4) Alternative[30] (o scelte)

Si tratta di una lista molto corta, apparentemente molto più corta di quella presente in altri approcci basati sulla concorrenza e sul parallelismo[31], di concetti ed astrazioni di alto livello che permettono all'utente[32] di costruire sistemi paralleli e di ragionare sul loro comportamento.

27 Il concetto di CSP fu introdotto da C. A. R. Hoare[20] nel 1978.
28 Si tratta della JCSP library[21][22] sviluppata all'università di Kent, UK.
29 Questo è vero per lo meno per la JCSP 1.1 rc4
30 In inglese, alternatives
31 Concorrenza e parallelismo sono concetti molto simili ma sottilmente diversi: normalmente con concorrenza si intende descrivere dei processi che devono, in qualche modo, contendersi il tempo di calcolo messo a disposizione da un singolo processore; con parallelismo si intendono dei processi che verranno eseguiti in contemporanea da processori diversi.
Normalmente, in un sistema parallelo, almeno una parte dei processi verranno comunque eseguiti concorrentemente perché il numero di thread software utilizzati è generalmente superiore al numero di thread fisici che possono essere computati contemporaneamente.
32 Ovviamente ci si sta riferendo all'utente del linguaggio di programmazione, non all'utilizzatore del software generato.

Uno degli aspetti fondamentali di questo stile di progettazione di sistemi paralleli è che i processi possono essere collegati in grandi reti il cui comportamento generale può essere previsto.

Processi

Un processo, nella sua forma più semplice, definisce una sequenza di istruzioni da eseguire.
Tipicamente un processo comunicherà con un altro processo utilizzando un canale per trasferire dati da un processo all'altro; in questo modo una rete di processi fornirà collettivamente la soluzione ad un qualche problema.
I processi hanno un solo metodo, run(), utilizzato per invocare il processo.
Una lista di istanze di processi sono passate ad un oggetto PAR che, quando eseguito, causa l'esecuzione parallela di tutti i processi in quella lista; un oggetto PAR termina solo quando tutti i processi nella lista a cui è associato hanno terminato la propria esecuzione.

Un processo incapsula i dati su cui opera, tali dati possono solamente essere inviati a o ricevuti da un altro processo (sono quindi privati) e, per quanto la definizione di un processo sia contenuta in una classe, non esistono metodi esplicitamente definiti che permettano l'accesso dall'esterno ad una qualsiasi proprietà o attributo del processo.

Una rete di processi può essere eseguita concorrente su un singolo processore oppure può essere eseguita su più processori collegati da un qualche tipo di meccanismo di comunicazione, per esempio da una rete basata su TCP/IP; in questo caso (e nel caso di processori multi core) i processi costituenti la rete possono effettivamente essere eseguiti in parallelo per quanto, probabilmente, alcuni di questi saranno comunque eseguiti concorrentemente.
La definizione di processo rimane la medesima sia che il processo sia eseguito concorrentemente o in parallelo e il programmatore non è tenuto a sapere, in fase di progettazione, in che modo sarà eseguito il processo che sta sviluppando.

Canali

Un canale è il metodo con cui un processo comunica con un altro processo; è importante sottolineare come un canale costituisca una connessione unidirezionale punto-punto unbuffered tra due processi: un processo scrive sul canale e un altro legge dal canale.
Se due processi devono comunicare in modo bidirezionale, dovranno essere creati due canali, uno in ciascuna direzione.

Due processi dovranno sincronizzarsi per comunicarsi dei dati: il processo che per primo tenta di comunicare attende, senza consumare risorse, che il processo con cui sta tentando di comunicare si renda disponibile; quando il secondo processo tenterà di comunicare scoprirà questa situazione e accetterà il trasferimento dei dati, successivamente entrambi i processi riprenderanno la loro esecuzione in parallelo.
Non importa se si tratta del processo che deve ricevere i dati o di quello che li deve inviare che tenta di comunicare per primo, il comportamento è simmetrico; il primo processo che tenta di comunicare attende passivamente sul canale: non esistendo comportamenti come il polling di un canale o il busy-wait-loop il processo non genera alcun tipo di overhead durante l'attesa.
Questo descrive il meccanismo di comunicazione fondamentale basato sui canali; è tuttavia possibile creare dei canali che presentano più processi collegati in scrittura o in lettura, la sintassi è leggermente più complessa ma il comportamento ultimo di questi canali è comunque riconducibile ad un sistema di comunicazione punto-punto.

In Groovy bisogna prestare una certa attenzione al passaggio dei dati tramite canali perché, nel caso in cui i processi che stanno comunicano siano in esecuzione sullo stesso processore, quello che effettivamente sarà comunicato sul canale sarà un riferimento ad un oggetto e non l'oggetto stesso; questo implica che il processo inviante potrebbe modificare l'oggetto inviato qualche tempo dopo la comunicazione, con tutti i problemi che ciò comporta.
Per ovviare al problema, si comunica sempre una nuova copia dell'oggetto che si vuole inviare acciocché entrambi i processi possano disporne liberamente.
La comunicazione di oggetti tra processi in esecuzione su processori diversi non soffre di questo problema: il sistema deve in ogni caso eseguire una copia dei dati da inviare perché un puntatore perde di significato quando viene inviato ad un altro processore; gli oggetti comunicati devono implementare l'interfaccia serializable.

I processi in esecuzione sullo stesso processore multi core devono essere considerati come processi in esecuzione concorrente (perché, tra l'altro, probabilmente condividono almeno parte della cache), questo implica che gli oggetti comunicati devono essere delle copie.

Timers

Un aspetto fondamentale del mondo reale è che molti sistemi si basano su un qualche concetto di tempo, sia esso assoluto o relativo.
I timers sono un componente fondamentale nell'ambito della programmazione parallela.
Il tempo è ricavato dal clock di sistema, è preciso al millisecondo e può essere utilizzato tramite vari operatori come un valore assoluto; è per esempio possibile sospendere un processo per un determinato intervallo espresso in millisecondi oppure è impostare degli allarmi per un tempo futuro, permettendo di programmare un qualche tipo di azione da eseguire alla loro scadenza.
Un processo che invochi il metodo sleep() o che sia in attesa di un allarme è sospeso e non utilizza risorse di calcolo.

Alternative

Le iterazioni con il mondo reale sono non deterministiche, questo significa che l'ordine in cui si verificano degli eventi esterni e in cui avvengono delle comunicazioni con l'esterno non può essere di norma noto a priori, l'ambiente di programmazione deve tener conto di questa problematica e fornire al programmatore gli strumenti necessari per gestirla.
Le alternative si occupano esattamente di questo: permettono di selezionare una o più comunicazioni in input oppure un allarme e in genere forniscono strumenti per la gestione della sincronia.
Gli eventi su cui l'alternativa esegue la selezione sono comunemente chiamati “guardie”; se una di queste guardie è disponibile (o pronta), verrà selezionata e il processo associato sarà eseguito.
Se nessuna guardia è disponibile, l'alternativa entra in uno stato di attesa, senza consumare risorse di calcolo, risvegliandosi quando una guardia si renderà disponibile.
Allorché più guardie siano disponibili, ne sarà scelta una in base a qualche criterio di selezione; la possibilità di selezionare una guardia è una caratteristica essenziale di ogni ambiente di programmazione parallela che presuma di poter modellare per lo meno una porzione del mondo reale.

5.2) Produttore – consumatore: un modello fondamentale.

Tradizionalmente, l'approccio che si segue inoltrandosi in un nuovo linguaggio di programmazione è quello di scrivere un programma che stampi a monitor la stringa “Hello World”, modificandolo successivamente acciocché accetti in input una stringa che permetta di trasformarlo in un programma che stampi “Hello name”.
Nell'universo parallelo si crea invece un programma che utilizzi un modello di programmazione fondamentale: il modello produttore – consumatore.

Un produttore è un processo che presenta in uscita una sequenza di dati; un consumatore è un processo che accetta in input una sequenza di dati processandoli poi in qualche modo.
Il semplice accoppiamento di un produttore con un consumatore genera istantaneamente una serie di problemi, ad esempio: cosa accade se il produttore è pronto per generare i dati prima che il consumatore sia pronto a riceverli ? Oppure, cosa accade se il consumatore è pronto per ricevere i dati prima che il produttore possa generarli ?
La prima soluzione che generalmente si adotta per ovviare a problemi di questo tipo è generalmente quella di introdurre un buffer tra produttore e consumatore ma questo serve solo a ritardare l'inevitabile: prima o poi il buffer risulterà pieno, e quindi bisognerà fermare il produttore, o risulterà vuoto, e quindi bisognerà fermare il consumatore.
Si è solo ottenuto di complicare il sistema, senza ottenerne alcun beneficio effettivo, con il risultato di complicarne l'analisi e di aggiungere la possibilità che il buffer generi degli errori indipendenti dal comportamento dei due processi produttore e consumatore.
Fortunatamente, le definizioni di processi e canali permettono di risolvere elegantemente il problema.
Se il processo produttore è collegato al processo consumatore attraverso un canale, i processi si sincronizzeranno affinché la trasmissione dati abbia luogo, quindi se il produttore tenterà di trasmettere dei dati sul canale prima che il consumatore sia pronto per riceverli, il produttore si metterà in attesa finché il consumatore non sarà pronto per la ricezione e vice-versa.

Hello World

Hello World – produttore
La classe ProduceHW implementa l'interfaccia CSProcess in cui è presente il singolo metodo, run(), utilizzato per invocare il processo.
L'interfaccia CSProcess è contenuta nel package org.jcsp.lang.

```
import org.jcsp.lang.*

class ProduceHW implements CSProcess {
      def ChannelOutput outChannel
      void run() {
            def hi = "Hello"
            def thing = "World"
            outChannel.write ( hi )
            outChannel.write ( thing )
      }
}
```

L'unico attributo della classe, outChannel, è il canale su cui il processo convoglia l'output utilizzando il metodo write(); per quanto Groovy non richieda di specificare il tipo degli attributi o delle variabili di volta in volta definite, si preferisce fornire la tipizzazione per favorire la comprensione del codice.

Ogni processo ha un solo metodo, run(); sono definite due variabili che vengono inviate in successione nel canale di output.

Hello World – consumatore
Il processo ConsumeHW ha una proprietà, inChannel di tipo ChannelInput che può essere utilizzata solamente per leggere oggetti dal canale usando il metodo read().
Il metodo run() legge due oggetti dal canale e li stampa a monitor.
La notazione ${v} indica che la variabile v deve essere considerata una stringa.

```
import org.jcsp.lang.*

class ConsumeHW implements CSProcess {
      def ChannelInput inChannel
      void run() {
            def first = inChannel.read()
            def second = inChannel.read()
            println "\n${first} ${second}!\n"
      }
}
```

Hello World – script
La figura mostra il diagramma di rete per questo semplice sistema composto dai due processi ProduceHW e ConsumeHW che comunicano utilizzando un canale chiamato connect.

Script utilizzato per costruire la rete:

```
import org.jcsp.lang.*
import org.jcsp.groovy.PAR
One2OneChannel connect = Channel.createOne2One()
def processList =       [
                              new ProduceHW ( outChannel: connect.out() ),
                              new ConsumeHW ( inChannel: connect.in() )
                        ]
new PAR (processList).run()
```

Il package org.jcsp.groovy contiene le definizioni delle classi che facilitano l'uso di JCSP[33] in Groovy; nello specifico: la classe PAR permette di lanciare l'esecuzione di una lista di processi parallelamente, questo è ottenuto eseguendo il metodo run() della classe PAR il quale a sua volta invoca il metodo run() di ogni processo presente nella lista fornita a PAR.

Il canale connect è un'interfaccia di tipo One2OneChannel, il canale in sé viene creato utilizzando il metodo createOne2One della classe Channel[34].
La lista processList è composta da un'istanza di ProducerHW e da un'istanza di ConsumerHW, i rispettivi attributi sono: outChannel, che collega l'output del processo al canale connect (utilizzando connect.out()) e inChannel, che collega l'input del processo al canale connect.
Il package JCSP implementa, per quanto possibile, la connessione dei processi in rete in modo che

33 JCSP: Communicating Sequential Processes for Java
34 a sua volta contenuta nel package org.jcsp.lang

sia facile da verificare la correttezza delle connessioni.
Per esempio, il canale connect è definito come interfaccia di di One2OneChannel e quindi deve avere un'entrata e un'uscita che sono impostate usando i metodi in() e out() rispettivamente.
Se una classe contiene un attributo di tipo ChannelOutput, l'attributo deve essere inizializzato con una chiamata ad out() all'atto dell'invocazione del processo ed è permesso l'utilizzo del solo metodo write() su quello specifico attributo di canale; viceversa per un canale di input.
Nei diagrammi che illustrano le reti di processi, la punta delle frecce è associata all'estremità che rappresenta l'input del canale nel processo.

Hello Name

Il sistema Hello Name è una semplice estensione di Hello World, l'unica differenza essendo nel processo ProducerHN: ora il processo chiede all'utente il nome da visualizzare e lo invia al processo consumer utilizzando la variabile thing.

```
import phw.util.*
import org.jcsp.lang.*
class ProduceHN implements CSProcess {
      def ChannelOutput outChannel
      void run() {
            def hi = "Hello"
            def thing = Ask.string ("\nName ? ")
            outChannel.write ( hi )
            outChannel.write ( thing )
      }
}
```

Il package phw.util contiene alcuni semplici metodi per l'interfacciamento con la consolle che possono essere utilizzati per ottenere input dall'utente. Il metodo Ask.string stampa a consolle Name ? su una nuova riga, la risposte dell'utente viene memorizzata nella variabile thing.
Il processo consumatore rimane identico, eccezion fatta per il nome, che diventa ConsumeHN, lo stesso vale per lo script che costruisce la rete: l'unica variazione è costituita dal nome dei due processi.

Elaborazione di un semplice flusso di dati

Il sistema in questo esempio richiede che l'utente inserisca un flusso di numeri interi in un processo produttore che li trasmetterà ad un processo consumatore, il quale li stamperà a monitor.

Processo produttore:

```
import phw.util.*
import org.jcsp.lang.*
class Producer implements CSProcess {
      def ChannelOutput outChannel
      void run() {
            def i = 1000
            while ( i > 0 ) {
                  i = Ask.Int ("next: ", -100, 100)
                  outChannel.write (i)
            }
      }
}
```

Il metodo run() è strutturato utilizzando un ciclo while che terminerà appena l'utente inserirà uno zero; il numero intero in input è ottenuto utilizzando il metodo Ask.int() che assicurerà che ogni numero inserito sia compreso tra -100 e +100.
Il ciclo while è costruito in modo da assicurare che anche lo zero inserito sia inviato nel canale.

Processo Consumatore:

```
import org.jcsp.lang.*
class Consumer implements CSProcess {
      def ChannelInput inChannel
      void run() {
            def i = 1000
            while ( i > 0 ) {
                  i = inChannel.read()
                  println "the input was : ${i}"
            }
            println "Finished"
      }
}
```

Il processo consumatore processa i dati ricevuti dal canale stampandoli a monitor; quando viene ricevuto uno zero il ciclo while termina scrivendo “Finished” a schermo.

Lo script che costruisce la rete di questo sistema è molto simile a quello dell'esempio precedente, com'è intuibile trattandosi sempre di una rete formata da due processi collegati da un singolo canale.

```
import org.jcsp.lang.*
import org.jcsp.groovy.*
One2OneChannel connect = Channel.createOne2One()
def processList = [
      new Producer ( outChannel: connect.out() ),
      new Consumer ( inChannel: connect.in() )
      ]
new PAR (processList).run()
```

L'output a monitor di questa semplice rete di processi è decisamente confuso, dato che entrambi stampano sulla consolle concorrentemente; esiste un processo chiamato GConsole, situato nel package org.jcsp.groovy.plugAndPlay, che crea una consolle divisa in aree di input e di output.

5.3) Reti di processi

Uno dei principali vantaggi del modo in cui il CSP è stato implementato in JCSP è che i processi possono essere composti in modo estremamente semplice.
In aritmetica, il significato di 1+2+3 è immediatamente ovvio, la composizione di processi è egualmente semplice; è possibile costruire un insieme di blocchi fondamentali[35], utilizzabili per costruire sistemi complessi.
I processi utilizzati come blocchi costruttivi sono contenuti nel package org.jcsp.groovy.plugAndPlay.

35 Questi processi di base sono costruiti sulla base di quelli contenuti nel package org.jcsp.plugNplay, questo è evidenziato dalla G presente nel loro nome.

Prefisso

Il processo GPrefix inizia scrivendo prefixValue in outChannel, continua poi riportando in outChannel tutto quello ricevuto da inChannel utilizzando un ciclo infinito.
L'attributo prefixValue, che può essere impostato alla creazione del processo, contiene il valore iniziale.

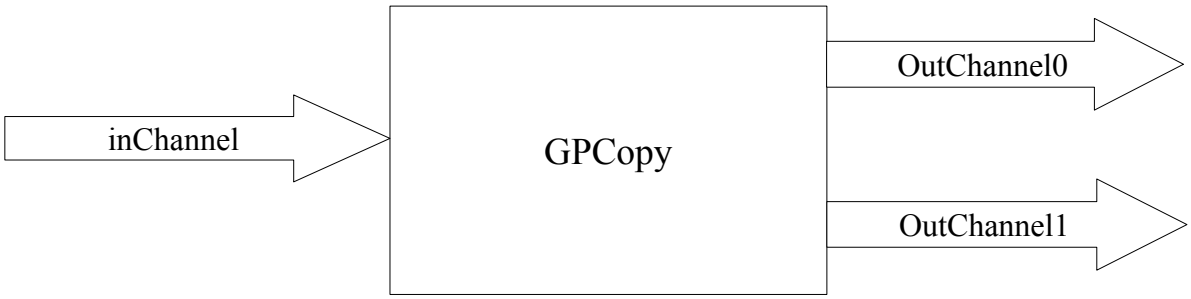
```
import org.jcsp.lang.*
class GPrefix implements CSProcess {
      def int prefixValue = 0
      def ChannelInput inChannel
      def ChannelOutput outChannel
      void run () {
            outChannel.write(prefixValue)
            while (true) {
                  outChannel.write( inChannel.read() )
            }
      }
}
```

Incremento

Il processo GSuccessor legge i valori forniti dal canale inChannel, li incrementa e li riporta in outChannel ripetendo tutto ciò indefinitamente.

```
import org.jcsp.lang.*
class GSuccessor implements CSProcess {
      def ChannelInput inChannel
      def ChannelOutput outChannel
      void run () {
            while (true) {
                  outChannel.write( inChannel.read() + 1 )
            }
      }
}
```

Copia



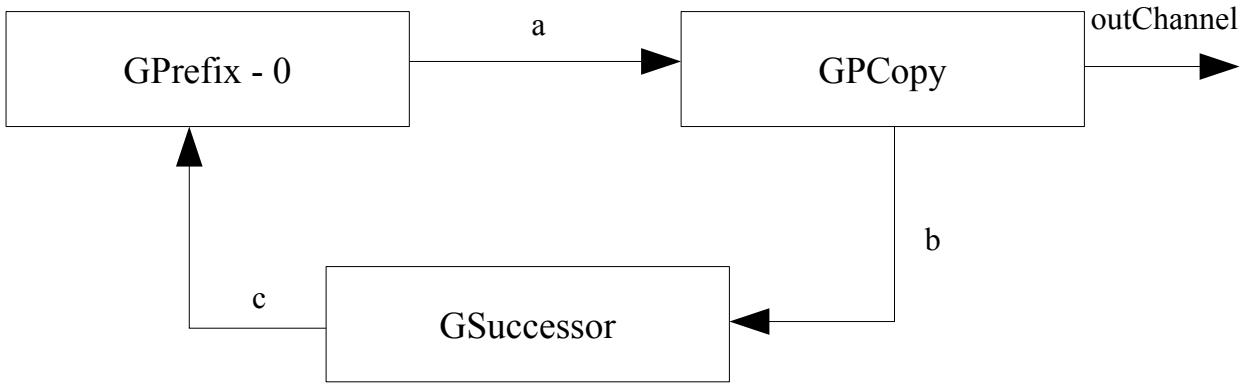
Il processo GPCopy riporta parallelamente una copia di ogni valore ricevuto da inChannel su outChannel0 e outChannel1, tutto questo è ripetuto indefinitamente.
Eseguire questa operazione in parallelo implica ricevere la garanzia che:

1) non importa in quale ordine i valori siano letti dai processi utilizzatori che si suppone collegati ai due canali outChannel
2) non si legge un altro valore da inChannel finché l'output non sia stato completato perché PAR non termina prima della lista di processi che sta eseguendo.

```
import org.jcsp.plugNplay.ProcessWrite
import org.jcsp.lang.*
import org.jcsp.groovy.*
class GPCopy implements CSProcess {
      def ChannelInput inChannel
      def ChannelOutput outChannel0
      def ChannelOutput outChannel1
      void run () {
            def write0 = new ProcessWrite ( outChannel0)
            def write1 = new ProcessWrite ( outChannel1)
            def parWrite2 = new PAR ( [ write0, write1 ] )
            while (true) {
                  def i = inChannel.read()
                  write0.value = i
                  write1.value = i
                  parWrite2.run()
            }
      }
}
```

GPCopy utilizza il processo ProcessWrite, preso da org.jcsp.plugNplay definendone due istanze collegate ai canali di output e raggruppandole in un PAR chiamato parWrite2.
Ogni istanza di ProcessWrite presenta un campo “value” pubblico contenente il valore da scrivere in output.
Il ciclo while infinito legge un valore dal canale di input, lo copia nei campi value delle due istanze di ProcessWrite ed esegue parWrite2; parWrite2 causa la scrittura del valore letto in ingresso nei due canali di uscita in parallelo e termina quando le due istanze di ProcessWrite che lo compongono hanno completato il compito assegnato.

Generazione di una sequenza di interi

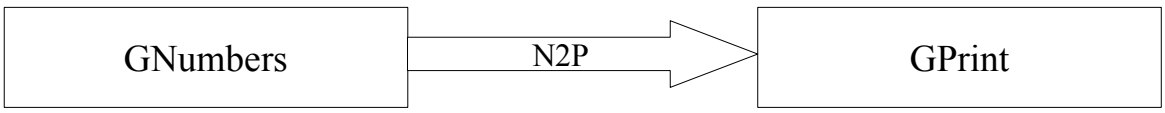


I tre processi, GPrefix, GSuccessor e GPCopy possono essere utilizzati per creare una rete che fornisca una sequenza di numeri interi in output.

```
import org.jcsp.lang.*
import org.jcsp.groovy.*
class GNumbers implements CSProcess {
      def ChannelOutput outChannel
      void run() {
            One2OneChannel a = Channel.createOne2One()
            One2OneChannel b = Channel.createOne2One()
            One2OneChannel c = Channel.createOne2One()
            def numbersList = [ new GPrefix (   prefixValue: 0,
                                                inChannel: c.in(),
                                                outChannel: a.out() ),
                                new GPCopy      (  inChannel: a.in(),
                                                outChannel0: outChannel,
                                                outChannel1: b.out() ),
                                new GSuccessor (  inChannel: b.in(),
                                                outChannel: c.out())
                                ]
            new PAR ( numbersList ).run()
      }
}
```

Il processo GNumbers ha un singolo attributo outChannel attraverso il quale fornisce il flusso di numeri; sono presenti tre canali a, b, c definiti come interfacce One2OneChannel, sono utilizzati per connettere internamente i processi come rappresentato dal diagramma di rete.
La fase di progettazione si riduce alla creazione del diagramma di rete, utilizzandolo successivamente per la definizione dei canali richiesti per la connessione dei processi previsti.
Il compilatore sarà in grado di verificare, utilizzando le interfacce specificate, che l'estremità di input di un canale specificata utilizzando il metodo in() sia collegata ad un attributo di tipo ChannelInput, lo stesso vale per le estremità di output; questo grazie al fatto che i tipi degli attributi di canale siano stati specificati, per quanto non richiesto in Groovy.

Test di GNumbers
Un semplice test per verificare la correttezza del funzionamento di GNumbers consiste nello stampare a monitor l'output della rete di processi da cui è costituito, a questo scopo si utilizza un processo GPrint.
GPrint è dotato di un attributo inChannel di tipo ChannelInput e un attributo “heading”, di tipo String, contenente un titolo per il flusso da stampare.




```
import org.jcsp.lang.*
import org.jcsp.groovy.*
One2OneChannel N2P = Channel.createOne2One()
def testList =    [     new GNumbers   (  outChannel: N2P.out() ),
                        new GPrint     (  inChannel: N2P.in(),
                                          heading : "Numbers" )
                  ]
new PAR ( testList ).run()
```

Il canale N2P è utilizzato per collegare GNumbers a GPrint; all'interno della lista di processi testList viene inizializzato l'attributo “heading” di GPrint.

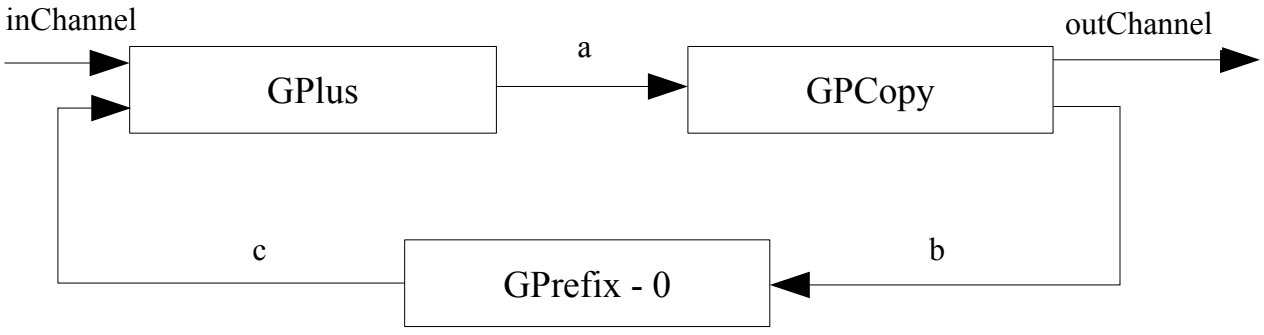
Somma cumulativa

Utilizzando il flusso di interi fornito da GNumbers come input di un processo GIntegrate, è possibile ottenere un processo che fornisca in uscita la sommatoria dei numeri via via ricevuti in ingresso.
Per ottenere ciò, è necessario disporre di un processo che si preoccupi di sommare i numeri ricevuti in ingresso, questo servizio è fornito da GPlus.

```
import org.jcsp.plugNplay.ProcessRead
import org.jcsp.lang.*
import org.jcsp.groovy.*
class GPlus implements CSProcess {
      def ChannelOutput outChannel
      def ChannelInput inChannel0
      def ChannelInput inChannel1
      void run () {
            ProcessRead read0 = new ProcessRead ( inChannel0 )
            ProcessRead read1 = new ProcessRead ( inChannel1 )
            def parRead2 = new PAR ( [ read0, read1 ] )
            while (true) {
                  parRead2.run()
                  outChannel.write(read0.value + read1.value)
            }
      }
}
```

Il processo GPlus legge da due canali utilizzando una tecnica simile a quella utilizzata in GCopy e scrive in output la somma dei valori letti.
Le due istanze di ProcessRead si preoccupano di leggere dai due canali di input in parallelo, grazie al raggruppamento effettuato da parRead2; il campo ProcessRead.value fornisce il valore letto dall'input.

GIntegrate
Il processo GPlus necessita di due valori, quindi dovrà essere inizializzato utilizzando un processo GPrefix, ad esempio, mentre l'altro valore sarà fornito dall'input del processo GIntegrate.
Quando un valore di ingresso sarà disponibile, il processo GPlus si sbloccherà, calcolerà la somma dei due valori in ingresso e la trasferirà al processo GPCopy attraverso il canale a.
GPCopy si preoccuperà di riportare il valore ricevuto in uscita e, attraverso il canale b, a GPrefix che, a sua volta, lo riporterà a GPlus attraverso il canale c.



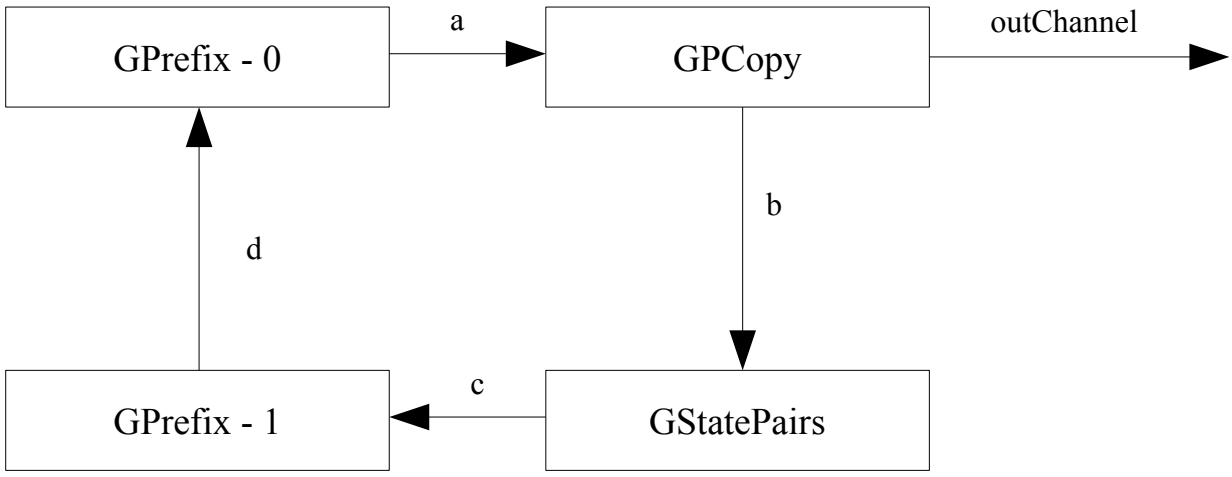

```
import org.jcsp.lang.*
import org.jcsp.groovy.*
```

```
class GIntegrate implements CSProcess {
      def ChannelOutput outChannel
      def ChannelInput inChannel
      void run() {
            One2OneChannel a = Channel.createOne2One()
            One2OneChannel b = Channel.createOne2One()
            One2OneChannel c = Channel.createOne2One()
            def integrateList = [   new GPrefix     ( prefixValue: 0,
                                                      outChannel: c.out(),
                                                      inChannel: b.in() ),
                                    new GPCopy      ( inChannel: a.in(),
                                                      outChannel0: outChannel,
                                                      outChannel1: b.out() ),
                                    new GPlus       ( inChannel0: inChannel,
                                                      inChannel1: c.in(),
                                                      outChannel: a.out() )
                                ]
            new PAR ( integrateList ).run()
      }
}
```

La successione di Fibonacci

Nella successione di Fibonacci, ogni elemento è dato dalla somma dei due elementi che lo precedono, i primi due elementi della successione devono essere inizializzati e vengono tipicamente impostati a zero e ad uno rispettivamente, possono tuttavia assumere un valore arbitrario.
I valori iniziali possono essere forniti dal processo GPrefix, tuttavia è necessario creare un processo in grado di sommare gli ultimi due numeri ricevuti e che si curi di sostituire il più vecchio numero ricevuto con l'input successivo.

Diagramma della rete di processi che implementa la generazione della successione dei numeri di Fibonacci:



Inizialmente, GPrefix – 0 è l'unico processo che può essere completato perché è l'unico in grado di comunicare un output, GPCopy è in attesa di un input così come GStatePairs, GPrefix – 1 sta tentando di consegnare un output ma non è in grado di farlo finché GPrefix – 0 non richiederà una

lettura dal suo canale di ingresso, cosa che farà non appena avrà consegnato il suo primo output a GPCopy.

GStatePairs
Il processo GStatePairs inizialmente legge due numeri dal canale di input, inChannel, li somma e li restituisce in output, sovrascrive poi il primo valore con il secondo e reitera il processo, leggendo un altro numero dal canale di input.

```
class GStatePairs implements CSProcess {
      def ChannelOutput outChannel
      def ChannelInput inChannel
      void run() {
            def n1 = inChannel.read()
            def n2 = inChannel.read()
            while (true) {
                  outChannel.write ( n1 + n2 )
                  n1 = n2
                  n2 = inChannel.read()
            }
      }
}
```

FibonacciV1
Il listato sotto riportato implementa direttamente il diagramma di rete: vi sono definiti i quattro canali utilizzati e i processi strutturati nella solita lista eseguita poi con un PAR.

```
class FibonacciV1 implements CSProcess {
      def ChannelOutput outChannel
      void run () {
            One2OneChannel a = Channel.createOne2One()
            One2OneChannel b = Channel.createOne2One()
            One2OneChannel c = Channel.createOne2One()
            One2OneChannel d = Channel.createOne2One()
            def testList = [  new GPrefix    (  prefixValue: 0,
                                                inChannel: d.in(),
                                                outChannel: a.out() ),
                              new GPrefix    (  prefixValue: 1,
                                                inChannel: c.in(),
                                                outChannel: d.out() ),
                              new GPCopy     (  inChannel: a.in(),
                                                outChannel0: b.out(),
                                                outChannel1: outChannel ),
                              new GStatePairs ( inChannel: b.in(),
                                                outChannel: c.out() ),
                           ]
            new PAR ( testList ).run()
      }
}
```

Il funzionamento della rete può essere testato utilizzando un semplice script:

```
One2OneChannel F2P = Channel.createOne2One()
def testList =    [     new FibonacciV1 ( outChannel: F2P.out() ),
                        new GPrint      ( inChannel: F2P.in(),
                                          heading: "Fibonacci V1")
                  ]
new PAR ( testList ).run()
```

Output generato dallo script:

```
Fibonacci V1
0
1
1
2
3
5
8
13
21
34
55
89
```

Questa soluzione, per quanto apparentemente corretta, presenta un paio di problemi:

- il processo definito per GStatePairs contiene dei valori che sono mantenuti tra un'iterazione del processo e la successiva, effettivamente introducendo uno stato interno nel processo,
- il processo GStatePairs esegue un'operazione, in questo caso una somma, che è già stata definita nel processo GPlus.

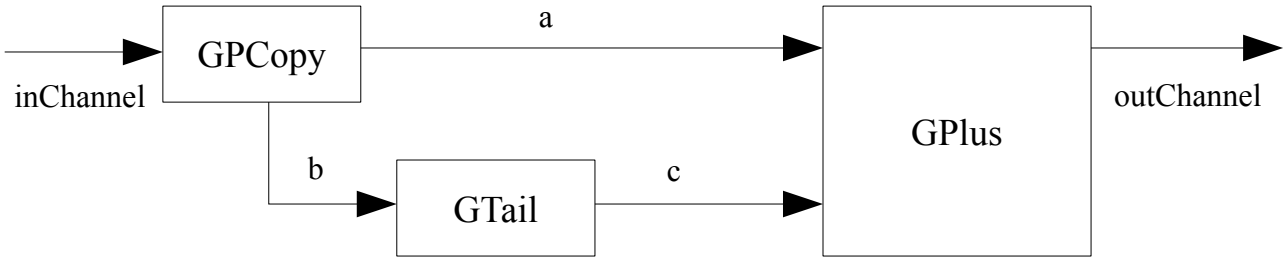
Il primo problema, che può sembrale marginale, diventa molto significativo nel caso in cui si abbia la necessità di modificare la rete, specie nel caso in cui il comportamento della rete si debba adattare dinamicamente all'input.

Utilizzo di GPlus per il calcolo della successione di Fibonacci

Lo scopo è eliminare la necessità di mantenere uno stato interno in uno o più processi della rete, si ha tuttavia la necessità di ritardare in qualche modo la lettura dell'elemento generato dal ciclo precedente della rete da parte di GPlus.
Un aiuto in questo senso viene fornito dal processo GTail.

```
class GTail implements CSProcess {
      def ChannelOutput outChannel
      def ChannelInput inChannel
      void run () {
            inChannel.read()
            while (true) {
                  outChannel.write( inChannel.read() )
            }
      }
}
```

IL processo GTail legge il primo elemento dall'input e lo ignora, successivamente riporta in output ogni elemento letto in input.
Come può ciò fornire la soluzione cercata ?
La risposta è da ricercare in come GPCopy esegue l'output.



GPCopy esegue l'output parallelamente sui due canali ma la coppia di operazioni di output è strutturata in modo da non poter completare finché l'output fornito sui due canali non viene accettato dai processi connessi all'altra estremità dei canali stessi.
Il comportamento totale della rete risulta dunque essere:

1) Primo ciclo: GPCopy riceve il primo input, lo copia sui due canali.
2) GTail riceve l'input da GPCopy e lo ignora mentre GPlus riceve l'input da GPCopy e rimane in attesa dell'input da GTail, la procedura di output di GPCopy termina.
3) Secondo ciclo: GPCopy accetta un nuovo input e lo riporta nei due canali, GTail accetta l'input da GPCopy e lo riporta diligentemente in output sul canale c.
4) GPlus non può al momento accettare il secondo input da GPCopy perché era ancora in attesa dell'input da GTail.
5) Essendo l'input da GTail disponibile, GPlus esegue la sua computazione, fornisce in output il risultato, accetta il nuovo input da GPCopy e rimane nuovamente in attesa dell'input da GTail.
6) GPCopy ha finalmente consegnato l'output su entrambi i canali ed è quindi pronto ad eseguire un altro ciclo.

Grazie a questo meccanismo, il processo GPlus riceve in input il valore corrente da GTail e il valore della computazione precedente da GPCopy; il meccanismo sfrutta il fatto che l'ordine in cui le comunicazioni avvengono su più canali in parallelo non è importante.
Il modo in cui sono implementati i canali assicura che non ci sia perdita di dati; si è riusciti ad ottenere una soluzione equivalente a quella fornita da GStatePairs senza avere la necessità di conservare lo stato interno di alcun processo.

Listato che implementa la rete:

```
class GPairs implements CSProcess {
      def ChannelOutput outChannel
      def ChannelInput inChannel
      void run() {
            One2OneChannel a = Channel.createOne2One()
            One2OneChannel b = Channel.createOne2One()
            One2OneChannel c = Channel.createOne2One()
            def pairsList =  [      new GPlus       ( outChannel: outChannel,
                                                      inChannel0: a.in(),
                                                      inChannel1: c.in() ),
                                    new GPCopy      ( inChannel: inChannel,
                                                      outChannel0: a.out(),
                                                      outChannel1: b.out() ),
                                    new GTail       ( inChannel: b.in(),
                                                      outChannel: c.out() )
                              ]
            new PAR ( pairsList ).run()
      }
}
```

La modifica da apportare alla classe FibonacciV1 per convertirla nella versione stateless consiste nel sostituire “GStatePairs” con “GPairs”.

Morale:
Bisogna sempre cercare di riutilizzare i processi esistenti nella costruzione di una nuova rete, questo è usualmente il miglior modo di risolvere un problema; modificare o estendere un processo esistente in genere comporta un incremento della difficoltà di comprendere il comportamento della rete.

Output su consolle da più processi contemporaneamente

Ci sono molti casi in cui è necessario stampare su consolle l'output di un gruppo di processi in esecuzione parallela in un modo che sia consistente con l'attuale stato di esecuzione dei vari processi del gruppo.
Il processo GParPrint fornisce questo servizio leggendo un certo numero di input in parallelo e stampandoli su consolle separati da tabulazioni in modo da stampare una linea sulla consolle per ogni set di input ricevuti

```
import org.jcsp.lang.*
import org.jcsp.groovy.*
import org.jcsp.plugNplay.ProcessRead
class GParPrint implements CSProcess {
      def ChannelInputList inChannels
      def List headings
      def long delay = 200
      void run() {
            def inSize = inChannels.size()      //numero di canali di input
            def readerList = []
            //costruisce la lista dei processi di input
            (0 ..< inSize).each { i ->
                  readerList [i] = new ProcessRead ( inChannels[i] )
            }
            def parRead = new PAR ( readerList )
            //stampa degli headings a consolle
            if ( headings == null ) {
                  println "No headings provided"
            }
            else {
                  headings.each { print "\t${it}" }
                  println ()
            }
            def timer = new CSTimer()
            while ( true) {
                  parRead.run()     //legge la lista di input
                  /*l'esecuzione continua solo quando tutti gli input sono stati
                  letti */
                  readerList.each { pr -> print "\t" + pr.value.toString() }
                  println ()
                  if (delay != 0 ) {
                        timer.sleep ( delay)
                  }
            }
      }
}
```

ChannelInputList: si tratta di una delle classi helper introdotte dal package org.jcsp.groovy, permette di gestire semplicemente gruppi di canali.
La lista headings viene utilizzata per stampare in consolle il titolo delle colonne di dati.
L'attributo delay è utilizzato per introdurre un ritardo tra la stampa di righe di valori successive con lo scopo di permettere una più facile lettura dell'output.
readerList contiene la lista dei processi utilizzati per leggere parallelamente da ciascuno degli inChannels.

Riassumendo, si è visto come i processi possono essere composti e come, utilizzando processi singolarmente estremamente semplici, sia possibile costruire strutture complesse.

5.4) Input non deterministico - alternative

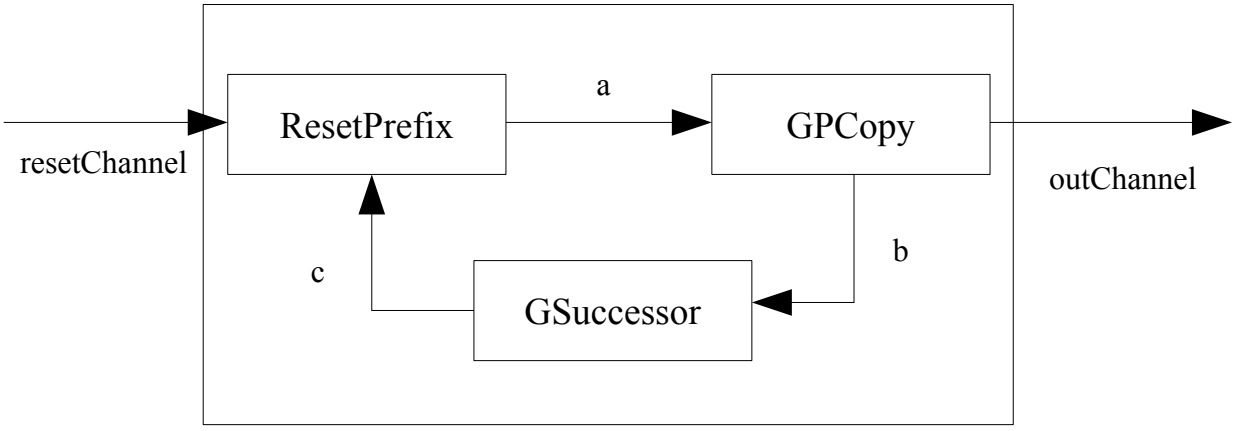
Si è visto come un sistema sia costituito da reti di processi raggruppati in blocchi a loro volta connessi tra loro.
Si consideri ora un sistema costituito da una sequenza di processi; può sorgere la necessità, qualora il sistema gestisca un processo industriale, di ricevere una retroazione o un feedback dal fondo della sequenza, per esempio per forzare la ricalibrazione automatica di uno dei macchinari che si sta controllando.
Il processo controllante il macchinario che può ricevere la richiesta di calibrazione è a conoscenza del fatto che potrebbe ricevere un input dal fondo della catena ma non è a conoscenza di quando questo avverrà e dovrà essere, quindi, in grado di riceverlo ad ogni momento.
Il comportamento di un processo di questo tipo è detto "non deterministico" perché non è possibile determinare quando verrà ricevuto l'input dal fondo della catena durante la fase di definizione del processo, si sa che il feedback potrebbe arrivare ma non si sa quando.
Una rete di processi potrebbe anche essere soggetta ad un intervento esterno che ne modifichi il comportamento, non è noto se e quando questo intervento avrà luogo.

Le alternative permettono di gestire casi di questo genere: nella sua forma più semplice l'alternativa gestisce un gruppo di canali di input.
Durante l'esecuzione dell'alternativa lo stato di tutti i canali di input viene determinato:

- se non sono disponibili input su nessuno dei canali l'alternativa si mette in attesa di un input
- se è presente un solo input il codice associato sarà eseguito
- se invece sono presenti più input su canali diversi l'alternativa sceglierà quale eseguire secondo un criterio e il codice associato a quel canale di input sarà eseguito.

Tipicamente l'alternativa è inglobata in un ciclo in modo da permettere il controllo periodico dei canali di input.

Esempio: ResetNumbers
Si tratta del processo GNumbers modificato introducendo la possibilità di resettare la sequenza di numeri generata ad un valore ricevuto in ingresso.



ResetNumbers

La modifica consiste nell'aver sostituito il processo GPrefix con il processo ResetPrefix, il cui listato è riportato di seguito.
La rete evidenzia come ResetPrefix sia connesso a due canali di input.

```
class ResetPrefix implements CSProcess {
      def int prefixValue = 0                  //valore iniziale della sequenza
      def ChannelOutput outChannel
      def ChannelInput inChannel
      def ChannelInput resetChannel
      void run () {
            def guards = [ resetChannel, inChannel ] //lista di guardie
            def alt = new ALT ( guards )       //definizione dell'alternativa
            outChannel.write(prefixValue)
            while (true) {
                  def index = alt.priSelect()
                  if (index == 0 ) {           // resetChannel input
                        def resetValue = resetChannel.read()
                        //scarta l'input normale
                        def inputValue = inChannel.read()
                        outChannel.write(resetValue)
                  }
                  else {                       //inChannel input
                        def inputValue = inChannel.read()
                        outChannel.write(inputValue)
                  }
            }
      }
}
```

Un'alternativa è costituita da un certo numero di guardie a cui sono associati dei comandi, in questo caso una guardia è semplicemente un canale di input e i comandi associati sono rappresentati dal codice associato al canale.
L'ordine con cui le guardie sono elencate nella lista è importante perché specifica l'ordine di priorità secondo cui verranno scelte.
Il metodo priSelect(), applicato all'alternativa, fornisce l'indice del canale selezionato tra quelli in cui un input è disponibile; nella fattispecie, se un input è disponibile su resetChannel l'indice fornito sarà 0, se non è disponibile un input su resetChannel ma è disponibile un input su inChannel l'indice fornito sarà 1.

Listato di ResetNumbers:
Come risulta evidente, si tratta di un'implementazione diretta delle rete.

```
class ResetNumbers implements CSProcess {
      def ChannelOutput outChannel
      def ChannelInput resetChannel
      def int initialValue = 0
      void run() {
            One2OneChannel a = Channel.createOne2One()
            One2OneChannel b = Channel.createOne2One()
            One2OneChannel c = Channel.createOne2One()
            def testList = [ new ResetPrefix ( prefixValue: initialValue,
                                               outChannel: a.out(),
                                               inChannel: c.in(),
                                               resetChannel: resetChannel ),
                             new GPCopy ( inChannel: a.in(),
                                          outChannel0: outChannel,
                                          outChannel1: b.out() ),
                             new GSuccessor  (  inChannel: b.in(),
                                               outChannel: c.out())
                           ]
            new PAR ( testList ).run()
      }
}
```

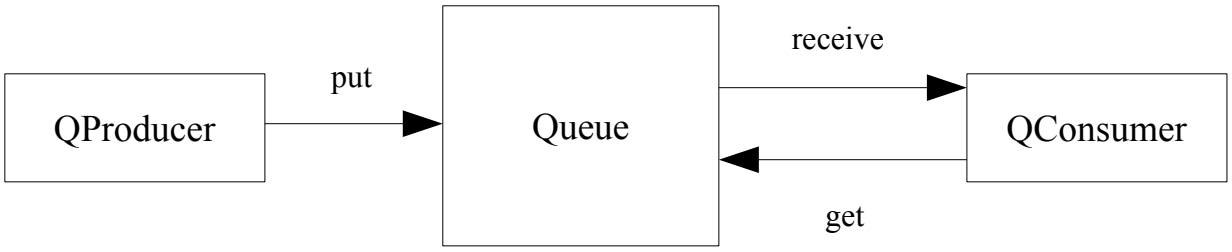
Soddisfare delle condizioni iniziali

Si è visto come introdurre la possibilità di scegliere uno tra vari input utilizzando il costrutto "alternative" che permette di eseguire una scelta tra diversi input disponibili utilizzando un criterio arbitrario; si possono tuttavia incontrare dei casi in cui non tutte le alternative tra cui siano possibili scelte siano attuabili, a volte bisogna sottostate a delle condizioni iniziali.

Consideriamo il caso in cui si debba costruire un sistema utilizzando il paradigma produttore-consumatore: il produttore produrrà un dato che il consumatore accetterà in ingresso, entrambi possono essere rappresentati da un processo collegato ad un canale unidirezionale.
Consideriamo lo stesso modello con lo scopo di massimizzare la produttività: risulta immediatamente chiaro che il meccanismo di trasmissione sincrona dei dati tra produttore e consumatore imposto da CSP può costituire un collo di bottiglia nel caso in cui il produttore produca dati a velocità costante ma il consumatore li consumi molto rapidamente in gruppi con lunghi tempi di attesa tra un gruppo e l'altro[36], questo porta alla situazione inefficiente in cui il produttore attende a lungo tra il consumo di diversi gruppi di dati mentre il consumatore attende la disponibilità del dato successivo dopo ogni consumo.

La risoluzione ovvia a questa problematica è l'introduzione di una coda di dimensione finita[37] tra produttore e consumatore, tuttavia si tratta di una struttura che deve sottostare a delle condizioni iniziali:

- un dato può essere inserito in coda solo se c'è spazio disponibile
- un dato può essere estratto dalla coda solo se la coda non è vuota.



Il processo QProducer fornisce una sequenza di numeri interi ad un processo Queue in cui saranno immagazzinati in una lista circolare, il processo QConsumer tenta di ricevere un dato da Queue che verrà fornito se è presente almeno un dato in coda.

```
class QProducer implements CSProcess {
      def ChannelOutput put
      def int iterations = 200
      def delay = 100
      void run () {
            def timer = new CSTimer()
            println "QProducer has started"
            for ( i in 1 .. iterations ) {
                  put.write(i)
                  timer.sleep (delay)
```

36 Per focalizzare il concetto, se pensi ad una catena di montaggio e ad un sistema di trasporto in containers.
37 In ultima analisi, per quanto grande, la capacità di storage di un sistema è sempre di dimensione finita.

```
            }
      put.write(null)
      /*
      *valore utilizzato per indicare agli altri due processi del sistema il
      *completamento della produzione di valori
      */
      }
}
```

I timers presenti in questi processi servono per simulare la discrepanza tra il tempo necessario a produrre un dato e il tempo necessario a consumarlo, il timer delayBust serve a simulare il consumo dei dati "in gruppi" da parte del secondo processo.

```
class QConsumer implements CSProcess {
      def ChannelOutput get
      def ChannelInput receive
      def long delay = 0
      def long delayBust = 5000 //attesa tra due gruppi di dati
      void run () {
            def pause = new CSTimer()
            pause.setAlarm ( delayBust )
            def guards = [ pause, receive ]     //lista di guardie
            def alt = new ALT ( guards )        //definizione dell'alternativa
            def timer = new CSTimer()
            println "QConsumer has started"
            def running = true
            while (running) {
                  get.write(1)      //comunica la disponibilità a consumare dati
                  def index = alt.priSelect()
                  if (index == 0 ) {
                        //utilizzo di pause per sospendere QConsumer
                        println "Suspended for ${delayBust} ms"
                        timer.sleep ( delayBust )
                        pause.setAlarm ( delayBust )
                  }
                  def v = receive.read()
                  println "QConsumer has read ${v}"
                  timer.sleep (delay)
                  if ( v == null ) {    //termina il ciclo quando riceve un null
                        running = false
                  }
            } //end while
      }
}
```

Si noti come sia possibile associare un timer e il sistema di guardie allo scopo di ottenere un processo che reagisca a qualche evento, in questo caso temporale.

Processo Queue:
Il vettore preCon è utilizzato per memorizzare se un nuovo elemento può essere inserito nella lista e se nella lista è presente almeno un elemento.
Inizialmente, preCon [PUT] è impostato a vero perché la lista è libera e, dato che è anche vuota, preCon [GET] è impostato a falso.
La lista "data" fornisce lo spazio di storage per la lista circolare, gli attributi count, front e rear indicano il numero di elementi in coda e le locazioni in cui un dato può essere aggiunto o rimosso.

L'intero processo è eseguito in un ciclo che viene terminato tramite un valore booleano quando

viene comunicato un valore null al processo QConsumer.

Selezione delle guardie:
Il vettore di booleani preCon viene passato al metodo priSelect che ne verificherà i valori, verificherà poi da quali canali sia possibile ricevere un input e costruirà il valore index di conseguenza.

A seconda della guardia selezionata, il processo aggiunge il dato letto dal canale nella posizione "front" della coda oppure estrae dalla posizione "rear" della coda un dato e lo invia nel canale "receive", il valore estratto dalla coda viene poi testato per verificare se il processo debba terminare o meno.
Al termine di ogni iterazione del ciclo, i valori del vettore preCon vengono aggiornati in base ai valori presenti negli attributi "count" e "elements".

```
class Queue implements CSProcess {
      //proprietà dei canali
      def ChannelInput put
      def ChannelInput get
      def ChannelOutput receive

      def int elements = 20   //numero di slot nella coda
      void run() {
            def qAlt = new ALT ( [ put, get ] ) //alternative
            def preCon = new boolean[2]
            def PUT = 0
            def GET = 1
            preCon[PUT] = true
            preCon[GET] = false
            def data = []
            def count = 0
            def front = 0
            def rear = 0

            def running = true
            while (running) {
                  def index = qAlt.priSelect(preCon)
                  switch (index) {

                        case PUT:
                              data[front] = put.read()
                              println "Q: put ${data[front]} at ${front}"
                              front = (front + 1) % elements
                              count = count + 1
                        Break

                        case GET:
                              get.read()
                              receive.write( data[rear])
                              if (data[rear] == null) {
                                    running = false
                              }
                              rear = (rear + 1) % elements
                              count = count - 1
                        break
                  }

                  preCon[PUT] = (count < elements)
```

```
                preCon[GET] = (count > 0 )

            } // end While
            println "Q finished"
        } //end run
}
```

Il beneficio di questa implementazione del costrutto “alternative” è che il funzionamento del processo viene modificato a seconda e condizioni contenute nel vettore delle condizioni iniziali: se preCon [PUT] è falso la coda è piena, non sarà possibile leggere dal canale “put” anche se QProducer sta cercando di fornire un output; similmente, se preCon [GET] è falso la coda è vuota e nessun segnale sarà accettato dal canale “get” anche se QConsumer ha tentato di scrivercelo.

5.5) Deadlock

Il deadlock è una situazione in cui una rete costituita da due o più processi entra in uno stato in cui è impossibile continuarne l'esecuzione.
Il caso più semplice in cui questo avviene è quando due processi necessitano del controllo di due risorse ciascuno ed entrambi ne acquisiscono una, attendendo poi in eterno che l'altro processo liberi la seconda risorsa di cui necessitano.
Un concetto simile è quello di livelock, che si verifica quando il comportamento della parte di una rete di processi impedisce ad alcuni processi della rete di continuare la propria esecuzione mentre altri possono procedere senza problemi.

Preso un sistema, affinché un deadlock si verifichi devono essere soddisfatte alcune condizioni:

- **Mutua esclusione:**
  Almeno una risorsa deve essere non condivisibile: in ogni momento, solo un processo può utilizzare la risorsa.
- **Hold and wait o resource holding:**
  Mentre un processo ha il controllo di una risorsa richiede il controllo di almeno un'altra risorsa controllata da un secondo processo.
- **No preemption:**
  Il sistema operativo non può forzare il rilascio di una risorsa una volta allocata, le risorse allocate devono essere rilasciate volontariamente dai processi che le controllano.
- **Circular wait:**
  Un processo è in attesa di una risorsa allocata da un secondo processo che, a sua volta, è in attesa della deallocazione di una risorsa controllata dal primo processo.
  In genere questo si verifica su una catena di processi in cui il processo P1 è in attesa del rilascio di una risorsa da un processo P2 che a sua volta sta attendendo il rilascio di una risorsa da un processo P3 che a sua volta sta aspettando che P1 deallochi una risorsa che sta controllando.

Queste quattro condizioni sono note come Coffman conditions.
La mancata verifica di una sola di queste condizioni è sufficiente per prevenire il deadlock.

Prevenzione

La prevenzione del deadlock si basa sulla strutturazione di un sistema in modo da prevenire il verificarsi di almeno una tra le condizioni di Coffman.

- Rimuovere la condizione di mutua esclusione implica che nessun processo può avere il controllo esclusivo di una risorsa. Gli algoritmi che evitano la mutua esclusione sono detti

non-blocking[38].

- La condizione di hold and wait o resource holding può essere eliminata imponendo ad un algoritmo di procedere all'acquisizione di tutte le risorse richieste prima di iniziare la propria esecuzione o prima di eseguire una particolare sequenza di operazioni.
  Questa situazione è difficilmente verificabile: un processo non può sempre essere a conoscenza di quali risorse necessiterà e, in ogni caso, questo costituisce un utilizzo inefficiente delle risorse disponibili.
  Un'altra possibilità è permettere ad un processo di acquisire risorse solo quando non ne sta controllando nessuna, questo significa che un processo, che sta controllando le risorse a e b, se necessiterà di una risorsa c durante la sua esecuzione, dovrà rilasciare le risorse a e b e dovrà poi tentare di riacquisire a, b e di acquisire c.
  Questo approccio risulta poco pratico perché spesso comporta problemi di resource starvation.
- La condizione di no preemption può essere impossibile da evitare perché un processo deve essere in grado di disporre di una risorsa almeno per un tempo limitato o il risultato del processo potrebbe essere inconsistente.
  L'impossibilità di eseguire il preemption, tuttavia, potrebbe interferire con l'esecuzione di un algoritmo prioritario, tipico di un sistema in cui sia necessario eseguire delle operazioni in tempo reale.
  Il preemption di una risorsa implica generalmente la necessità di effettuare un rollback, operazione estremamente costosa.
  Algoritmi che permettono il preemption sono di tipo lock-free, wait-free e optimistic concurrency control[39].
- Circular wait. Gli approcci che permettono di evitare situazioni di circular wait includono:
  a) disabilitare gli interrupt durante l'esecuzione di una regione critica
  b) utilizzare un ordinamento gerarchico almeno parziale delle risorse; se non è disponibile un ordinamento gerarchico delle risorse verranno utilizzati i corrispettivi indirizzi di memoria e le risorse andranno richieste seguendo un ordinamento gerarchico crescente[40].

Deadlock su produttore e consumatore

Vediamo ora quanto sia semplice introdurre un problema di deadlock nella struttura produttore-consumatore.

I due processi BadP e BadC sono collegati da due canali, iniziano stampando una stringa su consolle, poi entrano in un loop in cui:

1) comunicano che stanno per eseguire un output
2) tentano di eseguire l'output
3) comunicano che stanno per accettare un input
4) tentano di acquisire un input
5) comunicano che si sta iniziando un altro ciclo

```
class BadP implements CSProcess {
      def ChannelInput inChannel
      def ChannelOutput outChannel
      def void run() {
```

38 Per quanto, ultimamente l'utilizzo del termine non-blocking in un algoritmo indichi che la sospensione di uno o più thread (o task) non impedisca ai restanti thread di procedere nell'esecuzione.

39 Si tratta di algoritmi che, data una risorsa condivisa vasta (ad es un database) presuppongono che diverse transazioni possano essere eseguite contemporaneamente senza influenzarsi, un approccio simile è seguito da STM

40 Si veda come esempio [23].

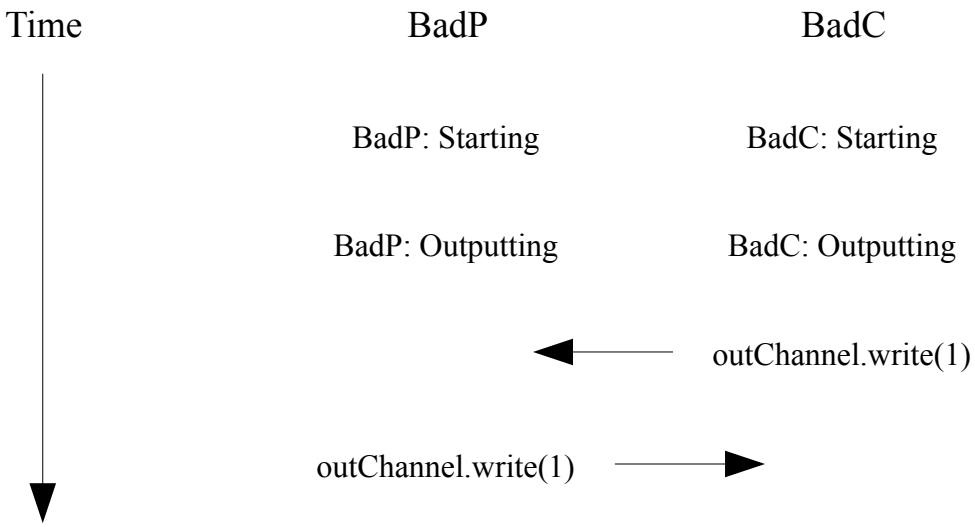
```
            println "BadP: Starting"
            while (true) {
                  println "BadP: outputting"
                  outChannel.write(1)
                  println "BadP: inputting"
                  def i = inChannel.read()
                  println "BadP: looping"
            }
      }
}

class BadC implements CSProcess {
      def ChannelInput inChannel
      def ChannelOutput outChannel
      def void run() {
            println "BadC: Starting"
            while (true) {
                  println "BadC: outputting"
                  outChannel.write(1)
                  println "BadC: inputting"
                  def i = inChannel.read()
                  println "BadC: looping"
            }
      }
}
```

Output generato dall'esecuzione della coppia di processi:

```
BadC: Starting
BadC: outputting
BadP: Starting
BadP: outputting
```

L'output generato su consolle evidenzia come entrambi i processi stiano cercando di eseguire un output sul rispettivo canale e come nessuno dei due possa proseguire oltre questo punto, perché nessuno tra i due processi è in grado di accettare il corrispondente input in quel momento.
Questo è evidenziato dal seguente diagramma temporale:

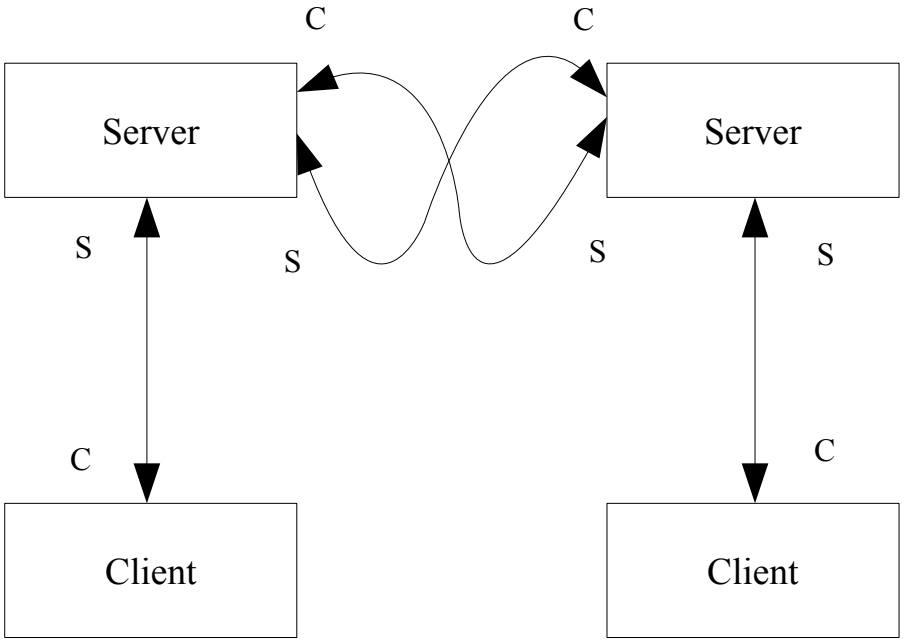


In questa situazione il problema è ovvio ed è immediatamente visibile grazie alla semplicità del caso presentato, vediamo ora un esempio più complesso.

Deadlock su server di rete

Una caratteristica comune, nelle moderne reti di calcolatori, è la possibilità di accedere a molti server dalla stessa workstation.
In background, l'amministratore di rete potrebbe impostare un sistema di backup tra i vari server allo scopo di creare un mirror dei dati contenuti in un server su un altro server.
Nelle prime incarnazioni di questo sistema sorgeva spesso un problema: c'erano dei periodi in cui le prestazioni della rete degradavano pesantemente o addirittura i server cessavano di rispondere e l'unica soluzione consisteva nel riavviarli; non era prevedibile quando questa situazione si sarebbe verificata.

Vediamo un esempio in cui siano presenti due server e due client.
I client sono in gradi di accedere ai dati presenti su entrambi i server ma, essendo collegati ad un solo server, se il dato a cui il client sta cercando di accedere non si trova sul server a cui è connesso, il server automaticamente richiederà il dato richiesto dal client all'altro server.
Allo scopo di incrementare le prestazioni dei server, ai client è permesso l'inoltro di un'altra richiesta se la richiesta precedente è stata inoltrata al secondo server. Le richieste risultano, quindi, alternate.
Le doppie frecce indicano come ci sia una prima fase di richiesta del dato e una seconda fase di risposta.



I dati contenuti nei server sono strutturati come coppie chiave_univoca - dato, ogni server contiene 10 coppie raggruppate in una mappa.

Processo client
Il processo client è collegato a due canali che utilizza per comunicare con il server a cui è connesso.
L'attributo selectList è una lista che conterrà l'elenco dei valori contenuti nei server a cui sarà necessario accedere.
In ogni iterazione del ciclo, il processo client invia una richiesta al server contenente uno dei valori chiave preso in sequenza dalla lista selectList e attende la risposta dal server.
L'attesa può durare diverso tempo perché il server potrebbe dover richiedere all'altro server il valore corrispondente alla chiave richiesta perché il valore cercato non è presente in locale.

Una volta ricevuta la risposta dal server, il client stampa a monitor il suo id e la coppia chiave-valore.

```
class Client implements CSProcess{
      def ChannelInput receiveChannel
      def ChannelOutput requestChannel
      def clientNumber
      def selectList = [ ]
      void run () {
            //numero di valori a cui è necessario accedere
            def iterations = selectList.size
            println "Client $clientNumber has $iterations values in $selectList"
            for ( i in 0 ..< iterations) {
                  def key = selectList[i]
                  requestChannel.write(key)
                  def v = receiveChannel.read()
                  println "Client $clientNumber: with $key has value $v"
            }
      }
}
```

Processo server

Il processo server dispone di tre coppie di canali, una coppia permette la comunicazione con il client e le altre due permettono di comunicare una richiesta da questo server e di riceverne il risultato mentre l'ultima coppia serve per accettare una richiesta dall'altro server e per comunicare il risultato corrispondente.
Si noti come le due coppie di canali che collegano i due server costituiscano un percorso circolare.
L'attributo dataMap contiene le coppie chiave-valore memorizzate sul server.

```
class Server implements CSProcess{
      def ChannelInput clientRequest
      def ChannelOutput clientSend
      def ChannelOutput thisServerRequest
      def ChannelInput thisServerReceive
      def ChannelInput otherServerRequest
      def ChannelOutput otherServerSend
      def dataMap = [ : ]
      void run () {
            def CLIENT = 0
            def OTHER_REQUEST = 1
            def THIS_RECEIVE = 2
            def serverAlt = new ALT ([    clientRequest,
                                          otherServerRequest,
                                          thisServerReceive])
            while (true) {
                  def index = serverAlt.select()
                  switch (index) {
                        case CLIENT :
                              def key = clientRequest.read()
                              if ( dataMap.containsKey(key) ) {
                                    clientSend.write(dataMap[key])
                              }
                              else {
                                    thisServerRequest.write(key)
                              }
                        break
                        case OTHER_REQUEST :
                              def key = otherServerRequest.read()
                              if ( dataMap.containsKey(key) ) {
```

```
                                    otherServerSend.write(dataMap[key])
                              }
                              else {
                                    otherServerSend.write(-1)
                              }
                        break
                        case THIS_RECEIVE :
                              clientSend.write(thisServerReceive.read() )
                        break
                  } //end switch
            } //end while
      } //end run
}
```

Un server può ricevere tre input: una richiesta dal client, una risposta dall'altro server o una richiesta dall'altro server, questo è evidente nella formulazione dell'alternativa e nei tre casi contenuti nello switch; quando il processo cicla la guardia opportuna viene selezionata e il codice corrispondente eseguito.

Script che costruisce la rete

La notazione x2y nella nomenclatura dei canali sta ad indicare che l'estremità in cui il dato è scritto nel canale è connessa al processo X mentre l'estremità da cui il dato è letto dal canale è connessa al processo Y.

```
def One2OneChannel S02S1request = Channel.createOne2One()
def One2OneChannel S12S0send = Channel.createOne2One()
def One2OneChannel S12S0request = Channel.createOne2One()
def One2OneChannel S02S1send = Channel.createOne2One()
def One2OneChannel C02S0request = Channel.createOne2One()
def One2OneChannel S02C0send = Channel.createOne2One()
def One2OneChannel C12S1request = Channel.createOne2One()
def One2OneChannel S12C1send = Channel.createOne2One()
def server0Map = [1:10,2:20,3:30,4:40,5:50,6:60,7:70,8:80,9:90,10:100]
def server1Map = [11:110,12:120,13:130,14:140,15:150,
                  16:160,17:170,18:180,19:190,20:200]
def client0List = [1,2,3,4,5,6,7,18,9,10]
def client1List = [11,12,13,4,15,16,17,18,19,20]

def client0 = new Client   ( requestChannel: C02S0request.out(),
                             receiveChannel: S02C0send.in(),
                             selectList: client0List,
                             clientNumber: 0)

def client1 = new Client   ( requestChannel: C12S1request.out(),
                             receiveChannel: S12C1send.in(),
                             selectList: client1List,
                             clientNumber: 1)

def server0 = new Server   ( clientRequest: C02S0request.in(),
                             clientSend: S02C0send.out(),
                             thisServerRequest: S02S1request.out(),
                             thisServerReceive: S12S0send.in(),
                             otherServerRequest: S12S0request.in(),
                             otherServerSend: S02S1send.out(),
                             dataMap: server0Map)

def server1 = new Server   ( clientRequest: C12S1request.in(),
                             clientSend: S12C1send.out(),
```

```
                                thisServerRequest: S12S0request.out(),
                                thisServerReceive: S02S1send.in(),
                                otherServerRequest: S02S1request.in(),
                                otherServerSend: S12S0send.out(),
                                dataMap: server1Map)

def network = [client0, client1, server0, server1]
new PAR (network).run()
```

L'output generato da questo script è:

```
Client 0 has 10 values in [1, 2, 3, 4, 5, 6, 7, 18, 9, 10]
Client 0: with 1 has value 10
Client 1 has 10 values in [11, 12, 13, 4, 15, 16, 17, 18, 19, 20]
Client 1: with 11 has value 110
Client 1: with 12 has value 120
Client 0: with 2 has value 20
Client 0: with 3 has value 30
Client 1: with 13 has value 130
Client 0: with 4 has value 40
Client 1: with 4 has value 40
Client 1: with 15 has value 150
Client 1: with 16 has value 160
Client 0: with 5 has value 50
Client 1: with 17 has value 170
Client 0: with 6 has value 60
Client 1: with 18 has value 180
Client 0: with 7 has value 70
Client 1: with 19 has value 190
Client 0: with 18 has value 180
Client 0: with 9 has value 90
Client 0: with 10 has value 100
Client 0 has finished
Client 1: with 20 has value 200
Client 1 has finished
```

Tutto sembra funzionare alla perfezione, almeno finché un client tenta di accedere solo ai dati contenuti nel server a cui è direttamente collegato.
Vediamo cosa accade se lo script viene leggermente modificato sostituendo gli elenchi delle richieste effettuate dai client con i due elenchi seguenti:

```
def client0List = [1,2,3,14,15,6,7,18,9,10]
def client1List = [11,12,13,4,5,16,17,8,19,20]
```

Il corrispettivo output è

```
Client 1 has 10 values in [11, 12, 13, 4, 5, 16, 17, 8, 19, 20]
Client 1: with 11 has value 110
Client 1: with 12 has value 120
Client 0 has 10 values in [1, 2, 3, 14, 15, 6, 7, 18, 9, 10]
Client 1: with 13 has value 130
Client 0: with 1 has value 10
Client 0: with 2 has value 20
Client 1: with 4 has value 40
Client 0: with 3 has value 30
```

Ops, qualcosa è andato storto, apparentemente l'ordine con cui i client eseguono le richieste ai server ha un impatto significativo sulla correttezza del procedimento; questo non dovrebbe essere

ammissibile.
Per ispezione si nota come entrambi i server entrino in uno stato in cui o entrambi tentano di inviare una richiesta all'altro server o entrambi sono in attesa di ricevere una risposta dall'altro server.
Si noti come diverse esecuzioni della stessa rete generino risultati diversi.

Come evitare il Deadlock in un'architettura client-server

Si è visto quanto sia semplice incorrere in un problema di deadlock, anche in sistemi in cui verrebbe da pensare altrimenti.
Si è resa necessaria l'introduzione di un modello di progettazione che assicuri l'immunità da condizioni di deadlock e di livelock.
Per Brinch Hansen[24] formulò, negli anni '70 un metodo di progettazione per sistemi operativi basato sull'architettura client server utilizzato con minime variazioni anche al giorno d'oggi.

Si basa su due regole fondamentali e su una metodologia per l'ispezione della rete:

1) Un processo client che invii una richiesta ad un server assicura di essere in grado di ricevere una qualunque risposta dal server immediatamente. Un'iterazione client server richiede che il client invii una richiesta al server ma non richiede che il server comunichi qualcosa al client.
2) Un processo server che abbia accettato una richiesta da un processo client deve assicurare la risposta in un tempo finito. In aggiunta, un processo server non invierà mai una risposta ad un processo client prima di ricevere una richiesta dal client stesso. Un processo server può comportarsi come un client nei confronti di un altro processo server.
3) Deadlock e livelock non si verificheranno in una tale rete di client e server se la descrizione della sequenza delle iterazioni tra processi non include uno o più percorsi circolari completi.

Risulta immediatamente chiaro come il terzo punto non sia verificato nell'esempio precedente.

Modelli per la costruzione di un client e di un server

```
class ClientTemplate implements CSProcess {
      def ChannelOutput request
      def ChannelInput response // Potrebbe non essere necessario
      void run() {
            // inizializzazione
            while (true) {
                  // creazione della richirsta al server
                  request.write ( requestObject ) // potrebbe essere un segnale
                  result = response.read() // potrebbe non essere richiesto
                  // elabora il risultato
            }
      }
}
```

Un processo client avrà un canale di output che utilizzerà per effettuare delle richieste al suo server, avrà probabilmente anche un canale di input che utilizzerà per ricevere le risposte dal server.
Il processo potrebbe eseguire qualche inizializzazione prima di entrare nel suo ciclo principale; a seconda di cosa rappresenti il processo, potrebbe essere creato un oggetto per eseguire una richiesta al server oppure potrebbe essere inviato un segnale al server, se non è necessario ricevere una risposta.
Il processo client si metterà immediatamente in attesa di una risposta dal server, se si suppone che il server ne debba inviare una, altrimenti riprenderà la propria esecuzione.

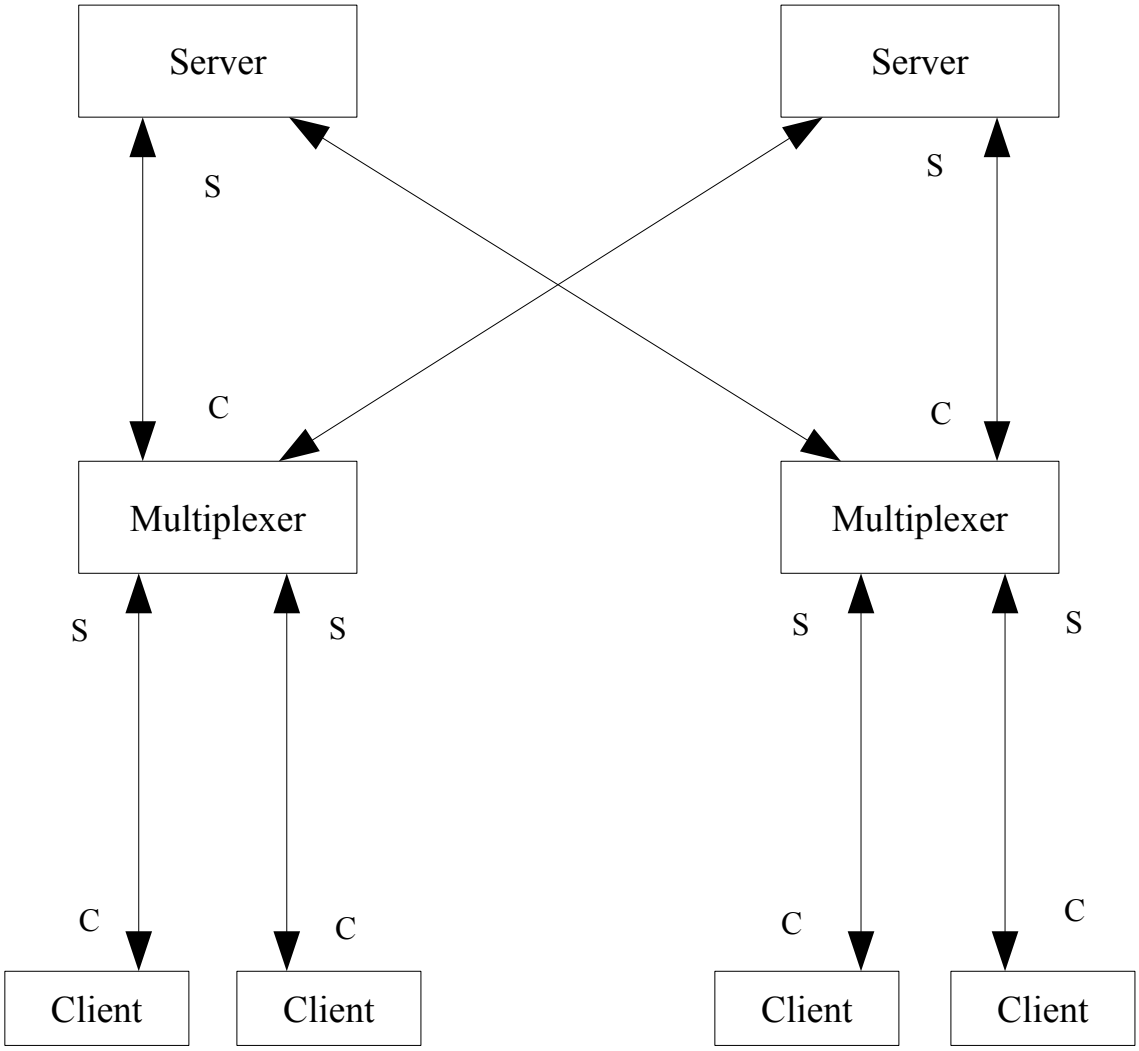
Specularmente per il processo server, il processo disporrà di un canale che utilizzerà per ricevere richieste dal client e potrebbe avere un canale utilizzato per inviare risposte al client.
All'interno del ciclo principale, il server riceve le richieste dal client, ne determina la natura se non è implicita nella ricezione della richiesta.
Il server prepara la risposta alla richiesta ricevuta, cosa che può implicare la necessità di accedere ad un altro server, e la invia al client; il server poi potrebbe dover aggiornare il suo stato prima di eseguire un altro ciclo.

```
class ServerTemplate implements CSProcess {
      def ChannelInput request
      def ChannelOutput response // Potrebbe non essere necessario
      void run() {
            // inizializzazione
            while (true) {
                  // potrebbe essere un segnale
                  def requestObject = request.read()
                  // elabora la richiesta
                  // determina il risultato, potrebbe implicare una richiesta
                  // ad un altro server
                  response.write(result) // potrebbe non essere necessario
                  // aggiorna un eventuale stato interno
            }
      }
}
```

Possiamo determinare per ispezione come questi due template soddisfino alle due condizioni di Hansen, tuttavia si analizzi più approfonditamente il comportamento del processo server.
Il processo server risponderà in tempo finito alla richiesta effettuata da un processo client se dovrà accedere solo a dati locali e se il tempo di computazione necessario alla preparazione della risposta è finito; se invece il processo server dovrà eseguire una richiesta ad un altro server, la eseguirà comportandosi come un client e, salvo problemi di comunicazione tra i server, riceverà una risposta in un tempo finito.

Il Multiplexer

Il multiplexer è un processo che accetta input da un insieme di canali ed esegue l'output su un solo canale.
Un client esegue una richiesta di un dato ad un multiplexer che si comporterà come un server, il multiplexer contiene le informazioni necessarie per determinare su quale server risiedono le informazioni richieste dal client; il multiplexer, comportandosi come un client, effettuerà la richiesta del dato al server corretto e, ricevutala, la comunicherà al client che l'ha effettuata inizialmente.
Per ispezione, la rete risulta libera da cicli e quindi non presenterà problemi di deadlock, supponendo che i processi da cui è costituita siano implementati secondo l'architettura client-server.



CSMux

Il processo CSMux è più complesso del generico multiplexer appena descritto.
Le richieste inviate dai client sono ricevute da CSMux che opererà come un server, CSMux determinerà poi in quale server sono contenuti i dati richiesti e inoltrerà al server corretto la richiesta proveniente dal client comportandosi da client a sua volta.
CSMux attenderà poi la risposta dal server che inoltrerà poi al client originario imitando un server.
Il processo CSMux utilizza attributi di tipo channelInputList e channelOutputList; si tratta di due classi che forniscono una lista dei canali di input e di output rispettivamente.

```
class CSMux implements CSProcess {
      def ChannelInputList inClientChannels
      def ChannelOutputList outClientChannels
      def ChannelInputList fromServers
      def ChannelOutputList toServers
      def serverAllocation = [ ]
      void run() {
            def servers = toServers.size() //numero di server connessi
            def muxAlt = new ALT (inClientChannels)
            while (true) {
```

```
                def index = muxAlt.select()
                def key = inClientChannels[index].read()
                def server = -1
                for ( i in 0 ..< servers) {
                      if (serverAllocation[i].contains(key)) {
                            server = i
                            break
                      }
                }
                toServers[server].write(key)
                def value = fromServers[server].read()
                outClientChannels[index].write(value)
          }
      }
}
```

inClientChannels è la lista dei canali di ingresso provenienti dai client.
fromServers è la lista dei canali di ingresso provenienti dai server.
outClientChannels e toServers sono le liste dei canali di output verso i client e i server rispettivamente.
L'attributo serverAllocation è costituito da una lista di liste contenenti gli elenchi delle chiavi corrispondenti ai dati immagazzinati in ciascun server.

Durante un'iterazione del ciclo principale di CSMux, il processo accetta una richiesta tra quelle disponibili dai client, inizializza l'attributo server ad un valore illegale e cerca quale server contiene il dato richiesto dal client.
Inoltra poi la richiesta del dato al server che lo contiene inviando la chiave corrispondente nel corretto canale della lista toServers e si mette in attesa della risposta; una volta ricevutala, la invia al client che l'ha richiesta in origine.

Non essendo presenti percorsi circolari nelle catene di operazioni generate, non si presentano problemi di deadlock.

Il processo server
Il processo utilizza due liste di canali: da e per i mux.
L'attributo dataMap contiene le coppie chiave-valore controllate dal server.
Durante un'iterazione del ciclo principale, il server seleziona un canale di input corrispondente ad un mux, ne legge la richiesta e restituisce al mux il valore corrispondente dopo averlo recuperato da dataMap.
Questo approccio rappresenta la più semplice implementazione possibile, in cui il server riceve una richiesta e fornisce immediatamente il dato richiesto.

```
class Server implements CSProcess{
      def ChannelInputList fromMux
      def ChannelOutputList toMux
      def dataMap = [ : ]
      void run() {
            def serverAlt = new ALT(fromMux)
            while (true) {
                  def index = serverAlt.select()
                  def key = fromMux[index].read()
                  toMux[index].write(dataMap[key])
            }
      }
}
```

Costruzione della rete

Convenzione usata nella nomenclatura dei canali:
C indica un client
S indica un server
M indica un multiplexer
Ad esempio, M0ToC0 indica le connessioni dal multiplexer 0 ai client connessi a quel multiplexer, M1ToS indica invece le connessioni dal multiplexer 1 ad entrambi i server.
La classe client è quella descritta nel precedente esempio sul deadlock.

```
import Client
//Richiesta del numero di client per server dall'input
def clients = Ask.Int ("Number of clients per server; 1 to 9 ? ", 1, 9)
def servers = 2
//dichiarazione dei canali
One2OneChannel[] C0ToM0 = Channel.createOne2One (clients)
One2OneChannel[] M0ToC0 = Channel.createOne2One (clients)
One2OneChannel[] C1ToM1 = Channel.createOne2One (clients)
One2OneChannel[] M1ToC1 = Channel.createOne2One (clients)
One2OneChannel[] M1ToS = Channel.createOne2One (servers)
One2OneChannel[] M0ToS = Channel.createOne2One (servers)
One2OneChannel[] S0ToM = Channel.createOne2One (servers)
One2OneChannel[] S1ToM = Channel.createOne2One (servers)

/*
*Conversione degli array di canali in istanze di channelInputList e
*channelOutputList
*/

def clientsToM0 = new ChannelInputList (C0ToM0)
def clientsToM1 = new ChannelInputList (C1ToM1)
def M0ToClients = new ChannelOutputList(M0ToC0)
def M1ToClients = new ChannelOutputList(M1ToC1)
def Mux0ToServers = new ChannelOutputList(M0ToS)
def Mux1ToServers = new ChannelOutputList(M1ToS)
def Server0ToMuxes = new ChannelOutputList (S0ToM)
def Server1ToMuxes = new ChannelOutputList (S1ToM)

def Server0FromMuxes = new ChannelInputList()
Server0FromMuxes.append(M0ToS[0].in())
Server0FromMuxes.append(M1ToS[0].in())

def Server1FromMuxes = new ChannelInputList()
Server1FromMuxes.append(M0ToS[1].in())
Server1FromMuxes.append(M1ToS[1].in())

def Mux0FromServers = new ChannelInputList ()
Mux0FromServers.append(S0ToM[0].in())
Mux0FromServers.append(S1ToM[0].in())

def Mux1FromServers = new ChannelInputList ()
Mux1FromServers.append(S0ToM[1].in())
Mux1FromServers.append(S1ToM[1].in())

/*
*Definizione delle strutture dataMap, delle liste di chiavi contenute nei due
*server e delle liste di richieste che saranno effettuate dai client.
*/
```

```
def server0Map = [1:10, 2:20, 3:30, 4:40, 5:50, 6:60, 7:70, 8:80, 9:90, 10:100]
def server1Map = [11:110,12:120,13:130,14:140,15:150,
                  16:160,17:170,18:180,19:190,20:200]
def serverKeyLists = [ [1,2,3,4,5,6,7,8,9,10],[11,12,13,14,15,16,17,18,19,20] ]
def client0List = [1,2,3,14,15,6,7,18,9,10]
def client1List = [11,12,13,4,5,16,17,8,19,20]

/*
*Costruzione della rete vera e propria:
*Il metodo collect() è utilizzato per costruire le due liste di nuove istanze di
*Client, una per ogni processo CSMux.
*Si noti come i singoli elementi degli array di canali sono inizializzati.
*/

def network = [ ]

def server0ClientList   = (0 ..< clients).collect { i ->
                          return new Client (  requestChannel: C0ToM0[i].out(),
                                               receiveChannel: M0ToC0[i].in(),
                                               clientNumber: i,
                                               selectList: client0List)
                          }

def server1ClientList   = (0 ..< clients).collect { i ->
                          return new Client (  requestChannel: C1ToM1[i].out(),
                                               receiveChannel: M1ToC1[i].in(),
                                               clientNumber: i+10,
                                               selectList: client1List)
                          }

/*
*La lista network viene poi popolata dalle necessarie istanze di CSMux e server
*e, finalmente, il tutto viene avviato invocando un PAR.
*/

network << new CSMux (  inClientChannels: clientsToM0,
                        outClientChannels: M0ToClients,
                        fromServers: Mux0FromServers,
                        toServers: Mux0ToServers,
                        serverAllocation: serverKeyLists)

network << new CSMux (  inClientChannels: clientsToM1,
                        outClientChannels: M1ToClients,
                        fromServers: Mux1FromServers,
                        toServers: Mux1ToServers,
                        serverAllocation: serverKeyLists)

network << new Server ( fromMux: Server0FromMuxes,
                        toMux: Server0ToMuxes,
                        dataMap: server0Map)

network << new Server ( fromMux: Server1FromMuxes,
                        toMux: Server1ToMuxes,
                        dataMap: server1Map)

new PAR(network + server0ClientList + server1ClientList).run()
```

Si noti, durante l'esecuzione della rete, come la sequenza di richieste effettuate dai due gruppi di client siano quelle che hanno provocato la situazione di stallo nell'esempio precedente e come questo suggerisca la correttezza della soluzione riportata.

6) Actors - active objects

L'implementazione del concetto di attore in GPars è stata inizialmente ispirata dalla libreria Actors di Scala; il modello è stato successivamente sviluppato ed ora GPars ne offre un'implementazione molto più completa di quella offerta da Scala come standard.
IL modello si basa sul concetto che ogni oggetto è un attore, cioè un oggetto che ha una "casella postale" e un "comportamento"; gli attori possono comunicare tra loro solo scambiandosi messaggi, la casella postale si comporta come un buffer contenente i messaggi ricevuti.
Quando un attore riceve un messaggio, il suo "comportamento" viene eseguito: l'attore può inviare un certo numero di messaggi ad altri attori, creare degli altri attori e assumere un nuovo comportamento da eseguire quando sarà ricevuto il prossimo messaggio.
Tutti gli attori sono eseguiti concorrentemente: un attore può essere visto come un piccolo processo indipendente.

Il modello di attore presenta tre importanti proprietà:

- Tutte le comunicazioni sono asincrone: questo implica che un attore non attende la ricezione del messaggio che ha inviato ma continua immediatamente la propria esecuzione; non c'è nessuna garanzia sull'ordine in cui il destinatario riceverà i messaggi, si sa solo che verranno eventualmente ricevuti.
- Tutte le comunicazioni avvengono tramite messaggi: non esistono stati condivisi tra attori, se un attore ha la necessità di conoscere lo stato interno di un altro attore dovrà inviargli una richiesta; questo permette agli attori di avere un controllo assoluto sul loro stato evitando problemi come il lost update. Anche la manipolazione dello stato di un attore avviene tramite messaggi.
- In ogni momento, un attore è in esecuzione su al più un thread, l'attore in sé è identificato da un task.
  Ogni qualvolta un thread venga assegnato ad un attore la corrispondente area di memoria viene sincronizzata in automatico, quindi lo stato dell'attore può essere modificato liberamente dal thread che lo sta eseguendo senza doversi preoccupare di ulteriori problemi di sincronia o di locking di risorse.

Idealmente, il codice di un attore non dovrebbe mai essere poter essere invocato dall'esterno, affinché tutto il codice contenuto nella classe che costituisce l'attore possa essere eseguito solo dal thread che sta gestendo l'ultimo messaggio ricevuto.
Se questo avviene, il codice e lo stato dell'attore sono implicitamente thread-safe; se invece almeno un metodo dell'attore può essere invocato direttamente dall'esterno, questa garanzia decade.

Gli active object sono stati costruiti come un wrap up degli attori per facilitare i passaggi mentali necessari all'uso degli attori stessi; apparentemente gli utenti hanno trovato poco comprensibile il concetto di "passare messaggi" tipico degli attori; generalmente, il concetto di "chiamata di metodi" risulta più semplice.

Paragone tra CSP e modello basato sugli attori

Sotto un certo punto di vista, i modelli basati sugli attori e CSP sono molto simili: in entrambi i casi si hanno dei processi in esecuzione concorrente che si scambiano dei messaggi.
I due modelli, tuttavia, raggiungono questo risultato seguendo due approcci sostanzialmente differenti:

- In CSP i processi sono anonimi mentre gli attori possiedono un'identità

- In CSP la comunicazione richiede fondamentalmente un rendezvous tra il processo che vuole inviare un messaggio e il processo che lo vuole ricevere. Nel modello basato sugli attori, invece, le comunicazioni sono fondamentalmente asincrone, un messaggio può essere inviato prima che il ricevente sia pronto ad accettarlo.
  Questi due approcci possono essere considerati duali, nel senso che un sistema basato su rendezvous può essere utilizzato per costruire un sistema di comunicazione bufferizzato che si comporta come un sistema basato sullo scambio asincrono di messaggi, allo stesso modo si può utilizzare un sistema basato sullo scambio di messaggi per costruire un protocollo di handshaking che sincronizzi lo scambio di messaggi tra trasmettitore e ricevitore.

- CSP utilizza dei canali specificamente creati per eseguire le comunicazioni mentre un sistema basato su attori indica esplicitamente il destinatario di ogni messaggio.
  I due approcci possono nuovamente essere considerati duali: un processo che possa ricevere dei messaggi esclusivamente attraverso un singolo canale può essere identificato con quel canale mentre l'accoppiamento nominativo utilizzato dagli attori può essere convertito in un accoppiamento basato su canali con l'introduzione di un attore intermedio che agisca come un canale.

Abbiamo visto come i modelli basati su CSP e su attori possano essere considerati simili perché è possibile ritenere, in qualche modo, uno il duale dell'altro.
Tuttavia bisogna tenere conto di un piccolo dettaglio, in qualche modo derivante dalla teoria dei grafi: è sensato equiparare un sistema concorrente ad un grafo connesso[41] o come un insieme di grafi connessi in cui i processi siano identificati con i nodi e i canali con i vertici; come sappiamo, in un grafo connesso esistono almeno tanti vertici quanti sono i nodi[42] ma generalmente esistono molti più vertici che nodi.
La realizzazione di un software usando il modello basato su CSP comporta la specificazione di ogni singolo vertice di un grafo connesso.
Questo porta ad un'ovvia considerazione: per la realizzazione di un software che non necessiti di una specificazione formale (perché per esempio non si tratta di un software per un sistema critico), l'applicazione del modello basato su CSP comporta una complicazione non strettamente indispensabile e che quindi può essere rimossa in favore di un modello che permetta di produrre software in modo più sintetico.
Un altro vantaggio del modello basato sugli attori è che, essendo le comunicazioni asincrone, un'eventuale congestione presente nel mezzo trasmissivo (ad esempio una infrastruttura di rete) non costituisce necessariamente un collo di bottiglia per quanto riguarda le prestazioni del sistema.

Tipi di attori

Esplorando gli approcci teorici utilizzati per la costruzione del modello di attore e le sue varie implementazioni, si incontrano due tipi principali di attore: uno dotato di stato e un altro stateless.

In GPars sono state implementate entrambe le varianti:

- Le classi DynamicDispatchActor e ReactiveActor sono implementazioni stateless, non conservano cioè traccia dei messaggi arrivati in precedenza; i messaggi ricevuti verranno elaborati tutti allo stesso modo.

41 Perché se così non fosso non avrebbe senso parlare di problemi di condivisione di memoria, condivisione di risorse, sincronia etc.

42 Meno uno

- La classe DefaultActor[43] implementa la versione “con stato implicito” del modello di attore e permette all'utente di gestire direttamente lo stato dell'attore.
  Dopo aver ricevuto un messaggio l'attore si porta in un nuovo stato, questo permetterà all'attore di gestire diversamente i messaggi successivi.
  Come esempio, si pensi ad un attore che implementi un sistema di decodifica: all'inizio l'attore potrebbe accettare solo un tipo di messaggio, verosimilmente la chiave di decodifica e, una volta ricevutala, si porterebbe in un altro stato in cui potrebbe accettare sia una nuova chiave di decodifica sia i messaggi cifrati da decodificare.
  La versione stateful del modello di attore permette di imprimere questo tipo di dipendenze direttamente nella struttura del codice; tuttavia questo comporta un piccolo costo in prestazioni dovuto all'assenza del supporto a al concetto di continuation[44] da parte della JVM.

Un attore può gestire una coda di messaggi in due modi:
- equamente: l'attore restituisce il thread che lo sta eseguendo al thread pool dopo aver completato la gestione di un messaggio ricevuto..
- non equamente: è il comportamento di default di DefaultActor, forza l'esecuzione continua dell'attore finché non vengano smaltiti tutti i messaggi presenti nella mailbox.

Tipicamente le prestazioni di un attore non equo sono molto migliori di quelle ottenute da un attore equo.

6.1) Utilizzo degli attori

Idealmente, un attore può eseguire solo tre tipi di operazioni: può ricevere un messaggio, può mandare un messaggio oppure può creare un altro attore.
Per quanto non imposto da GPars, un messaggio dovrebbe essere immutabile o, per lo meno, l'attore mittente dovrebbe astenersi dal modificare un messaggio dopo averlo inviato[45].

Invio e ricezione di messaggi

Invio di messaggi
Un messaggio può essere inviato utilizzando il metodo send(), l'operatore “<<” oppure chiamando implicitamente il metodo call():

```
def passiveActor = Actors.actor{
    loop {
        react { msg -> println "Received: $msg"; }
    }
}
passiveActor.send 'Message 1'
passiveActor << 'Message 2'    //utilizzo dell'operatore <<
passiveActor 'Message 3'       //utilizzo implicito del metodo call()
```

43 Un'altra implementazione della versione stateful era rappresentata dalla classe AbstractPooledActor.
44 Con continuation o continuations ci si riferisce ad una tecnica di programmazione funzionale che permette di salvare lo stato di esecuzione corrente di un thread, introducendo la possibilità di sospenderne l'esecuzione per riprenderla in un secondo tempo.
Si tratta di una tecnica utilizzata, ad esempio, per l'implementazione di user interfaces perché permette di effettuare una richiesta e di non bloccare un thread mentre si attende la risposta.
Ulteriori risorse possono essere trovate in [25] e in [26].
45 Tipicamente il problema si risolve inviando tramite messaggio una copia dell'oggetto che si voleva inviare.

Gli attori comunicano scambiandosi messaggi in modo asincrono, tuttavia un attore può sospendersi attendendo la risposta ad un messaggio inviato utilizzando il metodo sendAndWait() e le relative varianti; il valore restituito da sendAndWait() rappresenta la risposta ricevuta.

```
def replyingActor = Actors.actor{
    loop {
        react { msg ->
            println "Received: $msg";
            reply "I've got $msg"
        }
    }
}
def reply1 = replyingActor.sendAndWait('Message 4')
def reply2 = replyingActor.sendAndWait('Message 5', 10, TimeUnit.SECONDS)
use (TimeCategory) {
    def reply3 = replyingActor.sendAndWait('Message 6', 10.seconds)
}
```

È possibile specificare un timeout durante l'invocazione di sendAndWait().

Il metodo sendAndContinue() è complementare a sendAndWait(), permette all'attore di continuare la propria esecuzione mentre la chiusura fornita al metodo sendAndContinue() attende la risposta al messaggio inviato.

```
friend.sendAndContinue 'I need money!', {money -> pocket money}
println 'I can continue while my friend is collecting money for me'
```

Il metodo send() e le relative varianti lanceranno un'eccezione se si sta cercando di comunicare con un attore inesistente.

Ricezione di messaggi
Non esiste una corrispondenza biunivoca tra attori e thread di sistema, questo permette ad un elevato numero di attori di condividere un thread pool di dimensioni ridotte.
Questa architettura permette di scansare alcune limitazioni relative ai thread imposte dalla JVM: tipicamente la JVM è in grado di fornire un numero limitato di thread (alcune migliaia), tuttavia il numero di attori è limitato solamente dallo spazio disponibile in memoria dato che un attore normalmente non consuma thread mentre è sospeso.

Il corpo di un attore viene eseguito in blocchi, separati da periodi di quiete in cui l'attore attende la ricezione di un messaggio.
Il modello naturale per questo comportamento si baserebbe sull'uso di “continuations” (che non sono supportate dalla JVM); essendo impossibilitati ad utilizzare questo modello, si è costretti a simularlo nel framework degli attori e questo ha un impatto (limitato) sulla struttura del codice.

Ricezione non bloccante
Il punto di forza di un attore è la capacità di sospendersi attendendo un messaggio senza avere la necessità di mantenere il possesso di un thread di sistema.
Il metodo react(), a cui è possibile fornire un parametro timeout, consuma il messaggio più vecchio presente nell'inbox dell'attore mettendosi in attesa se non sono presenti messaggi.

```
println 'Waiting for a gift'
react {gift ->
    if (myWife.likes gift) reply 'Thank you!'
}
```

La chiusura fornita al metodo react() non è eseguita direttamente, la sua esecuzione viene programmata dal thread corrente per quando sarà disponibile un messaggio; fatto ciò il thread che sta eseguendo l'attore si distacca e si rende disponibile per l'esecuzione di un altro attore.
Per permettere il distacco di un attore da un thread, il metodo react() richiede che il codice sia scritto secondo uno speciale continuation-style.

```
Actors.actor {
    loop {
        println 'Waiting for a gift'
        react {gift ->
            if (myWife.likes gift) reply 'Thank you!'
            else {
                reply 'Try again, please'
                react {anotherGift ->
                    if (myChildren.like gift) reply 'Thank you!'
                }
                println 'Never reached'
            }
        }
        println 'Never reached'
    }
    println 'Never reached'
}
```

Il metodo react() implementa una semantica particolare che permette il distacco del thread che sta eseguendo l'attore allorquando non siano disponibili messaggi nell'inbox: essenzialmente react() schedula l'esecuzione della chiusura su cui è chiamato in modo tale che venga eseguita dopo la ricezione del prossimo messaggio e ritorna.
La chiusura fornita al metodo react() costituisce il punto in cui l'esecuzione dovrebbe continuare, da cui il continuation-style.

Per garantire che al più un thread sia attivo su un attore, i messaggi devono essere gestiti sequenzialmente, non è possibile elaborare il messaggio successivo prima del termine della gestione del messaggio corrente.
Il corpo di un attore dovrebbe essere interamente contenuto nella chiamata a react(): non dovrebbe essere necessario scrivere del codice all'esterno della chiamata a react(); questo è imposto in alcune implementazioni del modello di attore, in GPars questa imposizione non è presente per motivi prestazionali.
Il metodo loop() permette di iterare l'esecuzione del corpo dell'attore; a differenza di altri costrutti (come cicli while, for etc.) coopera con il metodo react() in modo da garantire l'iterazione mano a mano che i messaggi vengono ricevuti.

Invio di risposte
Quando un messaggio viene ricevuto, l'attributo “mittente” è disponibile all'interno della chiusura che si occupa della sua gestione

```
react {tweet ->
    if (isSpam(tweet)) ignoreTweetsFrom sender
    sender.send 'Never write me again!'
}
```

I metodi reply / replyIfExist sono disponibili nella classe actor e nella classe AbstractPooledActor (quindi non sono disponibili in DefaultActor, DynamicDispatchActor o ReactiveActor) e nei messaggi stessi, al momento della loro ricezione.

Sono utilizzati per:

1) rispondere ad un messaggio specifico
2) rispondere all'ultimo messaggio inviato da ogni mittente (se si ricevono messaggi da mittenti diversi)[46]

Forwarding

Al momento dell'invio, è possibile specificare come mittente un attore diverso da quello che ha effettivamente inviato il messaggio allo scopo di reindirizzare eventuali risposte ad un attore diverso da quello originario

```
def decryptor = Actors.actor {
    react {message ->
        reply message.reverse()
//        sender.send message.reverse()        //Esistono due modi di inviare
                                               //una risposta
    }
}
def console = Actors.actor {        //Questo attore stampa i messaggi decifrati
                                    //che gli vengono inoltrati
    react {
        println 'Decrypted message: ' + it
    }
}
decryptor.send 'lellarap si yvoorG', console   //Specifica a chi inviare le
                                               //risposte
console.join()
```

Messaggi non processati

A volte un attore non può elaborare correttamente un messaggio, per esempio a causa della terminazione prematura dell'attore stesso; quando il caso deve essere gestito, si invoca il metodo onDeliveryError() su tutti i messaggi presenti nell'inbox al momento della terminazione dell'attore.
Il metodo o la chiusura onDeliveryError() può essere specificato nel messaggio e può, ad esempio, notificare il mittente del messaggio della mancata ricezione.

```
final DefaultActor me
me = Actors.actor {
    def message = 1
    message.metaClass.onDeliveryError = {->
        //Invia una notifica al mittente
        me << "Could not deliver $delegate"
    }
    def actor = Actors.actor {
        react {
            //attende per due secondi in modo da ricevere entrambi i imessaggi
            Thread.sleep(2000)
            //termina l'attore dopo la gestione del primo messaggio
            stop()
        }
    }
    actor << message
    actor << message
    react {
        //stampa la notifica
        println it
    }
}
```

46 Un esempio può essere trovato in [27].

```
me.join()
```

Alternativamente il metodo onDeliveryError() può essere specificato nel mittente e può, in ogni caso, essere aggiunto ad entrambi dinamicamente.

```
final DefaultActor me
me = Actors.actor {
    def message1 = 1
    def message2 = 2
    def actor = Actors.actor {
        react {
            //attende per due secondi in modo da ricevere entrambi i imessaggi
            Thread.sleep(2000)
            //termina l'attore dopo la gestione del primo messaggio
            stop()
        }
    }
    me.metaClass.onDeliveryError = {msg ->
        //Notifica
        println "Could not deliver message $msg"
    }
    actor << message1
    actor << message2
    actor.join()
}
me.join()
```

Oppure può essere aggiunto staticamente nella definizione dell'attore.

```
class MyActor extends DefaultActor {
    public void onDeliveryError(msg) {
        println "Could not deliver message $msg"
    }
    …
}
```

Creazione di un attore

Un gruppo di attori condivide un thread pool, i thread vengono dinamicamente assegnati agli attori qualora debbano reagire a dei messaggi ricevuti; un attore restituisce il thread che lo sta gestendo al thread pool quando si mette in attesa di un nuovo messaggio (perché ha vuotato la coda se è un attore non equo o perché ha processato un messaggio se è un attore equo).

Esempio: attore che stampa tutti i messaggi ricevuti.

```
def console = Actors.actor {
    loop {
        react {
            println it
        }
    }
}
```

Si noti il metodo loop(): assicura che l'attore non termini dopo la ricezione del primo messaggio.

Esempio: decryptor service, invia i messaggi decifrati ai rispettivi mittenti.

```
final def decryptor = Actors.actor {
```

```
    loop {
        react {String message ->
            if ('stopService' == message) {
                println 'Stopping decryptor'
                stop()
            }
            else reply message.reverse()
        }
    }
}
Actors.actor {
    decryptor.send 'lellarap si yvoorG'
    react {
        println 'Decrypted message: ' + it
        decryptor.send 'stopService'
    }
}.join()
```

Joining di attori

Gli attori mettono a disposizione il metodo join(): permette a chi lo invoca di attendere la terminazione degli attori stessi (o del gruppo di attori creato); è disponibile una variante in cui viene specificato un timeout.

L'operatore *. si dimostra estremamente utile per eseguire il join di una lista di attori.

```
def master = new GameMaster().start()
def player = new Player(name: 'Player', server: master).start()
[master, player]*.join()
```

Loop condizionati o dotati di contatore

Nel metodo loop(), che assicura la non terminazione dell'attore dopo la ricezione del primo messaggio, è possibile specificare:

- una condizione
- un contatore
- del codice da eseguire quando il metodo termina, fornito sotto forma di chiusura[47].

Esempio: attore che riceve tre messaggi e poi stampa il massimo tra i valori ricevuti

```
final Actor actor = Actors.actor {
    def candidates = []
    def printResult = {-> println "The best offer is ${candidates.max()}"}
    loop(3, printResult) {
        react {
            candidates << it
        }
    }
}
actor 10
actor 30
actor 20
actor.join()
```

Esempio: attore che riceve messaggi finché non viene ricevuto un valore maggiore di 30.

```
final Actor actor = Actors.actor {
    def candidates = []
    final Closure printResult = {-> println "Reached best offer - $
{candidates.max()}"}
```

47 Noto come After Loop Termination Code Handler.

```
    loop({-> candidates.max() < 30}, printResult) {
        react {
            candidates << it
        }
    }
}
actor 10
actor 20
actor 25
actor 31
actor 20
actor.join()
```

L'eventuale chiusura eseguita al termine del ciclo può utilizzare il metodo react{} ma non può inizializzare un altro ciclo usando il metodo loop().

Schedulazione personalizzata
Gli attori utilizzano di default la standard concurrency library del JDK; è possibile fornire un thread scheduler personalizzato fornendo l'opportuno costruttore al momento della creazione del parallel group (classe PGroup).

6.2) Concetti fondamentali

Un gruppo di attori condivide un thread pool, i thread vengono dinamicamente assegnati agli attori qualora debbano reagire a dei messaggi ricevuti; un attore restituisce il thread che lo sta gestendo al thread pool quando si mette in attesa di un nuovo messaggio.
Non esiste una corrispondenza biunivoca tra thread e attori: trattandosi di entità separabili è possibile gestire un elevato numero di attori con un thread pool relativamente piccolo[48].
Uno dei vantaggi del modello di attore è la scalabilità[49] potenzialmente illimitata, ottenuta grazie al distacco dai thread fisici.

Esempio: attore che stampa tutti i messaggi ricevuti.

```
def console = Actors.actor {
    loop {
        react {
            println it
        }
    }
}
```

Alternativamente è possibile estendere la classe DefaultActor e sovrascrivere il metodo act().
Se si volesse creare a mano un attore bisognerebbe anche lanciarne l'esecuzione in modo da permettergli di collegarsi al thread pool per accettare i messaggi; il factory method[50] actor() si prende cura di questo.

```
class CustomActor extends DefaultActor {
    @Override
    protected void act() {
```

48 Contenente anche un solo thread.
49 Riferita al numero di attori utilizzabili.
50 Factory method: si tratta di una metodologia per la creazione di oggetti che permette, tra l'altro, di integrare le funzionalità offerte dal costruttore di una classe.
Maggiori informazioni sono disponibili in [28] e in [29].

```
        loop {
            react {
                println it
            }
        }
    }
}
def console=new CustomActor()
console.start()
```

Creazione di un servizio asincrono

```
import static groovyx.gpars.actor.Actors.*
final def decryptor = actor {
    loop {
        react {String message->
            reply message.reverse()
        }
    }
}
def console = actor {
    decryptor.send 'lellarap si yvoorG'
    react {
        println 'Decrypted message: ' + it
    }
}
console.join()
```

Come si è già visto, è possibile creare un nuovo attore utilizzando il factory method actor() a cui viene passato come parametro una chiusura contenente il codice costituente il corpo dell'attore.
All'interno dell'attore è possibile utilizzare i metodi loop() per iterare, react() per ricevere un messaggio e reply() per inviare una risposta al mittente del messaggio che si sta processando.
Quando il decifratore non trova un messaggio in coda al momento della chiamata a react(), il metodo react() restituisce il thread in uso al thread pool rendendolo disponibile agli altri attori in attesa. Solamente quando un nuovo messaggio arriva nell'inbox dell'attore viene schedulata, da parte del thread pool, l'esecuzione della chiusura contenuta nel metodo react().

Gli attori basati su eventi simulano il concetto di continuation internamente: il lavoro dell'attore è suddiviso in blocchi eseguiti sequenzialmente, mano a mano che arrivano messaggi nell'inbox.
Ogni blocco di esecuzione di un attore può essere eseguito da un diverso thread appartenente allo stesso pool.

Groovy presenta una certa flessibilità nella gestione delle chiusure e questo permette di definire gli attori in diversi modi.

Esempio: attore che attende una risposta da un altro attore per 30 secondi. Si noti come sia possibile utilizzare il DSL definito dalla classe org.codehaus.groovy.runtime.TimeCategory per la specifica del timeout all'interno del metodo react(), premesso che l'utente inserisca il blocco all'interno di un blocco TimeCategory.

```
def friend = Actors.actor {
    react {
            //questo attore non replica → il mittente non riceverà alcuna
            //risposta in tempo utile
        println it
```

```
        react {
            println it
        }
    }
}
def me = Actors.actor {
    friend.send('Hi')
    //Attesa di 30 secondi per la risposta
    react(30000) {msg ->
        if (msg == Actor.TIMEOUT) {
            friend.send('I see, busy as usual. Never mind.')
            stop()
        } else {
            //Continua la conversazione
            println "Thank you for $msg"
        }
    }
}
me.join()
```

Se mentre si è in attesa di un messaggio scatta un timeout, viene ricevuto il messaggio Actor.TIMEOUT e viene lanciato l'handler onTimeout(), se presente nell'attore.

```
def friend = Actors.actor {
    react {
            //questo attore non replica → il mittente non riceverà alcuna
            //risposta in tempo utile
        println it
        react {
            println it
        }
    }
}
def me = Actors.actor {
    friend.send('Hi')
    delegate.metaClass.onTimeout = {->
        friend.send('I see, busy as usual. Never mind.')
        stop()
    }
    //Attesa di 1 secondo per la risposta
    react(1000) {msg ->
        if (msg != Actor.TIMEOUT) {
            //Continua la conversazione
            println "Thank you for $msg"
        }
    }
}
me.join()
```

Gli attori garantiscono la thread-safety di codice non thread-safe

Nel modello di attore è garantito che al più un thread stia eseguendo uno specifico attore in ogni momento nel tempo e, in modo trasparente per l'utente, che le aree di memoria utilizzate da un attore vengano sincronizzate ogni qualvolta gli venga assegnato un thread affinché lo stato dell'attore possa essere modificato in modo sicuro dal codice, senza avere la necessità di ricorrere a lock o altri meccanismi simili.

```
class MyCounterActor extends DefaultActor {
    private Integer counter = 0
    protected void act() {
```

```
        loop {
            react {
                counter++
            }
        }
    }
}
```

In linea di principio, il codice di un attore non dovrebbe mai essere chiamato dall'esterno al fine di assicurarne l'esecuzione solo da parte del thread che sta seguendo l'attore nella gestione dell'ultimo messaggio ricevuto acciocché il codice e lo stato dell'attore siano implicitamente thread-safe.
Se viene permessa l'esecuzione di codice dell'attore dall'esterno, questa garanzia decade.

Esempio: un semplice sommatore
Un esempio un po' più realistico di un attore basato su eventi è costituito da un attore che riceve due valori, li somma e invia il risultato ad un attore consolle.

```
import groovyx.gpars.group.DefaultPGroup
//Non necessario, dimostra come un thread pool contenente un singolo thread
//possa gestire più attori
def group = new DefaultPGroup(1);
final def console = group.actor {
    loop {
        react {
            println 'Result: ' + it
        }
    }
}
final def calculator = group.actor {
    react {a ->
        react {b ->
            console.send(a + b)
        }
    }
}
calculator.send 2
calculator.send 3
calculator.join()
group.shutdown()
```

Si noti come gli attori orientati agli eventi richiedano una cura particolare per quello che riguarda il metodo react(): dato che questo tipo di attori necessitano di dividere l'esecuzione del codice in blocchi assegnabili a thread diversi e che le continuations non sono supportate dalla JVM, i blocchi devono essere creati artificialmente.
Il metodo react() crea l'handler del prossimo messaggio: appena l'handler del messaggio corrente termina, il prossimo handler (continuation) viene schedulato.

Esempio: Merge Sort concorrente
Esegue il merge sort di una lista di interi utilizzando gli attori.

```
import groovyx.gpars.group.DefaultPGroup
import static groovyx.gpars.actor.Actors.actor
Closure createMessageHandler(def parentActor) {
    return {
        react {List<Integer> message ->
            assert message != null
            switch (message.size()) {
                case 0..1:
                    parentActor.send(message)
                    break
                case 2:
                    if (message[0] <= message[1]) parentActor.send(message)
                    else parentActor.send(message[-1..0])
                    break
                default:
                    def splitList = split(message)
                    def child1 = actor(createMessageHandler(delegate))
                    def child2 = actor(createMessageHandler(delegate))
                    child1.send(splitList[0])
                    child2.send(splitList[1])
                    react {message1 ->
                        react {message2 ->
                            parentActor.send merge(message1, message2)
                        }
                    }
            }
        }
    }
}
def console = new DefaultPGroup(1).actor {
    react {
        println "Sorted array:t${it}"
        System.exit 0
    }
}
def sorter = actor(createMessageHandler(console))
sorter.send([1, 5, 2, 4, 3, 8, 6, 7, 3, 9, 5, 3])
console.join()
def split(List<Integer> list) {
    int listSize = list.size()
    int middleIndex = listSize / 2
    def list1 = list[0..<middleIndex]
    def list2 = list[middleIndex..listSize - 1]
    return [list1, list2]
}
List<Integer> merge(List<Integer> a, List<Integer> b) {
    int i = 0, j = 0
    final int newSize = a.size() + b.size()
    List<Integer> result = new ArrayList<Integer>(newSize)
    while ((i < a.size()) && (j < b.size())) {
        if (a[i] <= b[j]) result << a[i++]
        else result << b[j++]
    }
    if (i < a.size()) result.addAll(a[i..-1])
    else result.addAll(b[j..-1])
    return result
}
```

Metodi per la gestione del ciclo vitale di un attore

In ogni attore possono essere definiti dei metodi che permettono di osservarne il ciclo vitale e che vengono invocati automaticamente al verificarsi di determinati eventi.

- afterStart() - invocato quando l'attore viene avviato.
- afterStop(List undeliveredMessages) – invocato immediatamente dopo la terminazione dell'attore, riceve tutti i messaggi non ancora gestiti presenti in coda come parametro.
- onInterrupt(InterruptedException e) – invocato quando il thread che sta eseguendo l'attore viene interrotto. L'interruzione del thread causa in ogni caso l'interruzione dell'attore.
- onTimeout() - viene chiamato quando l'attore non ha ricevuto un messaggio prima dello scadere del timeout specificato nel metodo react() che lo sta aspettando.
- onException(Throwable e) – invocato quando viene generata un'eccezione nell'handler degli eventi dell'attore (in pratica si tratta della gestione del messaggio corrente); l'attore termina quando questo metodo ritorna.

Questi metodi possono essere definiti staticamente nella classe attore che si sta utilizzando o possono essere aggiunti dinamicamente tramite metaclasse.

```
class MyActor extends DefaultActor {
    public void afterStart() {
        …
    }
    public void onTimeout() {
        …
    }
    protected void act() {
       …
    }
}

def myActor = actor {
    delegate.metaClass.onException = {
        log.error('Exception occurred', it)
    }
…
}
```

Gestione del thread pool

Gli attori vengono organizzati in gruppi, di default è sempre presente un gruppo a livello di applicazione in cui vengono creati gli attori utilizzando il factory method Actors.actor[51].
Così come si utilizza il metodo Actors.actor per creare degli attori nel gruppo di default, è possibile specificare degli altri gruppi in cui è possibile creare attori.

```
def myGroup = new DefaultPGroup()
def actor1 = myGroup.actor {
…
}
def actor2 = myGroup.actor {
…
}
```

Gli attori appartenenti allo stesso gruppo condividono il corrispondente thread pool; il thread pool è

51 Actors si riferisce al default group.

costituito di default da n+1 threads, dove n è il numero di CPU individuate dalla JVM.
Il numero di thread nel pool può essere specificato in due modi: a livello globale impostando la proprietà di sistema gpars.poolsize o individualmente per ogni singolo gruppo specificando il numero di thread desiderato al momento della creazione del gruppo.

```
def myGroup = new DefaultPGroup(10)  //Il pool conterrà 10 thread
```

La dimensione del thread pool può essere manipolata tramite la classe DefaultPGroup che a sua volta delegerà il compito all'interfaccia del pool.
I metodi che permettono la manipolazione della dimensione del pool sono:

- resize() come suggerisce il nome, permette di variare la dimensione del pool
- resetDefaultSize() imposta la dimensione del pool al valore di default
- shutdown() utilizzato per completare in modo sicuro tutti i task, terminare i thread e distruggere il pool permettendo di uscire dalla JVM in modo organizzato.

```
… (n+1 threads nel pool dopo lo startup)
Actors.defaultActorPGroup.resize 1  //usa un pool con un solo thread
… (1 thread nel pool)
Actors.defaultActorPGroup.resetDefaultSize()
… (n+1 threads nel pool)
Actors.defaultActorPGroup.shutdown()
```

Il thread pool creato utilizzando la classe DefaultPGroup è costituito da demoni; se si richiede l'uso di thread che non siano dei demoni, è possibile creare un thread pool utilizzando la classe NonDaemonPGroup.

```
def nonDaemonGroup = new NonDaemonPGroup()
def actor2 = nonDaemonGroup.actor {
…
}
class MyActor {
    def MyActor() {
        this.parallelGroup = nonDaemonGroup
    }
    void act() {...}
}
```

Attori appartenenti allo stesso gruppo condividono lo stesso thread pool, questo fa sì che sia possibile variare le prestazioni di vari componenti di un'applicazione utilizzando pool di dimensioni diverse.

```
def coreActors = new NonDaemonPGroup(5)  //5 non-daemon threads pool
def helperActors = new DefaultPGroup(1)  //1 daemon thread pool
def priceCalculator = coreActors.actor {
…
}
def paymentProcessor = coreActors.actor {
…
}
def emailNotifier = helperActors.actor {
…
}
def cleanupActor = helperActors.actor {
…
}
//incrementa la dimensione del core pool
```

```
coreActors.resize 6
//arresta il thread pool del gruppo per liberare risorse di sistema
helperActors.shutdown()
```

Attori Bloccanti

Nel caso in cui in un'applicazione si preveda di avere uno o più attori la cui coda di messaggi si svuoti raramente si potrebbe optare per l'utilizzo di attori bloccanti in alternativa all'utilizzo di attori basati su eventi.
Un attore bloccante acquisisce il controllo di un thread (dal thread pool in uso) quando viene creato e non rilascerà mai quel thread durante il suo intero ciclo vitale, incluso il tempo in cui è in attesa di un messaggio.

Gli attori bloccanti fanno risparmiare parte dell'overhead dovuto alla gestione dei thread, dato che non devono mai competere per l'acquisizione di un thread dopo essere essere stati inizializzati, e permettono di scrivere un codice dalla struttura più lineare, perché cade la necessità di simulare le continuations e perché si limitano a bloccare il thread in uso utilizzando il metodo receive() quando sono in attesa di un messaggio.
Ovviamente il numero di attori bloccanti in esecuzione contemporanea è limitato dalla dimensione del thread pool tuttavia, se la coda dell'attore bloccante si svuota raramente, questo tipo di attori fornirà prestazioni migliori rispetto a quelle fornite da un attore non bloccante basato su eventi.

```
def decryptor = blockingActor {
    while (true) {
        receive {message ->
            if (message instanceof String) reply message.reverse()
            else stop()
        }
    }
}
def console = blockingActor {
    decryptor.send 'lellarap si yvoorG'
    println 'Decrypted message: ' + receive()
    decryptor.send false
}
[decryptor, console]*.join()
```

Gli attori bloccanti forniscono un ulteriore strumento per l'ottimizzazione delle prestazioni un una rete di attori; un attore bloccante potrebbe essere un buon candidato per una posizione ad elevato traffico.

6.3) Stateless actors

Dynamic Dispatch Actor

La classe DynamicDispatchActor fornisce una struttura alternativa per la gestione dei messaggi ricevuti da un attore.
Un attore costruito con DynamicDispatchActor controlla ciclicamente la propria mailbox e, quando riceve un messaggio, lo invia ad uno dei metodi onMessage(message) definiti nell'attore; la struttura di un attore di tipo DynamicDispatchActor ricorda molto quella di uno switch.
DynamicDispatchActor sfrutta il dynamic method dispatch di Groovy e, dato che non vine

conservato lo stato dell'attore tra la gestione di un messaggio e il successivi (questo vale anche per ReactiveActor), generalmente fornisce prestazioni migliori di quelle fornite dai discendenti di DefaultActor.

```
import groovyx.gpars.actor.Actors
import groovyx.gpars.actor.DynamicDispatchActor
final class MyActor extends DynamicDispatchActor {
    void onMessage(String message) {
        println 'Received string'
    }
    void onMessage(Integer message) {
        println 'Received integer'
        reply 'Thanks!'
    }
    void onMessage(Object message) {
        println 'Received object'
        sender.send 'Thanks!'
    }
    void onMessage(List message) {
        println 'Received list'
        stop()
    }
}
final def myActor = new MyActor().start()
Actors.actor {
    myActor 1
    myActor ''
    myActor 1.0
    myActor(new ArrayList())
    myActor.join()
}.join()
```

In determinati scenari, ad esempio quando il comportamento dell'attore non deve variare in base ai messaggi ricevuti in passato, la struttura del codice in un DynamicDispatchActor può risultare più intuitiva rispetto a quella di un attore che utilizzi una struttura costituita da chiamate a loop() e a react() annidate.
La classe DynamicDispatchActor mette a disposizione un metodo per aggiungere dinamicamente ad un attore degli ulteriori message handlers all'atto della sua creazione o in un momento successivo.
Si tratta del costrutto when{} che viene opzionalmente inserito in un metodo become{}:

```
final Actor myActor = new DynamicDispatchActor().become {
    when {String msg -> println 'A String'; reply 'Thanks'}
    when {Double msg -> println 'A Double'; reply 'Thanks'}
    when {msg -> println 'A something ...'; reply 'What was that?';stop()}
}
myActor.start()
Actors.actor {
    myActor 'Hello'
    myActor 1.0d
    myActor 10 as BigDecimal
    myActor.join()
}.join()
```

Ovviamente i due approcci possono essere combinati:

```
final class MyDDA extends DynamicDispatchActor {
    void onMessage(String message) {
        println 'Received string'
```

```
    }
    void onMessage(Integer message) {
        println 'Received integer'
    }
    void onMessage(Object message) {
        println 'Received object'
    }
    void onMessage(List message) {
        println 'Received list'
        stop()
    }
}
final def myActor = new MyDDA().become {
    when {BigDecimal num -> println 'Received BigDecimal'}
    when {Float num -> println 'Got a float'}
}.start()
Actors.actor {
    myActor 'Hello'
    myActor 1.0f
    myActor 10 as BigDecimal
    myActor.send([])
    myActor.join()
}.join()
```

I message handlers creati con when{} hanno la precedenza sugli handlers definiti nell'attore.
Come sempre, DynamicDispatchActor può utilizzare i thread del pool in modo equo o non equo, di default è utilizzato il comportamento non equo perché fornisce le migliori prestazioni.
Per creare un attore equo si può utilizzare il factory method fairMessageHandler() o il metodo makeFair().

```
def fairActor = Actors.fairMessageHandler {…}
```

Reactive Actor

La classe ReactiveActor rappresenta un tipo di attore che viene solitamente costruito mediante una chiamata a Actors.reactor() o a DefaultPGroup.reactor(); il reactive actor costituisce un attore, sempre privo di stato, ma più orientato agli eventi rispetto a DynamicDispatchActor.
Quando un reactive actor riceve un messaggio, la chiusura contenuta nel corpo dell'attore viene eseguita utilizzando il messaggio come parametro, il risultato della computazione viene inviato come risposta al mittente del messaggio originario.

```
final def group = new DefaultPGroup()
final def doubler = group.reactor {
    2 * it
}
group.actor {
    println 'Double of 10 = ' + doubler.sendAndWait(10)
}
group.actor {
    println 'Double of 20 = ' + doubler.sendAndWait(20)
}
group.actor {
    println 'Double of 30 = ' + doubler.sendAndWait(30)
}
for(i in (1..10)) {
    println "Double of $i = ${doubler.sendAndWait(i)}"
}
```

```
doubler.stop()
doubler.join()
```

Essenzialmente un reactive actor fornisce una conveniente scorciatoia per la costruzione di un attore che attenda l'arrivo dei messaggi, li processi, e invii al mittente il risultato della computazione di ogni messaggio.

```
public class ReactiveActor extends DefaultActor {

    Closure body

    void act() {
        loop {
            react {message ->
                reply body(message)
            }
        }
    }
}
```

Anche ReactiveActor può essere equo o non equo.
Si usi il factory method fairReactor() o il metodo makeFair() per costruire reactive actors equi.

```
def fairActor = Actors.fairReactor {...}
```

6.4) Active Objects

Il concetto di active object fornisce agli attori una facciata in stile orientato agli oggetti, permettendo all'utente di evitare il confronto diretto con il meccanismo degli attori, la necessità di gestire messaggi e comunicazioni.

Attori dalle sembianze amichevoli

```
import groovyx.gpars.activeobject.ActiveObject
import groovyx.gpars.activeobject.ActiveMethod
@ActiveObject
class Decryptor {
    @ActiveMethod
    def decrypt(String encryptedText) {
        return encryptedText.reverse()
    }
    @ActiveMethod
    def decrypt(Integer encryptedNumber) {
        return -1*encryptedNumber + 142
    }
}
final Decryptor decryptor = new Decryptor()
def part1 = decryptor.decrypt(' noitcA ni yvoorG')
def part2 = decryptor.decrypt(140)
def part3 = decryptor.decrypt('noittide dn')
print part1.get()
print part2.get()
println part3.get()
```

Un oggetto attivo deve essere marcato usando l'annotazione @ActiveObject, questo assicurerà che

verrà creato un attore (invisibile all'utente) per ogni istanza creata della classe annotata con @ActiveObject.
L'annotazione @ActiveMethod serve per indicare che il metodo su cui è stata posta dovrà essere eseguito asincronamente dall'attore interno all'oggetto.

Di default, tutti i metodi sono impostati come non bloccanti, tuttavia i metodi che dichiarino esplicitamente il tipo del risultato restituito devono essere configurati come bloccanti, oppure il compilatore segnalerà un errore.
Un metodo non bloccante può solo ritornare valori di tipo def, void e DataflowVariable.
Un metodo viene configurato come bloccante impostando a vero il valore dell'attributo blocking.

```
@ActiveMethod(blocking=true)
```

GPars tradurrà internamente le chiamate ai metodi in messaggi inviati all'attore interno, l'attore si occuperà eventualmente dei messaggi applicando il comando desiderato in vece dell'oggetto che ha chiamato il metodo e, una volta terminata la corrispondente computazione, gli invierà il risultato.
Chiamate a metodi non bloccanti restituiranno promesse, cioè variabili Dataflow.

Bloccante significa non asincrono

Se un metodo è marcato come bloccante, il chiamante si bloccherà in attesa del risultato, come accadrebbe nel caso di una qualsiasi chiamata ad un metodo di una classe; si è ottenuto di essere al sicuro da problemi di concorrenza all'interno dell'oggetto, ma questo è garantito anche da una chiamata a synchronized.
Come si è potuto dedurre, sono le chiamate non bloccanti che dovrebbero far gravitare l'utente verso l'uso degli active objects.
I metodi bloccanti sono tuttavia utili quando vengono utilizzati assieme a dei metodi non bloccanti perché offrono garanzia di consistenza tra le varie chiamate concorrenti.

```
import groovyx.gpars.activeobject.ActiveMethod
import groovyx.gpars.activeobject.ActiveObject
import groovyx.gpars.dataflow.DataflowVariable
@ActiveObject
class Decryptor {
    @ActiveMethod(blocking=true)
    String decrypt(String encryptedText) {
        encryptedText.reverse()
    }
    @ActiveMethod(blocking=true)
    Integer decrypt(Integer encryptedNumber) {
        -1*encryptedNumber + 142
    }
}
final Decryptor decryptor = new Decryptor()
print decryptor.decrypt(' noitcA ni yvoorG')
print decryptor.decrypt(140)
println decryptor.decrypt('noittide dn')
```

Semantica non bloccante
La chiamata ad un metodo non bloccante restituirà la promessa di un risultato non appena il corrispondente messaggio sia stato inviato all'attore interno; l'entità chiamante sarà in grado di procedere come meglio crede, mentre l'attore si occuperà della computazione.
Lo stato della computazione può essere interrogato utilizzando l'attributo bound della promessa.

Una chiamata al metodo get() della promessa bloccherà il chiamante finché un risultato non sarà disponibile; get() restituirà eventualmente un valore o lancerà un'eccezione, a seconda dell'evolversi della computazione.

Regole sulle annotazioni

Bisogna seguire alcune regole nell'annotazione degli oggetti:

1. Annotazioni ActiveMethod sono accettate solamente in classi annotate come ActiveObject.
2. É possibile annotare con ActiveMethod solo metodi che saranno contenuti in istanze, non metodi statici.
3. Metodi annotati con ActiveMethod possono essere sostituiti da varianti non attive e viceversa.
4. Sottoclassi di un oggetto attivo possono dichiarare ulteriori metodi attivi, premesso che siano a loro volta annotate come ActiveObject.
5. L'uso contemporaneo di metodi attivi e non attivi può risultare in race conditions, idealmente tutti i metodi non privati di un oggetto attivo dovrebbero essere marcati come attivi.

Ereditarietà

L'annotazione @ActiveObject può apparire in ogni punto di una gerarchia ereditaria ma il campo destinato a contenere l'attore verrà creato solo nella prima classe annotata della gerarchia. Le altre classi lo riutilizzeranno.

```
import groovyx.gpars.activeobject.ActiveObject
import groovyx.gpars.activeobject.ActiveMethod
import groovyx.gpars.dataflow.DataflowVariable
@ActiveObject
class A {
    @ActiveMethod
    def fooA(value) {
        …
    }
}
class B extends A {
}
@ActiveObject
class C extends B {
    @ActiveMethod
    def fooC(value1, value2) {
        …
    }
}
```

In questo esempio, il campo attore sarà generato nella classe A. La classe C dovrà essere annotata con @ActiveObject perché contiene un metodo, fooC, annotato con @ActiveMethod.
La classe B non necessita di annotazioni perché non contiene alcun metodo attivo.

Gruppi

Così come gli attori vengono raggruppati attorno a thread pool, anche gli oggetti attivi possono essere configurati in modo da utilizzare i thread appartenenti ad un determinato gruppo.

```
@ActiveObject("group1")
class MyActiveObject {
    …
}
```

Il parametro value dell'annotazione @ActiveObject specifica il nome del thread pool a cui associare l'attore interno all'oggetto; solo thread appartenenti pool specificato verranno utilizzati per l'esecuzione degli attori interni alle istanze della classe in cui questo è specificato.
Ovviamente il thread pool deve essere creato prima della creazione di istanze appartenenti al pool stesso.
In assenza della specifica di un thread pool nell'annotazione @ActiveObject, gli attori creati con le istanze dell'oggetto utilizzeranno il pool di default: Actors.defaultActorPGroup.

```
final DefaultPGroup group = new DefaultPGroup(10)
ActiveObjectRegistry.instance.register("group1", group)
```

Assegnare un nome all'attore interno

Capiterà molto raramente di incorrere in un conflitto con il nome di default dell'attore interno ad un oggetto attivo; nel caso in cui questo si verifichi, è sufficiente assegnare un nome personalizzato specificandolo nell'annotazione @ActiveObject:

```
@ActiveObject(actorName = "alternativeActorName")
class MyActiveObject {
    …
}
```

7) Agent

In computer science il concetto di agente si riferisce normalmente ad un'entità software complessa, in grado di operare parzialmente in autonomia, che si occupa di portare a compimento un compito assegnatogli.
In GPars il concetto di agente si riferisce a qualcosa di completamente diverso: deriva da Clojure e rappresenta una struttura, concettualmente molto simile ad un attore, che si preoccupa di proteggere dei dati o delle risorse che devono necessariamente essere condivise tra thread diversi.
L'esempio classico è rappresentato da un carrello della spesa in un ambiente di e-commerce.

Un agente protegge i dati al suo interno dall'accesso diretto, un client che voglia utilizzare i dati protetti dall'agente deve inviare all'agente la lista dei comandi (funzioni, chiusure) da applicare ai dati, i comandi verranno serializzati ed eseguiti dall'agente, internamente, sui dati protetti.
In pratica, un agente protegge un dato o dei dati che devono necessariamente essere condivisi consentendone la modifica ad un solo thread controllato dall'agente stesso.
Le richieste di elaborazione dei dati ricevute dall'agente vengono processate sequenzialmente e il risultato della computazione diventerà il nuovo stato interno dell'agente; la sequenzialità della computazione permette all'agente di garantire in ogni momento la consistenza dei dati.

In questo semplicissimo esempio, si utilizza un agente per proteggere un intero.

```
agent = new Agent(0)    //Creazione di un nuovo agente contenente un intero di
                        //valore iniziale 0
agent.send {increment()}      //invio della funzione increment() all'agente
…
//Dopo qualche tempo, necessario all'elaborazione del messaggio, lo stato
//dell'agente è stato aggiornato
…
assert agent.val== 1
```

Così come gli attori, anche gli agenti elaborano i messaggi ricevuti in modo asincrono.

7.1) Concetti fondamentali

GPars mette a disposizione la classe agente: si tratta di una implementazione degli agenti ispirata a Clojure che presenta la caratteristica di essere thread-safe e non bloccante.
Un agente prende in consegna il riferimento ad un oggetto[52], lo memorizza in un campo interno e lo nasconde dall'esterno rendendone possibile l'accesso solo attraverso l'agente stesso.
L'agente si mette poi in attesa di ricevere dei messaggi che conterranno le chiusure o i comandi da eseguire sull'oggetto; i messaggi potranno essere inviati all'agente utilizzando gli stessi metodi usati per l'invio di un messaggio ad un attore: si tratta dell'operatore '<<', il metodo send() e la chiamata implicita al metodo call().
L'agente esegue le chiusure ricevute tramite messaggio in sequenza, garantendo ad ognuna il controllo esclusivo (e la possibilità di modifica) della risorsa protetta dall'agente durante l'intero intervallo temporale necessario alla computazione.
Il nuovo valore di volta in volta assunto dall'oggetto protetto diventa il nuovo stato dell'agente.

L'intero processo è di tipo fire and forget: il messaggio è inviato all'agente dal un processo mittente che, una volta inviato il messaggio, potrà liberamente procedere nella sua esecuzione inviando dopo

52 Deve trattarsi di qualcosa che debba essere necessariamente condiviso tra più thread ma per cui non sia possibile un accesso diretto concorrentemente sicuro. L'esempio banale è costituito da una stampante.

qualche tempo una richiesta all'agente per conoscerne lo stato corrente.

Regole fondamentali

- I comandi inviati all'agente ricevono lo stato dell'agente come parametro al momento della loro esecuzione.
- I comandi e le chiusure inviate all'agente possono eseguire qualunque metodo sullo stato dell'agente.
- É possibile rimpiazzare lo stato dell'agente indipendentemente dallo stato corrente utilizzando il metodo updateValue().
- Il valore di ritorno della chiusura inviata all'agente non ha alcun significato ed è ignorato.
- Se il messaggio inviato all'agente non è una chiusura, viene considerato come nuovo valore da far assumere all'oggetto interno all'agente.
- Una chiamata all'attributo val dell'agente attenderà che tutti le chiamate precedentemente effettuate all'agente vengano eseguite prima di restituire lo stato dell'agente stesso.
- Una chiamata alla proprietà instantVal restituisce immediatamente lo stato corrente dell'agente.
- Il metodo valAsync si comporterà come una chiamata a val senza bloccare il processo chiamante.
- Tutti gli agenti utilizzeranno lo stesso thread pool se non diversamente specificato.
  É possibile specificare quale thread pool debba utilizzare un agente impostandone l'attributo threadPool.
- Le eccezioni generate durante l'esecuzione dei comandi inviati all'agente possono essere recuperate tramite l'attributo errors.

Esempi

Elenco condiviso
Supponiamo di voler simulare un'urna contenente dei foglietti in cui sia scritto il nome di un membro del gruppo a cui l'urna si riferisce[53]; per aggiungere un nuovo membro bisognerà inserire nell'urna un nuovo foglietto in cui sia specificato il suo nome.
Sia l'urna simulata da un agente che riceverà tramite messaggi i nominativi dei membri da inserire.

```
import groovyx.gpars.agent.Agent
import java.util.concurrent.ExecutorService
import java.util.concurrent.Executors
/**
 * Creazione di un nuovo agente contenente una lista di stringhe
 */
def jugMembers = new Agent<List<String>>(['Me'])     //aggiungi Me
jugMembers.send {it.add 'James'}    // aggiungi James
final Thread t1 = Thread.start {
    jugMembers.send {it.add 'Joe'}  // aggiungi Joe
}
final Thread t2 = Thread.start {
    jugMembers << {it.add 'Dave'}   // aggiungi Dave (usando l'operatore << )
    jugMembers {it.add 'Alice'}     // aggiungi Alice (usando una chiamata
                                    //implicita al metodo call() )
}
[t1, t2]*.join()
```

53 Per fissare le idee, si pensi al modo in cui potrebbe avvenire la scelta di un elemento in un insieme chiuso, l'esempio si riferisce alla costruzione di tale insieme.

```
println jugMembers.val
jugMembers.valAsync {println "Current members: $it"}
jugMembers.await()
```

Convegno con registrazione dei partecipanti
Esempio leggermente più complesso del precedente: un agente tiene traccia del numero di partecipanti ad un congresso a cui è possibile iscriversi o cancellarsi. L'iscrizione e la cancellazione sono possibili solamente tramite comandi inviati all'agente.

```
import groovyx.gpars.agent.Agent
/**
 * Conference è un agente (contenente un intero) contenente il numero di
 * partecipanti ad una data conferenza; permette a delle entità esterne di
 * iscrivere o di rimuovere partecipanti dalla lista degli iscritti.
 * Conference estende la classe Agent aggiungendo i metodi register() e
 * unregister() che potranno essere utilizzati per aggiungere o rimuovere
 * partecipanti dalla conferenza.
 */
class Conference extends Agent<Long> {
    def Conference() { super(0) }
    private def register(long num) { data += num }
    private def unregister(long num) { data -= num }
}
final Agent conference = new Conference()      //Crea una nuova conferenza
/**
 * Tre entità esterne cercheranno di iscrivere o cancellare dei partecipanti
 * concorrentemente
 */
final Thread t1 = Thread.start {
    conference << {register(10L)}   //invio di un comando per la registrazione
                                    //di 10 partecipanti
}
final Thread t2 = Thread.start {
    conference << {register(5L)}    //invio di un comando per la registrazione
                                    //di 5 partecipanti
}
final Thread t3 = Thread.start {
    conference << {unregister(3L)} //invio di un comando per la rimozione
                                    //di 3 partecipanti
}
[t1, t2, t3]*.join()
assert 12L == conference.val
```

Factory methods

Si possono creare istanze di agenti anche utilizzando il factory method Agent.agent()

```
def jugMembers = Agent.agent ['Me']  //aggiungi Me
```

7.2) Osservatori e controllori

Si può incorrere nella necessità di monitorare o limitare i cambiamenti di stato di un agente, a questo scopo l'implementazione del modello di agente in GPars mette a disposizione gli osservatori (listener) e i controllori (validators).
L'agente invia una notifica a tutti gli osservatori e i controllori ad esso collegati ad ogni variazione

del suo stato, i controllori hanno la possibilità di rigettare il cambiamento lanciando un'eccezione.

```
final Agent counter = new Agent()
counter.addListener {oldValue, newValue -> println "Changing value from
$oldValue to $newValue"}
counter.addListener {agent, oldValue, newValue -> println "Agent $agent changing
value from $oldValue to $newValue"}
counter.addValidator {oldValue, newValue -> if (oldValue > newValue) throw new
IllegalArgumentException('Things can only go up in Groovy')}
counter.addValidator {agent, oldValue, newValue -> if (oldValue == newValue)
throw new IllegalArgumentException('Things never stay the same for $agent')}
counter 10
counter 11
counter {updateValue 12}
counter 10  //Questo cambiamento verrà rigettato
counter {updateValue it - 1}  //Questo cambiamento verrà rigettato
counter {updateValue it}  //Questo cambiamento verrà rigettato
counter {updateValue 11}  //Questo cambiamento verrà rigettato
counter 12  //Questo cambiamento verrà rigettato
counter 20
counter.await()
```

Osservatori e controllori sono essenzialmente delle chiusure con due o tre parametri di ingresso.
Le eccezioni lanciate dai controllori sono loggate nell'agente, si può verificarne la presenza utilizzando il metodo hasErrors() o estrarle usando la proprietà errors.

```
assert counter.hasErrors()
assert counter.errors.size() == 5
```

Possibili problemi di validazione
Se le chiusure inviate all'agente ne modificano lo stato direttamente, un controllore potrebbe non essere in grado di annullare la modifica.
Esistono un paio di metodi per ovviare a questo problema:

1. Assicurarsi di non modificare mai direttamente l'oggetto che rappresenta lo stato dell'agente.
2. Permettere all'agente di creare copie del proprio stato, in modo da poter aggiornare il valore dell'oggetto protetto dall'agente solo dopo aver validato la modifica effettuata alla copia.

Esempio: il carrello della spesa

Costituisce uno dei casi più ovvi di come non sia sempre possibile evitare la condivisione di un oggetto da parte di più thread.

```
import groovyx.gpars.agent.Agent
class ShoppingCart {
    private def cartState = new Agent([:])
//Metodi pubblici
    public void addItem(String product, int quantity) {
        cartState << {it[product] = quantity}  //L'operatore << invia un
                                               //messaggio all'agente
    }
    public void removeItem(String product) {
        cartState << {it.remove(product)}
    }
    public Object listContent() {
        return cartState.val
    }
```

```
    public void clearItems() {
        cartState << performClear
    }
    public void increaseQuantity(String product, int quantityChange) {
        cartState << this.&changeQuantity.curry(product, quantityChange)
    }
//Metodi privati
    private void changeQuantity(String product, int quantityChange, Map items) {
        items[product] = (items[product] ?: 0) + quantityChange
    }
    private Closure performClear = { it.clear() }
}

//Script
final ShoppingCart cart = new ShoppingCart()
cart.addItem 'Pilsner', 10
cart.addItem 'Budweisser', 5
cart.addItem 'Staropramen', 20
cart.removeItem 'Budweisser'
cart.addItem 'Budweisser', 15
println "Contents ${cart.listContent()}"
cart.increaseQuantity 'Budweisser', 3
println "Contents ${cart.listContent()}"
cart.clearItems()
println "Contents ${cart.listContent()}"
```

In questo esempio si sono utilizzate un paio di strategie implementative:
1) I metodi pubblici possono solo inviare all'agente il codice da eseguire, non possono modificare direttamente il valore contenuto.

Questo implica che codice sequenziale come
```
public void addItem(String product, int quantity) {
    cartState[product]=quantity
}
```

Diventi
```
public void addItem(String product, int quantity) {
    cartState << {it[product] = quantity}
}
```

2) I metodi pubblici possono indicare all'agente quali sono i metodi privati da utilizzare per ottenere la funzionalità richiesta

```
public void clearItems() {
    cartState << performClear
}
private Closure performClear = { it.clear() }
```

Il currying potrebbe essere necessario, se la chiusura utilizza altri argomenti oltre allo stato dell'agente.
Si veda il metodo increaseQuantity.

7.3) Raggruppamento

Di default tutti gli agenti utilizzano lo stesso thread pool.
É possibile creare agenti anche su thread pool creati dagli utenti utilizzando il factory method

agent() sul gruppo.
Questo permette, come avviene per gli attori, di ottimizzare le prestazioni di diversi gruppi di attori.

```
final def group = new NonDaemonPGroup(5)  //create a group around a thread pool
def jugMembers = group.agent(['Me'])  //add Me
```

Si noti come gli agenti siano strutturati per essere eseguiti da demoni; nel caso si abbia la necessità di eseguire degli agenti utilizzando un pool composto da thread che non siano dei demoni, sarà necessario avere cura di terminare esplicitamente il gruppo parallelo o il thread pool utilizzando il metodo shutdown(), altrimenti sarà impossibile uscire dall'applicazione.

Sostituzione del thread pool
Alternativamente, è possibile collegare direttamente un agente ad un particolare thread pool chiamando il metodo attachToThreadPool() sull'istanza dell'agente che si vuole collegare ad un pool specifico.

```
def jugMembers = new Agent<List<String>>(['Me'])  //Aggiungi Me
final ExecutorService pool = Executors.newFixedThreadPool(10)
jugMembers.attachToThreadPool(new DefaultPool(pool))
```

Si ricorda che, come gli attori, anche gli agenti non possono mai essere eseguiti da più di un thread contemporaneamente.

7.4) Lettura dello stato dell'agente

Seguendo l'approccio di clojure, la classe Agente considera le operazioni di lettura come di priorità più alta rispetto alle operazioni di scrittura: utilizzando una chiamata ad instantVal, la richiesta di lettura scavalcherà tutti i messaggi presenti in nella coda dell'agente e restituirà un'istantanea dello stato dell'agente.
Una chiamata a val o a valAsync(chiusura cl) verrà invece inserita in coda ed attenderà il suo turno come una qualsiasi altra richiesta inviata all'attore.

É necessario notare come, per quanto l'istantanea dello stato restituita da instantVal sia corretta, potrebbe non essere sensata perché non è possibile determinare a quale istante temporale o in quale punto della coda di richieste inviate all'agente venga eseguita.

Il metodo await() permette di attendere l'esecuzione di tutti i messaggi precedentemente ricevuti dall'agente, di conseguenza blocca il thread chiamante.

Copia dello stato interno

Per evitare di rendere accessibile al pubblico l'oggetto protetto dall'agente (perché magari si tratta dell'indirizzo di una risorsa), la classe Agent permette di specificare un metodo di copia come secondo parametro da passare al suo costruttore.
Quando un metodo di copia è specificato, qualora l'agente riceva la richiesta di rendere noto il suo stato, quello che effettivamente viene reso noto è l'output del metodo di copia applicato allo stato e non lo stato stesso.
Questo si applica a instantVal, val e a valAsync.

7.5) Gestione degli errori

Le eccezioni generate dall'esecuzione dei comandi inviati all'agente vengono immagazzinate nell'agente e possono essere recuperate attraverso l'attributo errors; l'attributo errors viene resettato una volta letto.

```
def jugMembers = new Agent<List>()
    assert jugMembers.errors.empty
    jugMembers.send {throw new IllegalStateException('test1')}
    jugMembers.send {throw new IllegalArgumentException('test2')}
    jugMembers.await()
    List errors = jugMembers.errors
    assertEquals(2, errors.size())
    assert errors[0] instanceof IllegalStateException
    assertEquals 'test1', errors[0].message
    assert errors[1] instanceof IllegalArgumentException
    assertEquals 'test2', errors[1].message
    assert jugMembers.errors.empty
```

7.6) Agenti equi e non equi

Così come gli attori, anche gli agenti possono essere equi o non equi.
Un agente equo rilascia il thread che sta utilizzando ogni qualvolta abbia processato un messaggio mentre un agente non equo mantiene il controllo sul thread che lo sta eseguendo finché sia presente almeno un messaggio in coda.
Come sempre, un agente non equo tende a fornire prestazioni migliori rispetto ad un agente equo.
Un agente è non equo di default e può essere trasformato in un agente equo tramite una chiamata al metodo makeFair().

```
def jugMembers = new Agent<List>(['Me'])  //Aggiungi Me
    jugMembers.makeFair()
```

8) Dataflow

Il concetto di dataflow è molto simile al concetto di CSP, entrambi si basano sulla strutturazione di una rete di processi ma, come suggerisce il nome, mentre in CSP il punto focale è il processo nel concetto di dataflow il punto focale è il dato che naviga nella dataflow network.
Altre fondamentali differenze tra la concorrenza basata su dataflow e il CSP è che la comunicazione tra processi è generalmente asincrona[54] e che i canali di comunicazione non sono strettamente punto-punto[55].
Nella concorrenza basata su dataflow un processo è visto come una scatola nera con input e output ben definiti che viene eseguita non appena tutti i suoi input siano disponibili; un po' come un operaio in una catena di montaggio.

Com'è facile immaginare, non è sempre possibile modellare un sistema parallelo come una rete di scatole nere con input ed output ben definiti e questo suggerisce come il modello di concorrenza basato su dataflow non si presti all'implementazione degli algoritmi più complessi.

La concorrenza basata su dataflow si basa su alcuni punti chiave.
Una variabile dataflow[56] può essere inizializzata una sola volta, può essere letta prima della sua inizializzazione (nel qual caso il processo che sta tentando di leggerla si sospenderà[57]) e può essere condivisa tra un numero arbitrario di processi.
Non è possibile modificare il valore di una variabile dataflow una volta che sia stata inizializzata, questo evita possibili race conditions.
Il fatto che una variabile Dataflow possa essere inizializzata una sola volta e il fatto che la concorrenza nel modello basato su dataflow sia deterministica comportano un interessante effetto collaterale: il deadlock diventa a sua volta deterministico.

Un po' di teoria - Deadlock

Le variabili dataflow introducono un ordinamento parziale nei task in cui sono utilizzate; vediamo un esempio

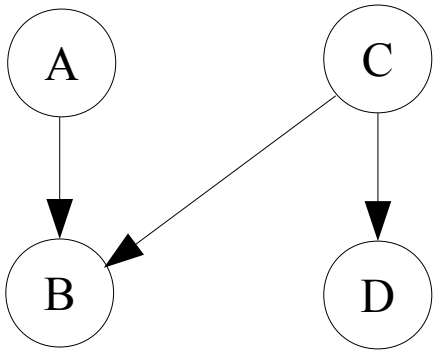
```
//task 1
task {
    do_A
    read_dataflow
    do_B
}
//task 2
task {
    do_C
    write_dataflow
    do_D
}
```

Il task 2 scrive su una variabile dataflow che viene letta dal task 1; l'operazione di lettura e l'operazione di scrittura dividono ogni task in due parti, chiamate A, B, C, e D.
Utilizzando ora la notazione “x < y” per indicare che la parte “x” viene eseguita prima della parte

54 La possibilità di comunicare in modo sincrono verrà introdotta nella versione 1.0 di GPars.
55 Per quanto anche in CSP un canale possa essere trasformato da 1 a 1 in 1 a molti senza sforzo, ad esempio.
56 In Groovy si tratta di un'istanza della classe DataflowVariable
57 Se il processo che sta cercando di accedere alla variabile non verifica preventivamente che la variabile sia stata inizializzata.

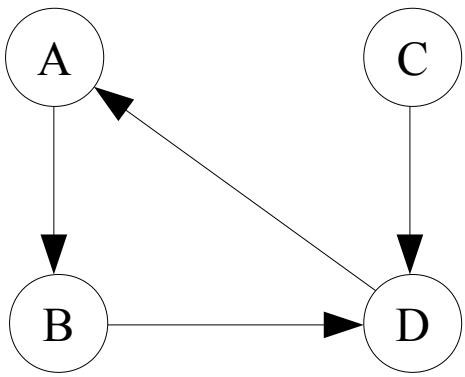
“y” si ottiene il seguente ordinamento: A < B e C < D dal fatto che i componenti di ogni task vengono eseguiti in sequenza.
La variabile dataflow non può essere letta prima di essere inizializzata: un task che tenti di leggere una variabile dataflow prima della sua inizializzazione si blocca in attesa che venga assegnato un valore alla variabile; questo genera un terzo vincolo di ordinamento: C < B.
La variabile dataflow assicura implicitamente che il blocco B possa essere eseguito solo dopo l'esecuzione del blocco C, ad un utente non è richiesto alcuno sforzo aggiuntivo per assicurarsi che il blocco B abbia a disposizione tutti i dati di cui necessita per poter essere processato correttamente.

Identificando le relazioni di ordinamento tra le varie parti di processo con i lati e le parti di processo con i nodi di un grafo orientato, è possibile dare una rappresentazione grafica del problema in esame:



Com'è possibile intuire, la ricerca di una situazione di deadlock è riconducibile alla ricerca di un ciclo nel grafo orientato[58]; vediamolo con un altro esempio:

```
//task 1
task {
    read_dataflow_2
    do_A
    do_B
    write_dataflow_1
}
//task 2
task {
    do_C
    read_dataflow_1
    do_D
    write_dataflow_2
}
```



58 Questo procedimento ricorda quello utilizzato per verificare la correttezza di un grafo di precedenza[30][31]

I nodi A, B, e D formano un ciclo e le parti di processo rappresentate non potranno mai essere eseguite.

8.1) Concetti di base

Dataflow programming

Si tratta di un paradigma di programmazione in cui un programma viene modellato come un grafo orientato rappresentante il flusso dei dati attraverso le varie operazioni, rappresentate dai nodi del grafo.
Nella programmazione basata su dataflow, ci si focalizza su come gli operatori sono connessi, al contrario della programmazione imperativa in cui si basa su come gli operatori vengono eseguiti.

Gli operatori sono delle "scatole"[59] dotate di input ed output sempre esplicitamente definiti; un operatore viene eseguito non appena tutti gli input a cui è collegato diventano validi (o non appena diventano validi una parte degli input a cui è collegato, se l'operatore ha la facoltà di scegliere tra gli input disponibili).
Mentre un programma tradizionale è strutturato come una serie di istruzione del tipo "fa questo, dopo fa quest'altro" la struttura di un codice che implementi il concetto di dataflow ricorda una serie di operai in una catena di montaggio: eseguiranno i compiti a loro assegnati non appena arriveranno i materiali di cui necessitano.
Questo è il motivo per cui i linguaggi che implementano il modello di dataflow sono implicitamente paralleli: gli operatori non devono tener traccia di uno stato e sono tutti "pronti" nello stesso istante.

Variabili dataflow

Nel modello di concorrenza basato su dataflow è possibile condividere in modo sicuro variabili tra task diversi.
Queste variabili sono in Groovy istanze della classe DataflowVariable, possono essere inizializzate (usando l'operatore <<) solo una volta durante la loro intera esistenza e possono essere lette più volte da task diversi (attraverso l'attributo val) prima ancora di venire inizializzate.
Se un task tenta la lettura di una variabile dataflow prima dell'inizializzazione della suddetta, il task in questione si sospenderà fino all'inizializzazione della variabile da parte di un altro task; è quindi possibile utilizzare le variabili dataflow nella scrittura del codice relativo ai vari task senza preoccuparsi di problemi di sincronia che saranno risolti automaticamente dal meccanismo di gestione interno alle variabili stesse.

In breve, generalmente è possibile eseguire tre operazioni sulle variabili dataflow:

- È possibile creare una variabile dataflow
- È possibile leggere una variabile dataflow (sospendendosi se non è stata ancora inizializzata)
- È possibile inizializzare una variabile dataflow.

Un programma scritto utilizzando il modello di dataflow segue tre regole essenziali:

- Se incontra una variabile non inizializzata attende che venga inizializzata.
- Non è possibile cambiare il valore di una variabile dataflow una volta inizializzata.
- Le variabili dataflow semplificano la creazione di stream agents.

59 Altrimenti note come "macchinette mumble mumble". Cit: prof. Trevisan Noè.

Esempio: semplice calcolo eseguito da tre task in esecuzione concorrente.

```
import static groovyx.gpars.dataflow.Dataflow.task
final def x = new DataflowVariable()
final def y = new DataflowVariable()
final def z = new DataflowVariable()
task {
    z << x.val + y.val
}
task {
    x << 10
}
task {
    y << 5
}
println "Result: ${z.val}"
```

Stesso esempio, riscritto utilizzando la classe Dataflow.

```
import static groovyx.gpars.dataflow.Dataflow.task
final def df = new Dataflows[60]()
task {
    df.z = df.x + df.y
}
task {
    df.x = 10
}
task {
    df.y = 5
}
println "Result: ${df.z}"
```

Nell'esempio, tre task vengono eseguiti contemporaneamente; durante la loro esecuzione devono scambiarsi dei dati e fanno questo utilizzando delle variabili dataflow che, come dovrebbe essere chiaro a questo punto, è possibile immaginare come canali di comunicazione one-shot usati per trasferire dati dai produttori ai consumatori.

Le variabili dataflow hanno una semantica piuttosto semplice: se un task deve leggere un valore da una variabile dataflow (utilizzando l'attributo val) si sospende finché il valore non viene inizializzato da un altro task o da un altro thread.
Ogni variabile dataflow può essere inizializzata solo una volta durante la sua esistenza[61].
Si noti come non sia necessario preoccuparsi dell'ordinamento temporale dei task e della sincronizzazione tra task: i valori delle variabili dataflow vengono magicamente trasmessi tra i vari task o thread senza l'intervento del programmatore.

Dataflow Queues e Broadcasts

Le dataflow queue e i dataflow broadcast sono degli altri strumenti messi a disposizione dal modello di concorrenza basato su dataflow; si tratta di canali di comunicazione (implementano l'interfaccia DataflowChannel) thread-safe utilizzati per inserire buffer o code tra task o thread che

60 Può sembrare un errore di battitura ma è invece corretto, si veda [32]

61 Una variabile dataflow già inizializzata accetta silenziosamente una nuova inizializzazione se il nuovo valore con cui si vuole inizializzare la variabile corrisponde al valore già assunto. É possibile impedire questo comportamento utilizzando il metodo bindUnique.

debbano scambiarsi messaggi.
Esempio: produttore consumatore.

```
import static groovyx.gpars.dataflow.Dataflow.task
def words = ['Groovy', 'fantastic', 'concurrency', 'fun', 'enjoy', 'safe',
'GPars', 'data', 'flow']
final def buffer = new DataflowQueue()
task {
    for (word in words) {
        buffer << word.toUpperCase()  //invia gli elementi di words
    }
}
task {
    while(true) println buffer.val  //consuma dei messaggi dal buffer
}
```

L'interfaccia DataflowChannel combina due interfacce che servono scopi specifici:

- DataflowReadChannel, contenente tutti i metodi necessari per leggere dei messaggi da un canale.
- DataflowWriteChannel, contenete tutti i metodi necessari per scrivere su un canale.

A volte è consigliabile utilizzare direttamente queste sotto interfacce per meglio specificare l'utilizzo dei canali.

Comunicazioni punto-punto
La classe DataflowQueue può essere vista come un canale di comunicazione punto-punto (1 a 1, molti a 1), permette a uno o più produttori di mandare dei messaggi ad un consumatore.
Più consumatori che stiano leggendo dalla stessa DataflowQueue consumeranno messaggi diversi, cioè ogni messaggio sarà consumato da un solo lettore.
Si può immaginare un semplice meccanismo di bilanciamento del carico di lavoro in cui i consumatori vengono aggiunti dinamicamente ad una DataflowQueue quando la parte "consumatore" di un algoritmo produttore-consumatore debba essere potenziata.

Comunicazioni tramite pubblicazione e sottoscrizione
La classe DataflowBroadcast offre un modello di comunicazione di tipo editore-abbonato (cioè un canale di comunicazione del tipo 1 a molti o molti a molti).
Ogni abbonato riceve tutti i messaggi pubblicati da tutti gli editori, cioè ogni messaggio viene ricevuto da tutti i lettori con una sottoscrizione al canale valida al momento della pubblicazione del messaggio.
Un lettore può abbonarsi ad un canale utilizzando il metodo createReadChannel().

```
DataflowWriteChannel broadcastStream = new DataflowBroadcast()
DataflowReadChannel stream1 = broadcastStream.createReadChannel()
DataflowReadChannel stream2 = broadcastStream.createReadChannel()
broadcastStream << 'Message1'
broadcastStream << 'Message2'
broadcastStream << 'Message3'
assert stream1.val == stream2.val
assert stream1.val == stream2.val
assert stream1.val == stream2.val
```

La classe DataflowBroadcast utilizza la classe DataflowStream per l'implementazione del meccanismo di consegna dei messaggi.

DataflowStream

La classe DataflowStream rappresenta un canale dataflow deterministico: è costruita attorno al concetto di functional queue[62] e quindi fornisce un'implementazione lock-free e thread safe per lo scambio di messaggi.
Fondamentalmente si può pensare ad un DataflowStream come ad un canale uno a molti in cui tutti i messaggi arrivano ai lettori nello stesso ordine.
Dato che DataflowStream è implementato come una functional queue, le sue API richiedono che i lettori gestiscano la ricezione dei messaggi dal canale; d'altra parte, la classe DataflowStream offre delle prestazioni interessanti.

La classe DataflowStream non implementa l'interfaccia DataflowChannel, le interfacce che si limitano a fornire i metodi per la lettura e la scrittura in uno stream sono DataflowStreamReadAdapter e DataflowStreamWriteAdapter rispettivamente.

```
import groovyx.gpars.dataflow.stream.DataflowStream
import groovyx.gpars.group.DefaultPGroup
import groovyx.gpars.scheduler.ResizeablePool
/**
Implementazione concorrente dell'algoritmo Sieve of Eratosthenes[63] utilizzando i
dataflow task.
L'algoritmo consiste in una serie di filtri eseguiti concorrentemente in cui
ogni filtro controlla che l'elemento che sta filtrando non sia divisibile per
uno specifico numero primo.
(genera sequenza di numeri 1, 2, 3, 4, 5, ...) -> (filtra per modulo 2) ->
(filtra per modulo 3) -> (filtra per modulo 5) -> (filtra per modulo 7) ->
(filtra per modulo 11) -> (Attenzione! I numeri escono dalla catena qui)

La catena cresce in itinere, mano a mano che nuovi numeri primi vengono
individuati.
**/

/**
È necessario l'uso di un thread pool di dimensione variabile, perché i dataflow
task bloccano il thread che li sta eseguendo (utilizzando DataflowQueue.val)
mentre sono in attesa di un valore.
**/

group = new DefaultPGroup(new ResizeablePool(true))
final int requestedPrimeNumberCount = 100

//Generazione della sequenza di numeri
final DataflowStream candidates = new DataflowStream()
group.task {
    candidates.generate(2, {it + 1}, {it < 1000})
}
/**
Concatena un nuovo filtro per un particolare numero primo alla fine della catena
@param inChannel L'end channel corrente
@param prime Il nuovo numero primo da utilizzare come filtro
@return Un nuovo canale rappresentante il termine della catena
**/
```

62 Una functional queue è una struttura dati in cui le operazioni di accodamento o di rimozione di un elemento non avvengono modificando la struttura dati originaria ma creando ogni volta una nuova struttura dati contenente le modifiche. La vecchia struttura dati non viene distrutta in modo che il codice che la stava utilizzando non sia affetto dalla modifiche apportate.

63 Sieve of Eratosthenes si tratta di un algoritmo per l'individuazione dei numeri primi[33].

```
def filter(DataflowStream inChannel, int prime) {
    inChannel.filter { number ->
        group.task {
            number % prime != 0
        }
    }
}

/**
Consuma l'output della catena e aggiunge filtri per ogni nuovo numero primo
individuato
**/
def currentOutput = candidates
requestedPrimeNumberCount.times {
    int prime = currentOutput.first
    println "Found: $prime"
    currentOutput = filter(currentOutput, prime)
}
```

La classe DataflowStream è progettata per l'utilizzo da parte di produttori e consumatori serviti da un singolo thread; se si suppone che più thread scrivano o leggano dallo stream, l'accesso allo stream deve essere serializzato esternamente oppure è necessario l'uso di adattatori.

DataflowStream Adapters

La semantica e le API relative all'uso di un dataflow stream sono decisamente diverse da quelle necessarie per l'utilizzo di un dataflow channel ed è quindi necessario introdurre degli adattatori che permettano la comunicazione tra stream e channel.
La classe DataflowStreamReadAdapter fornirà i metodi necessari a leggere da un'istanza di DataflowStream e la classe DataflowStreamWriteAdapter fornirà i metodi necessari per scrivere in un'istanza di DataflowStream.

DataflowStreamWriteAdapter è thread safe e permette, quindi, a più thread di accedere allo stesso stream; DataflowStreamReadAdapter non lo è e si suppone che venga utilizzata da un singolo thread.
Se più thread devono leggere dallo stesso stream, ognuno dovrà creare una diversa istanza di DataflowStreamReadAdapter.

Grazie agli adattatori, DataflowStream può essere utilizzato per comunicazioni tra operatori e selettori, che si aspettano di leggere da o scrivere su dataflow channels

```
import groovyx.gpars.dataflow.DataflowQueue
import groovyx.gpars.dataflow.stream.DataflowStream
import groovyx.gpars.dataflow.stream.DataflowStreamReadAdapter
import groovyx.gpars.dataflow.stream.DataflowStreamWriteAdapter
import static groovyx.gpars.dataflow.Dataflow.selector
import static groovyx.gpars.dataflow.Dataflow.operator
/**
Dimostrazione di come sia possibile utilizzare DataflowStreamAdapters per
permettere ad operatori e selettori dataflow di interagire con dei DataflowSream
**/
final DataflowStream a = new DataflowStream()
final DataflowStream b = new DataflowStream()
def aw = new DataflowStreamWriteAdapter(a)
def bw = new DataflowStreamWriteAdapter(b)
```

```
def ar = new DataflowStreamReadAdapter(a)
def br = new DataflowStreamReadAdapter(b)
def result = new DataflowQueue()
def op1 = operator(ar, bw) {
    bindOutput it
}
def op2 = selector([br], [result]) {
    result << it
}
aw << 1
aw << 2
aw << 3
assert([1, 2, 3] == [result.val, result.val, result.val])
op1.stop()
op2.stop()
op1.join()
op2.join()
```

La possibilità di selezionare un valore da più DataflowChannels può essere utilizzata solo attraverso adattatori che incapsulino i DataflowStream.

```
import groovyx.gpars.dataflow.Select
import groovyx.gpars.dataflow.stream.DataflowStream
import groovyx.gpars.dataflow.stream.DataflowStreamReadAdapter
import groovyx.gpars.dataflow.stream.DataflowStreamWriteAdapter
import static groovyx.gpars.dataflow.Dataflow.select
import static groovyx.gpars.dataflow.Dataflow.task
/**
Mostra l'utilizzo di DataflowStreamAdapters per permettere la selezione tra
diversi stream
 */
final DataflowStream a = new DataflowStream()
final DataflowStream b = new DataflowStream()
def aw = new DataflowStreamWriteAdapter(a)
def bw = new DataflowStreamWriteAdapter(b)
def ar = new DataflowStreamReadAdapter(a)
def br = new DataflowStreamReadAdapter(b)
final Select<?> select = select(ar, br)
task {
    aw << 1
    aw << 2
    aw << 3
}
assert 1 == select().value
assert 2 == select().value
assert 3 == select().value
task {
    bw << 4
    aw << 5
    bw << 6
}
def result = (1..3).collect{select()}.sort{it.value}
assert result*.value == [4, 5, 6]
assert result*.index == [1, 0, 1]
```

Bind handlers

```
def a = new DataflowVariable()
a >> {println "The variable has just been bound to $it"}
a.whenBound {println "Just to confirm that the variable has been really set to
```

```
$it"}
...
```

Un bind handler può essere associato a un qualsiasi canale dataflow (quindi ad una variabile, ad una coda o ad un canale di broadcast) utilizzando l'operatore >> o il metodo whenBound().
L'handler verrà eseguito ogni qualvolta un valore verrà inserito nel canale.
Da notare come l'operatore >> e il metodo whenBound() costituiscano chiamate non bloccanti (al contrario di una chiamata a .val).

Dataflow queues e broadcast presentano anche un metodo wheneverBound() che permette di eseguire una chiusura ogni qualvolta un valore venga inserito nel canale.

```
def queue = new DataflowQueue()
queue.wheneverBound {println "A value $it arrived to the queue"}
```

Comunicazioni sincrone

Tutte le comunicazioni in una dataflow network sono asincrone; nell'implementazione di un algoritmo basato su un modello produttore - consumatore entrambe le parti potranno essere eseguite alla massima velocità consentita lasciando che siano i canali dataflow utilizzati a prendersi cura dei problemi, per esempio, un produttore potrebbe inserire sistematicamente dati in eccesso in un canale rispetto a quanti dati un consumatore sia in grado di rimuovere.

Il modello di comunicazione asincrono è un'ottima soluzione di default e si presta magnificamente alla soluzione di problemi semplici ma, cosa accade a tutti quei dati di tipo work in progress che si accumulano nei canali ?
Ci si può trovare nella situazione di dover limitare l'occupazione di memoria di un algoritmo implementato con il modello basato su dataflow, esistono varie tecniche per affrontare questo problema, una delle più semplici consiste nel ricorrere a comunicazioni sincrone[64].

Canali sincroni
I canali sincroni risolvono il problema dell'accumulo di dati work in progress: un canale sincrono non solo blocca un lettore se nel canale non è disponibile un messaggio, ma impedisce l'inserimento di un nuovo messaggio nel canale se è già presente un altro messaggio in attesa di consegna.
Questo modello di comunicazione ricorda un po' quello utilizzato in CSP[65]; i canali sincroni possono essere 1:1, N:1, 1:N o N:M.

In questo esempio, un produttore molto rapido invia dati ad un consumatore lento; la natura sincrona del canale di comunicazione assicura che il produttore non sopravanzerà mai troppo il consumatore.

```
final SyncDataflowQueue channel = new SyncDataflowQueue()
def producer = task {
      (1..30).each {
            channel << it      //trasmissione nel canale
            println "Just sent $it"
      }
}
def consumer = task {
      while (true) {
```

64 I canali dataflow sincroni sono disponibili in GPars a partire dalla versione 1.0
65 Communicating Sequential Processes

```
            sleep 500         //simulazione di un consumatore lento
            final Object msg = channel.val
            println "Received $msg"
      }
}
producer.join()
```

I canali sincroni possono essere utilizzati ovunque possano essere utilizzati i canali asincroni e possono essere composti con qualunque componente del modello di dataflow.

Operatori sincroni
Anche un operatore può essere "sincronizzato", questo è utile nel caso di una comunicazione broadcast 1:N (realizzata da un canale SyncDataflowBroadcast*)* in cui un produttore invia dati ad un insieme di consumatori, uno dei quali è particolarmente lento.
In questo caso, tutti gli operatori componenti il gruppo procederanno di comune accordo, alla velocità dell'operatore più lento.

```
final SyncDataflowBroadcast channel = new SyncDataflowBroadcast()
def subscription1 = channel.createReadChannel()
def fastConsumer = operator(inputs: [subscription1], outputs: []) {value ->
      sleep 10               //simulazione di un consumatore veloce
      println "Fast consumer received $value"
}
def subscription2 = channel.createReadChannel()
def slowConsumer = operator(inputs: [subscription2], outputs: []) {value ->
      sleep 500              //simulazione di un consumatore lento
      println "Slow consumer received $value"
}
def producer = task {
      (1..30).each {
            println "Sending $it"
            channel << it     //trasmissione nel canale
            println "Sent $it"
      }
}
producer.join()
[fastConsumer, slowConsumer]*.terminate()
```

Come è possibile vedere in questo esempio, si ha una perdita di prestazioni per evitare una crescita incontrollata del consumo di memoria dell'applicazione in esame; questo costituisce un ennesimo esempio di come, a volte, sia necessario scendere a compromessi.

8.2) Task

I dataflow task costituiscono un'implementazione di task o thread mutualmente indipendenti che scambiano dati utilizzando solamente canali dataflow (quindi variabili, code e broadcast) o stream.
I dataflow task, permettono di implementare facilmente gli UML activity diagrams[66] grazie alla chiara espressione delle mutue dipendenze tra i task e al fatto che l'esecuzione dei task è essenzialmente sequenziale.

Esempio: download della pagina iniziale di tre siti, ogni download avviene in un task distinto e successiva ricerca della chiave "Groovy" tra i contenuti scaricati.

66 Activity diagram si tratta di diagrammi per la rappresentazione di workflow, concettualmente sono simili ai flowchart [34].

```
import static groovyx.gpars.GParsPool.*
import groovyx.gpars.dataflow.DataflowVariable
import static groovyx.gpars.dataflow.Dataflow.task

/**
Scaricamento dei contenuti
**/
def dzone = new DataflowVariable()
def jroller = new DataflowVariable()
def theserverside = new DataflowVariable()
task {
    println 'Started downloading from DZone'
    dzone << 'http://www.dzone.com'.toURL().text
    println 'Done downloading from DZone'
}
task {
    println 'Started downloading from JRoller'
    jroller << 'http://www.jroller.com'.toURL().text
    println 'Done downloading from JRoller'
}
task {
    println 'Started downloading from TheServerSide'
    theserverside << 'http://www.theserverside.com'.toURL().text
    println 'Done downloading from TheServerSide'
}
/**
Task che ricerca la parola chiave "Groovy" tra i contenuti scaricati (la ricerca
avviene ricercando attualmente la chiave "GROOVY" dopo aver convertito tutto in
maiuscolo).
Il task si sincronizza automaticamente con i tre task precedenti che si occupano
del download grazie al funzionamento intrinseco delle variabili dataflow.
**/
task {
    withPool {
        println "Number of Groovy sites today: " +
                ([dzone, jroller, theserverside].findAllParallel {
                    it.val.toUpperCase().contains 'GROOVY'
                }).size()
    }
}.join()
```

Variante: in questo esempio si utilizza un metodo specifico per lo scaricamento delle pagine, il metodo verrà istanziato dal task che si occupa della ricerca della chiave "Groovy" tra i contenuti scaricati.

```
import static groovyx.gpars.GParsExecutorsPool.*
import groovyx.gpars.dataflow.DataflowVariable
import static groovyx.gpars.dataflow.Dataflow.task

final List urls = ['http://www.dzone.com', 'http://www.jroller.com',
'http://www.theserverside.com']
task {
    def pages = urls.collect { downloadPage(it) }
    withPool {
        println "Number of Groovy sites today: " +
                (pages.findAllParallel {
                    it.val.toUpperCase().contains 'GROOVY'
                }).size()
    }
}.join()
```

```
/**
Metodo che si occupa dell'effettivo scaricamento dei contenuti.
Questo esempio mostra come un metodo possa restituire un'istanza di dataflow
variable come valore di ritorno.
**/
def downloadPage(def url) {
    def page = new DataflowVariable()
    task {
        println "Started downloading from $url"
        page << url.toURL().text
        println "Done downloading from $url"
    }
    return page
}
```

Gruppi di task

I dataflow task possono essere raggruppati in modo da utilizzare thread pool specifici (e non quello presente di default) allo scopo di calibrare in modo ottimale le prestazioni dell'algoritmo che si sta implementando.

```
import groovyx.gpars.group.DefaultPGroup
def group = new DefaultPGroup()
group.with {
    task {
        …
    }
    task {
        …
    }
}
```

Il gruppo fornisce il factory method task() per la costruzione di task collegati al thread pool del gruppo.
Il thread pool del gruppo di default è costituito da demoni, ciò richiede di assicurarsi di terminare esplicitamente il gruppo chiamando il metodo shutdown() nel caso in cui il thread pool del gruppo non sia composto da demoni altrimenti l'applicazione che si sta realizzando non terminerà correttamente.

Deadlock deterministico

Nel modello di concorrenza basato sui dataflow, un task blocca il thread che lo sta eseguendo solo quando cerca di leggere (utilizzando l'attributo .val) un valore non ancora disponibile.
Questo, e il fatto che una variabile dataflow possa essere inizializzata una sola volta, implica che gli unici casi in cui sia possibile incorrere in un deadlock siano quelli che rispecchiano l'esempio seguente:

```
task {
    println a.val
    b << 'Hi there'
}
task {
    println b.val
    a << 'Hello man'
}
```

Questo è uno dei benefici del modello di concorrenza basato su dataflow: data la struttura delle dipendenze presente tra gli elementi del modello, se un deadlock si verificherà una volta allora si verificherà sempre, indipendentemente dall'attuale schedulazione dei task.

Dataflow map

Si tratta di una mappa (o di una tabella) costituita da variabili dataflow, come ad esempio:

```
def df = new Dataflows()
df.x = 'value1'
assert df.x == 'value1'
Dataflow.task {df.y = 'value2}
assert df.y == 'value2'
```

in cui le chiavi della mappa sono costituite dal nome delle variabili mentre i valori della mappa sono costituiti dai valori delle variabili.
La semantica per l'accesso e l'inizializzazione delle variabili rimane inalterata ma la sintassi cambia leggermente.
Le dataflow maps vengono utilizzate per esprimere gruppi di variabili dataflow in modo più sintetico.

Sfruttare i dataflow e i blocchi with di Groovy

È possibile creare un blocco with{} di un'istanza di dataflow in modo da poter accedere direttamente alle variabili dataflow contenute nell'istanza senza avere la necessità di identificarle con l'identificatore dell'istanza a cui appartengono (si può accedere ad una variabile scrivendo x invece di df.x).

```
new Dataflows().with {
    x = 'value1'
    assert x == 'value1'
    Dataflow.task {y = 'value2}
    assert y == 'value2'
}
```

Si tratta di una ristrutturazione dell'esempio precedente.

Ritorno di un valore da un task

Un dataflow task può fornire un valore di ritorno sotto forma di variabile dataflow: questo accade quando il task viene creato usando un factory method task() perché il factory method restituisce un'istanza di DataflowVariable che può essere ignorata, oppure può essere utilizzata come qualsiasi altra variabile dataflow.

```
final DataflowVariable t1 = task {
        return 10
    }
    final DataflowVariable t2 = task {
        return 20
    }
    def results = [t1, t2]*.val
    println 'Both sub-tasks finished and returned values: ' + results
```

Ovviamente si può leggere il valore della variabile dataflow associata al task in modo non bloccante usando il metodo whenBound() o l'operatore >>.

Il metodo join(), chiamato su un elenco di variabili dataflow restituite da altrettanti task, serve per bloccare il proseguimento dell'esecuzione di un algoritmo finché tutti i task nell'elenco non abbiano completato la propria esecuzione.

```
task {
     final DataflowVariable t1 = task {
         println 'First sub-task running.'
     }
     final DataflowVariable t2 = task {
         println 'Second sub-task running'
     }
     [t1, t2]*.join()
     println 'Both sub-tasks finished'
 }.join()
```

8.3) Selectors / guards

Quando un nodo di una dataflow network è connesso a più canali di ingresso (di tipo variabile, coda, stream o broadcast), può sorgere la necessità di consumare un solo valore tra quelli disponibili nei canali di ingresso, magari scegliendo tra quelli disponibili secondo un qualche criterio prioritario[67].
Il nodo in questione potrebbe, ad esempio, rappresentare il sistema di controllo di una rete di sensori di vario genere.
La classe Select è costruita per gestire uno scenario di questo tipo: può tenere sotto controllo un gruppo di canali dataflow selezionando di volta in volta un canale di input tra quelli con un input disponibile; il valore letto e l'indice del canale corrispondente vengono inviati al metodo che ha invocato Select.
La scelta del canale di input tra quelli disponibili può essere casuale o può seguire un criterio di priorità, in questo secondo caso i canali all'inizio della lista presente nel costrutto select hanno una priorità più alta.

Selezione di un input tra più canali

Questo esempio mostra un uso basilare di Select: Select monitora un insieme di canali e inoltra i valori che via via trova in input nel suo output indipendentemente dal canale di input in cui sono stati originariamente ricevuti.

```
import groovyx.gpars.dataflow.DataflowQueue
import groovyx.gpars.dataflow.DataflowVariable
import static groovyx.gpars.dataflow.Dataflow.select
import static groovyx.gpars.dataflow.Dataflow.task

def a = new DataflowVariable()
def b = new DataflowVariable()
def c = new DataflowQueue()
task {
    sleep 3000
    a << 10
```

67 Si tratta di guardie di selezione, l'eventuale scelta prioritaria di quale canale utilizzare viene effettuata in base all'indice corrispondente al canale (i canali sono considerati come una lista in cui il primo ha priorità più alta).

```
}
task {
    sleep 1000
    b << 20
}
task {
    sleep 5000
    c << 30
}
def select = select([a, b, c])
println "The fastest result is ${select().value}"
```

Il dato inviato da Select nel canale di output è di tipo SelectResult, include il valore letto dall'input e l'indice del canale corrispondente.

Si può leggere un valore da un costrutto Select in vari modi:

```
def sel = select(a, b, c, d)

//Selezione random
def result = sel.select()

//Selezione random (sintassi compatta)
def result = sel()

//Selezione random con guardie
def result = sel.select([true, true, false, true])

//Selezione random con guardie (sintassi compatta)
def result = sel([true, true, false, true])

//Selezione prioritaria
def result = sel.prioritySelect()

//Selezione prioritaria con guardie
def result = sel.prioritySelect([true, true, false, true])
```

Di default, una chiamata a Select blocca il processo chiamante finché non sia disponibile un valore, alternativamente Select può inviare il valore ad un oggetto MessageStream (un attore) che gli sia stato fornito senza bloccare il processo chiamante.

```
def handler = actor {...}
def sel = select(a, b, c, d)

//Selezione random
sel.select(handler)

//Selezione random (sintassi compatta)
sel(handler)

//Selezione random con guardie
sel.select(handler, [true, true, false, true])

//Selezione random con guardie (sintassi compatta)
sel(handler, [true, true, false, true])

//Selezione prioritaria
sel.prioritySelect(handler)
```

```
//Selezione prioritaria con guardie
sel.prioritySelect(handler, [true, true, false, true])
```

Guardie

Le guardie permettono al processo che invoca Select di eseguire la selezionare su un sotto insieme dei canali di ingresso.
Le guardie sono espresse come una lista di valori booleani inviati ai metodi select() o prioritySelect().

```
def sel = select(leaders, seniors, experts, juniors)

//Solo 'leaders' e 'seniors' sono qualificati come teamLead
def teamLead = sel([true, true, false, false]).value
```

Le guardie sono tipicamente utilizzate per far si che Select sia adattabile a cambiamenti dello stato del processo che lo utilizza.

```
import groovyx.gpars.dataflow.DataflowQueue
import static groovyx.gpars.dataflow.Dataflow.select
import static groovyx.gpars.dataflow.Dataflow.task
/**
Dimostra la possibilità di abilitare/disabilitare canali durante la selezione
tramite l'utilizzo di guardie
**/
final DataflowQueue operations = new DataflowQueue()
final DataflowQueue numbers = new DataflowQueue()
def t = task {
    final def select = select(operations, numbers)
    3.times {
        def instruction = select([true, false]).value
        def num1 = select([false, true]).value
        def num2 = select([false, true]).value
        //La formula è costruita ricavando operatore e operandi dai canali di
        //ingresso
        final def formula = "$num1 $instruction $num2"
        println "$formula = ${new GroovyShell().evaluate(formula)}"
    }
}
task {
    operations << '+'
    operations << '+'
    operations << '*'
}
task {
    numbers << 10
    numbers << 20
    numbers << 30
    numbers << 40
    numbers << 50
    numbers << 60
}
t.join()
```

Selezione prioritaria

La selezione prioritaria viene utilizzata quando le informazioni provenienti da determinati canali devono avere la precedenza rispetto alle informazioni provenienti da altri canali.

Questo esempio mostra un uso basilare della selezione prioritaria: un insieme di canali di ingresso viene monitorato ed input disponibili su canali con un indice più basso nell'insieme di canali monitorati (il primo canale della lista passata a Select ha priorità maggiore) vengono serviti per primi.
L'ordine relativo dei messaggi ricevuti da un singolo canale di ingresso è preservato.

```
def critical = new DataflowVariable()
def ordinary = new DataflowQueue()
def whoCares = new DataflowQueue()
task {
    ordinary << 'All working fine'
    whoCares << 'I feel a bit tired'
    ordinary << 'We are on target'
}
task {
    ordinary << 'I have just started my work. Busy. Will come back later...'
    sleep 5000
    ordinary << 'I am done for now'
}
task {
    whoCares << 'Huh, what is that noise'
    ordinary << 'Here I am to do some clean-up work'
    whoCares << 'I wonder whether unplugging this cable will eliminate that
nasty sound.'
    critical << 'The server room goes on UPS!'
    whoCares << 'The sound has disappeared'
}
def select = select([critical, ordinary, whoCares])
println 'Starting to monitor our IT department'
sleep 3000
10.times {println "Received: ${select.prioritySelect().value}"}
```

8.4) Operators

Il modello di concorrenza basato su dataflow è costruito sull'immagine di un insieme di canali che connettono una certa quantità di operatori e selettori, questi due elementi consumeranno dei valori ricevuti dai canali di input, li trasformeranno in nuovi valori e li espelleranno tramite i canali di output.
Mentre un operatore attende che tutti i canali di input abbiano un valore disponibile prima di elaborarli, un selettore è attivato appena si renda disponibile un valore su uno qualsiasi dei canali di input.

```
operator(inputs: [a, b, c], outputs: [d]) {x, y, z ->
    …
    bindOutput 0, x + y + z
}
```

In questo esempio si costruisce una cache contenente copia di diversi siti.
La cache riceve in input degli indirizzi di pagine web e, se la pagina è presente in cache allora la versione contenuta in cache viene resa disponibile in output, se la pagina non è presente in cache si

effettua una richiesta di download.

```
operator(inputs: [urlRequests], outputs: [downloadRequests, sites]) {request ->
    if (!request.content) {
        println "[Cache] Retrieving ${request.site}"
        def content = cache[request.site]
        if (content) {
            println "[Cache] Found in cache"
            bindOutput 1, [site: request.site, word:request.word, content:
content]
        } else {
            def downloads = pendingDownloads[request.site]
            if (downloads != null) {
                println "[Cache] Awaiting download"
                downloads << request
            } else {
                pendingDownloads[request.site] = []
                println "[Cache] Asking for download"
                bindOutput 0, request
            }
        }
    } else {
        println "[Cache] Caching ${request.site}"
        cache[request.site] = request.content
        bindOutput 1, request
        def downloads = pendingDownloads[request.site]
        if (downloads != null) {
            for (downloadRequest in downloads) {
                println "[Cache] Waking up"
                bindOutput 1, [site: downloadRequest.site,
word:downloadRequest.word, content: request.content]
            }
            pendingDownloads.remove(request.site)
        }
    }
}
```

Il meccanismo standard per la gestione degli errori, attivato nel caso in cui sia lanciata un'eccezione non gestita durante l'esecuzione dell'operatore, stampa un messaggio nello standard error output e ferma l'operatore.
Se questo non è accettabile, è possibile definire un metodo reportError() all'interno dell'operatore che verrà invocato automaticamente nel caso in cui si verifichi un errore non gestito.

```
op.metaClass.reportError = {Throwable e ->
        //gestisce l'eccezione
        stop()  //è possibile terminare l'operatore manualmente
    }
```

Tipi di operatori
Esistono diversi tipi di operatori:

- operator: la versione di base.
- selector: operatore attivato dall'arrivo di un valore in uno qualsiasi dei suoi canali di input.
- prioritySelector: variante di selector che, in caso di disponibilità di più input, seleziona quello proveniente dal canale di indice minore.
- splitter: operatore ad input singolo che copia i dati ricevuti in input in tutti i canali di output a cui è collegato.

Reti di operatori
Gli operatori sono tipicamente collegati in reti in cui alcuni operatori consumano l'output di altri operatori.

```
operator(inputs:[a, b], outputs:[c, d]) {...}
splitter(c, [e, f])
selector(inputs:[e, d]: outputs:[]) {...}
```

In alternativa, è possibile riferirsi ai canali di output di un operatore attraverso il nome dell'operatore stesso:

```
def op1 = operator(inputs:[a, b], outputs:[c, d]) {…}

//prende in input il primo output di op1
def sp1 = splitter(op1.outputs[0], [e, f])

//prende in input il primo output di sp1 e il secondo output di op1
selector(inputs:[sp1.outputs[0], op1.outputs[1]]: outputs:[]) {…}
```

Raggruppamento di operatori

In GPars è possibile organizzare operatori in gruppi[68] con lo scopo di meglio calibrarne le prestazioni.
I gruppi forniscono un comodo factory method operator() allo scopo di creare operatori associati al gruppo.

```
import groovyx.gpars.group.DefaultPGroup
def group = new DefaultPGroup()
group.with {
    operator(inputs: [a, b, c], outputs: [d]) {x, y, z ->
        …
        bindOutput 0, x + y + z
    }
}
```

Il thread pool del gruppo di default è costituito da demoni, ciò richiede di assicurarsi di terminare esplicitamente il gruppo chiamando il metodo shutdown() nel caso in cui il thread pool del gruppo non sia composto da demoni altrimenti l'applicazione che si sta realizzando non terminerà correttamente.

Parallelizzazione di operatori

Generalmente un operatore viene eseguito da un singolo thread.
Per quanto questo caso si adatti alla stragrande maggioranza dei problemi da affrontare e permetta di codificare un operatore senza avere la necessità di renderlo thread safe, a volte può essere necessario permettere a più thread di eseguire contemporaneamente l'operatore; per esempio a causa di problemi di bilanciamento del carico di lavoro e nel caso in cui si preveda un accumulo notevole di dati da processare nei canali di input dell'operatore.
Introdurre la possibilità di eseguire un operatore concorrentemente implica doverlo rendere thread safe, proteggendo eventuali risorse che devono essere condivise tra i thread.

Per permettere a più thread di eseguire l'operatore concorrentemente, è sufficiente dichiarare un

68 Ovviamente ci si sta riferendo a gruppi paralleli, come ad esempio DefaultPGroup.

parametro maxForks al momento della creazione dell'operatore.

```
def op = operator(inputs: [a, b, c], outputs: [d, e], maxForks: 2) {x, y, z ->
    bindOutput 0, x + y + z
    bindOutput 1, x * y * z
}
```

Il valore del parametro maxForks indica il massimo numero di thread a cui è permessa l'esecuzione contemporanea dell'operatore.
Nel caso in cui l'operatore sia collegato ad un gruppo definito dall'utente, è necessario assicurarsi che il gruppo disponga di un numero di thread sufficiente per coprire tutti i fork richiesti.
Il thread group di default è in grado di ridimensionarsi automaticamente in caso sia necessario e, salvo situazioni patologiche, non si correrà mai il rischio di terminare i thread a disposizione.

Sincronizzazione dell'output
L'introduzione della possibilità di eseguire concorrentemente un operatore introduce un problema: possono verificarsi dei race-conditions, specie se gli operatori hanno più canali di output, perché la scrittura nei canali non avviene atomicamente.
Per ovviare a questi problemi è necessario usare un qualche tipo di sincronizzazione, implicita o esplicita.

```
operator(inputs:[inputChannel], outputs:[a, b], maxForks:5) {msg ->
    bindOutput 0, msg
    bindOutput 1, msg
}
inputChannel << 1
inputChannel << 2
inputChannel << 3
inputChannel << 4
inputChannel << 5
```

L'esecuzione di questo semplice operatore potrebbe risultare in output di questo tipo:

```
a -> 1, 3, 2, 4, 5
b -> 2, 1, 3, 5, 4
```

L'utilizzo di una sincronia esplicita è una delle possibili vie percorribili per la risoluzione di problemi di questo tipo: permette di scrivere atomicamente su tutti i canali di output e di proteggere lo stato non thread safe dell'operatore.

```
def lock = new Object()
operator(inputs:[inputChannel], outputs:[a, b], maxForks:5) {msg ->
    doStuffThatIsThreadSafe()
    synchronized(lock) {
        doSomethingThatMustNotBeAccessedByMultipleThreadsAtTheSameTime()
        bindOutput 0, msg
        bindOutput 1, 2*msg
    }
}
```

Ovviamente, limitare l'esecuzione di parte del codice di un operatore ad un singolo thread alla volta potrebbe limitare notevolmente l'efficacia dell'esecuzione dell'operatore da parte di diversi thread.
Esistono un paio di metodi che vengono in aiuto all'utente, si tratta del metodo *bindAllOutputsAtomically* che permette di scrivere in modo atomico un valore su tutti i canali di output dell'operatore, e del metodo *bindAllOutputValuesAtomically* che accetta una lista di parametri da scrivere nei canali di output dell'operatore.

Ogni parametro nella lista fornita al metodo *bindAllOutputValuesAtomically* verrà inviato sul canale di indice corrispondente alla posizione del parametro nella lista.

```
operator(inputs:[inputChannel], outputs:[a, b], maxForks:5) {msg ->
    doStuffThatIsThreadSafe()
        bindAllOutputValuesAtomically msg, 2*msg
    }
}
```

Terminazione di operatori
Gli operatori possono essere terminati in due modi:

1. chiamando il metodo terminate() su tutti gli operatori che devono essere terminati
2. mandando un poison message.

Vediamo un esempio del primo caso:

```
def op1 = operator(inputs: [a, b, c], outputs: [d, e]) {x, y, z -> }
def op2 = selector(inputs: [d], outputs: [f, out]) { }
def op3 = prioritySelector(inputs: [e, f], outputs: [b]) {value, index -> }

//terminazione di tutti gli operatori utilizzando il metodo termiante()
[op1, op2, op3]*.terminate()
op1.join()
op2.join()
op3.join()
```

Vediamo ora un esempio del secondo caso:

```
def op1 = operator(inputs: [a, b, c], outputs: [d, e]) {x, y, z -> }
def op2 = selector(inputs: [d], outputs: [f, out]) { }
def op3 = prioritySelector(inputs: [e, f], outputs: [b]) {value, index -> }
a << PoisonPill.instance  //Invio del veleno
op1.join()
op2.join()
op3.join()
```

Dopo aver ricevuto il veleno un operatore termina, si assicura solamente di inviare il veleno in tutti i suoi canali di output prima di terminare in modo che il veleno possa propagarsi a tutti gli operatori a cui è connesso.
Un operatore attende sempre che ogni suo input abbia un valore prima di procedere, l'unica eccezione è costituita dal messaggio PoisonPill: una volta ricevuto un messaggio PoisonPill su uno qualsiasi degli input, il messaggio viene inoltrato e l'operatore termina.
Bisogna prestare attenzione nella terminazione di parte di una dataflow network dall'esterno, indipendentemente dal metodo utilizzato perché, data la natura intrinseca delle computazioni che si stavano effettuando, è possibile che parte del lavoro non ancora completato venga perduto.

Terminiamo un operatore con gentilezza
L'unico momento in cui è possibile terminare un operatore avendo la certezza di non perdere dati si ha quando un operatore ha appena terminato la computazione relativa all'input corrente, ha inviato il risultato in output e si sta apprestando ad accettare un nuovo input.
Il metodo terminateAfterNextRun() termina un operatore esattamente in questo istante, schedula la terminazione dell'operatore non appena sia completata la gestione del prossimo messaggio.
Questo tipo di approccio è utilizzato per gestire situazioni di bilanciamento del carico di lavoro: dopo una sessione di lavoro estenuante, quando il carico di lavoro cala, il metodo

terminateAfterNextRun() può essere utilizzato per terminare le istanze di un operatore non più necessarie.

Veleno con contatore
Quando si invia un PoisonPill in una rete di operatori si può avere la necessità di conoscere quando tutti gli operatori o un certo numero di questi siano stati terminati.
La classe *CountingPoisonPill* serve esattamente a questo scopo.

```
operator(inputs: [a, b, c], outputs: [d, e]) {x, y, z -> }
selector(inputs: [d], outputs: [f, out]) { }
prioritySelector(inputs: [e, f], outputs: [b]) {value, index -> }
//Invio del veleno indicando quanti operatori devono essere terminati prima che
//sia possibile continuare
final pill = new CountingPoisonPill(3)
a << pill
//Attende la terminazione degli operatori
pill.join()
//In questo momento almeno 3 operatori sono stati terminati
```

La proprietà termination della classe *CountingPoisonPill* è di tipo *Promise<Boolean>* ciò implica che può essere utilizzata in quanto tale:

```
//Invio del veleno indicando quanti operatori devono essere terminati prima che
//sia possibile continuare
final pill = new CountingPoisonPill(3)
pill.termination.whenBound {println "Reporting asynchronously that the network
has been stopped"}
a << pill
if (pill.termination.bound) println "Wow, that was quick. We are done already!"
else println "Things are being slow today. The network is still running."
//Attende la terminazione degli operatori
assert pill.termination.get()
//In questo momento almeno 3 operatori sono stati terminati
```

Un esempio un po' più complesso
Si ricorda che in GPars un task è associato ad una variabile dataflow che viene inizializzata ad un valore non appena il task termina.
L'operatore “terminator” sfrutta il fatto che DataflowVariable è un'implementazione dell'interfaccia DataflowReadChannel e che può quindi essere consumata da un operatore.
Quando entrambi i produttori terminano, l'operatore “terminator” sfrutta questa meccanica per inviare una PoisonPill nel canale q per terminare il consumatore non appena abbia consumato tutti i dati.

```
import groovyx.gpars.dataflow.DataflowQueue
import groovyx.gpars.group.NonDaemonPGroup

def group = new NonDaemonPGroup()
final DataflowQueue q = new DataflowQueue()
//Destinazione finale
def customs = group.operator(inputs: [q], outputs: []) { value ->
    println "Customs received $value"
}
//Grosso produttore
def green = group.task {
    (1..100).each {
        q << 'green channel ' + it
```

```
        sleep 10
    }
}
//Piccolo produttore
def red = group.task {
    (1..10).each {
        q << 'red channel ' + it
        sleep 15
    }
}
def terminator = group.operator(inputs: [green, red], outputs: []) { t1, t2 ->
    q << PoisonPill.instance
}
customs.join()
group.shutdown()
```

Selettori

Un selettore dovrebbe essere costituito da una chiusura che consumi uno o due argomenti

```
selector (inputs : [a, b, c], outputs : [d, e]) {value ->
    ....
}
```

La versione con due argomenti riceve come parametri il valore letto e l'indice del canale da cui è stato letto, questo permette al selettore di distinguere tra dati provenienti da canali di input diversi.

```
selector (inputs : [a, b, c], outputs : [d, e]) {value, index ->
    ....
}
```

Selettore prioritario
Quando sia necessario tenere conto di priorità esistenti tra i canali di input, bisognerebbe utilizzare un DataflowPrioritySelector.

```
prioritySelector(inputs : [a, b, c], outputs : [d, e]) {value, index ->
    …
}
```

Il selettore prioritario darà sempre priorità a valori provenienti da canali con l'indice più basso.

Selettore Join
Si tratta di un selettore in cui non è presente una chiusura che ne specifichi il comportamento, un tale selettore si limita a copiare in tutti i canali di uscita tutti i dati ricevuti dai canali di ingresso.

```
def join = selector (inputs : [programmers, analysis, managers], outputs :
[employees, colleagues])
```

Parallelismo
L'attributo maxForks permette l'esecuzione concorrente di un selettore.

```
selector (inputs : [a, b, c], outputs : [d, e], maxForks : 5) {value ->
    ....
}
```

Guardie
Così come avviene per Select, anche in Selector (si ricorda che Selector è un operatore) è possibile includere o escludere temporaneamente dei canali di input da una selezione.
La proprietà *guards*, dichiarata al momento della creazione dell'operatore, può essere utilizzata per impostare la maschera iniziale da utilizzare sui canali di input.
La maschera può essere modificata poi, durante l'esecuzione dell'operatore, mediante l'utilizzo dei metodi *setGuards* e *setGuard*, come è possibile vedere nell'esempio.

```
import groovyx.gpars.dataflow.DataflowQueue
import static groovyx.gpars.dataflow.Dataflow.selector
import static groovyx.gpars.dataflow.Dataflow.task

final DataflowQueue operations = new DataflowQueue()
final DataflowQueue numbers = new DataflowQueue()
def instruction
def nums = []
selector(inputs: [operations, numbers], outputs: [], guards: [true, false])
{value, index ->   //impostazione iniziale della maschera
    if (index == 0) {
        instruction = value
        setGuard(0, false)  //modifica della maschera utilizzando setGuard()
        setGuard(1, true)
    }
    else nums << value
    if (nums.size() == 2) {
        //modifica della maschera utilizzando setGuards()
        setGuards([true, false])
        final def formula = "${nums[0]} $instruction ${nums[1]}"
        println "$formula = ${new GroovyShell().evaluate(formula)}"
        nums.clear()
    }
}
task {
    operations << '+'
    operations << '+'
    operations << '*'
}
task {
    numbers << 10
    numbers << 20
    numbers << 30
    numbers << 40
    numbers << 50
    numbers << 60
}
```

Si eviti l'utilizzo di guardie in selettori con maxForks impostato ad un valore maggiore di uno; per quanto il selettore sia thread safe e non sarà danneggiato in alcun modo, le guardie non saranno probabilmente impostate in modo prevedibile.

8.5) Implementazione – legame con gli attori

In GPars, la concorrenza basata su dataflow si basa sui stessi principi implementativi della concorrenza basata sugli attori.
Tutti i dataflow task condividono un thread pool cosicché il numero di thread creati con il factory method Dataflow.task() non deve necessariamente corrispondere al numero di thread fisici richiesti al sistema.

Il factory method Pgroup.task() può essere utilizzato per creare un task associato ad un gruppo; dato che ogni gruppo definisce il proprio thread pool, è possibile organizzare facilmente i task in thread pool diversi esattamente come è possibile per gli attori.

Composizione di dataflow e attori

Dataflow e attori possono essere composti per ottenere la soluzione ad ogni problema che si stia affrontando.
É possibile utilizzare liberamente le variabili dataflow all'interno degli attori.

```
final DataflowVariable a = new DataflowVariable()
final Actor doubler = Actors.actor {
    react {message->
        a << 2 * message
    }
}
final Actor fakingDoubler = actor {
    react {
        doubler.send it  //invia un messaggio a doubler
        //attendi che il risultato venga associato ad 'a'
        println "Result ${a.val}"
    }
}
fakingDoubler << 10
```

Nell'esempio è possibile vedere come fakingDoubler utilizzi messaggi e una variabile dataflow per comunicare con l'attore doubler.

Utilizzo diretto dei dataflow nei thread di Java

Istanze delle classi DataflowVariable e DataflowQueue possono ovviamente essere utilizzate in ogni thread dell'applicazione che si sta creando, non solamente nei task creati utilizzando Dataflow.task().

Nel seguente esempio sono stati creati due thread che scambiano dati utilizzando due variabili dataflow.

```
import groovyx.gpars.dataflow.DataflowVariable
final DataflowVariable a = new DataflowVariable<String>()
final DataflowVariable b = new DataflowVariable<String>()
Thread.start {
    println "Received: $a.val"
    Thread.sleep 2000
    b << 'Thank you'
}
Thread.start {
    Thread.sleep 2000
    a << 'An important message from the second thread'
    println "Reply: $b.val"
}
```

Ovviamente, così facendo si perdono molte delle funzionalità tipiche degli attori.

Conclusioni

Come spesso avviene quando ci si trova ad affrontare una problematica, anche nell'affrontare la concorrenza esistono due tipi di approcci: si possono prevenire problemi relativi all'accesso concorrente a risorse condivise nascondendole, proteggendole o, in generale serializzandone l'accesso, oppure si può lasciare piena libertà di accesso e modifica concorrente alle risorse condivise preoccupandosi di gestire poi i conflitti che immancabilmente verranno a verificarsi.

Ultimamente, per spiegare la potenza di un sistema concorrente a chi ha una certa, per quanto limitata, dimestichezza con l'ambiente informatico, sono solito portare questo esempio.
Considerate una rete di calcolatori: si tratta di macchine, in esecuzione contemporanea, che cercano di scambiarsi dati attraverso un qualche tipo di mezzo trasmissivo condiviso; ogni macchina ignora lo stato delle altre macchine e non è in grado di prevedere quando tenteranno una trasmissione nel mezzo.

Come evitare, quindi, la sovrapposizione dei messaggi ?
Una macchina che abbia la necessità di inviare un messaggio potrebbe mettersi in ascolto sul mezzo condiviso per verificarne la disponibilità ma, per quanto deve rimanere in ascolto ?
Come può avere la certezza che un'altra macchina non tenti, a sua volta, una trasmissione in contemporanea trovando il mezzo libero?
Questi, e altri problemi, portarono alla creazione di reti tipo token ring: si tratta di reti in cui un token viene assegnato ciclicamente alle macchine connesse alla rete che potranno trasmettere nel mezzo per un tempo limitato solo ed esclusivamente quando entreranno in possesso del token.
Per quanto questo, e altri simili approcci funzionino, comportano uno spreco non indifferente di risorse, specialmente nel caso di scarso traffico nel mezzo o di presenza in rete di macchine molto più loquaci delle altre, e comportano l'introduzione di una rigidità artificiale nella struttura della rete.

Ora, qual'è la struttura di rete in assoluto più diffusa al giorno d'oggi ?
Ethernet, nelle sue varie incarnazioni.
Cos'ha di particolare quell'architettura di rete da permetterle di soppiantare la stragrande maggioranza delle altre architetture esistenti, al giorno d'oggi relegate in nicchie specifiche ?

Non implementa praticamente alcun tipo di vincolo preventivo (per lo meno non rigido come nel caso del token ring) per la trasmissione nel mezzo trasmissivo condiviso.
Praticamente solo questo fatto ha permesso al protocollo ethernet di venire applicato praticamente ovunque si abbia la necessità di trasmettere dei dati utilizzando un mezzo trasmissivo condiviso.

Torniamo ad un sistema concorrente; in un sistema concorrente si possono seguire due approcci per la gestione dell'accesso e della modifica di una risorsa condivisa: si può incapsulare la risorsa in uno strumento che ne serializzi gli accessi e questa parte, nelle sue varie incarnazioni, è almeno parzialmente trattata in questo documento, oppure si può lasciare piena libertà di accesso e modifica concorrente ad una risorsa condivisa avendo però cura di gestire poi gli eventuali conflitti dovuti a due o più processi che modifichino contemporaneamente la risorsa.
Quale dei due approcci risulterà il vincitore ?
Il secondo è ancora nella sua infanzia, in GPars è supportato con l'introduzione del supporto a STM (Software Transactional Memory).

La strada che al momento sembra seguire l'evoluzione dei sistemi paralleli sta portando allo sviluppo di applicazioni dalla granularità sempre più fine, in cui una moltitudine di componenti in

sé sempre più semplici collaborano concorrentemente alla risoluzione di problemi sempre più complessi.
Come si può intuire, per quanto non sia sempre possibile evitarlo, costringere questa moltitudine di elementi all'accesso sequenziale ad una singola risorsa condivisa rappresenta un notevole ostacolo per quello che riguarda l'efficienza prestazionale ottenibile.

Lo scopo ultimo di questo documento, però, non è quello di disquisire sulla futura evoluzione dei sistemi paralleli ma è quello di mostrare come sia possibile, con uno sforzo relativamente ridotto, introdurre la concorrenza nei sistemi esistenti e come sia possibile sviluppare delle applicazioni concorrenti senza essere dei massimi esperti in materia.

Allo scopo di introdurre tutto questo in un ambiente che, per quanto possa sembrare assurdo, è resistente al cambiamento.

Bibliografia

1: Dierk Koenig, Andrew Glover, Paul King, Guillaume Laforge, Jon Skeet, "Groovy in Action", ISBN: 978-1932394849
2: "Scala as the long term replacement for java/javac", http://macstrac.blogspot.com/2009/04/scala-as-long-term-replacement-for.html, gennaio 2012
3: "Multiple dispatch", http://en.wikipedia.org/wiki/Multiple_dispatch, gennaio 2012
4: "Currying", http://en.wikipedia.org/wiki/Currying, gennaio 2012
5: "Partial function application", http://en.wikipedia.org/wiki/Partial_application, gennaio 2012
6: "Tail call", http://en.wikipedia.org/wiki/Tail_call, gennaio 2012
7: "Evaluation strategy", http://en.wikipedia.org/wiki/Evaluation_strategy, gennaio 2012
8: "Flatland", http://en.wikipedia.org/wiki/Flatland, gennaio 2012
9: "Things you can do but better leave undone", http://groovy.codehaus.org/Things+you+can+do+but+better+leave+undone, gennaio 2012
10: "Duck typing", http://en.wikipedia.org/wiki/Duck_typing, gennaio 2012
11: "Variable interpolation", http://en.wikipedia.org/wiki/Variable_interpolation, gennaio 2012
12: "Plain Old Java Object", http://en.wikipedia.org/wiki/Plain_Old_Java_Object, gennaio 2012
13: "Groovy", http://groovy.codehaus.org/, gennaio 2012
14: Mohamed Seifeddine, "Introduction to Groovy and Grails", November 6, 2009
15: "Strings and GString", http://groovy.codehaus.org/Strings+and+GString, gennaio 2012
16: "GPars", http://gpars.codehaus.org/, gennaio 2012
17: "Class Collection", http://groovy.codehaus.org/groovy-jdk/java/util/Collection.html, gennaio 2012
18: "Domain-specific language", http://en.wikipedia.org/wiki/Domain-specific_language, gennaio 2012
19: "Eventually Consistent - Revisited", http://www.allthingsdistributed.com/2008/12/eventually_consistent.html, gennaio 2012
20: "Sir Charles Antony Richard Hoare", http://en.wikipedia.org/wiki/C._A._R._Hoare, gennaio 2012
21: "JCSP library", http://www.cs.kent.ac.uk/projects/ofa/jcsp/, gennaio 2012
22: "Jon Kerridge", http://www.soc.napier.ac.uk/~cs10/, http://www.iidi.napier.ac.uk/c/people/peopleid/51, gennaio 2012
23: "Dining philosophers problem", http://en.wikipedia.org/wiki/Dining_philosophers_problem#Resource_hierarchy_solution, gennaio 2012
24: "Per Brinch Hansen", http://en.wikipedia.org/wiki/Per_Brinch_Hansen, gennaio 2012
25: "Continuation-passing style", http://en.wikipedia.org/wiki/Continuation-passing_style, gennaio 2012
26: "The Javaflow Component", http://commons.apache.org/sandbox/javaflow/, gennaio 2012
27: "Esempio", http://git.codehaus.org/gitweb.cgi?p=gpars.git;a=blob_plain;f=src/test/groovy/groovyx/gpars/samples/actors/stateful/DemoMultiMessage.groovy;hb=HEAD, gennaio 2012
28: "Factory method pattern", http://en.wikipedia.org/wiki/Factory_method_pattern, gennaio 2012
29: "Replace Constructor with Factory Method", http://sourcemaking.com/refactoring/replace-constructor-with-factory-method, gennaio 2012
30: "Precedence graph", http://en.wikipedia.org/wiki/Precedence_graph, gennaio 2012

31: "Serializability", http://en.wikipedia.org/wiki/Serializability, gennaio 2012
32: "Dataflows", http://gpars.org/SNAPSHOT/groovydoc/groovyx/gpars/dataflow/Dataflows.html, gennaio 2012
33: "Sieve of Eratosthenes", http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes, gennaio 2012
34: "Activity diagram", http://en.wikipedia.org/wiki/Activity_diagram, gennaio 2012