Manual de uso para programación de módulos del protocolo DCNP en Java

Pablo Martínez Sánchez

2 de agosto de 2017

Índice

1. Introducción **2**

2. Antes de comenzar **2**

3. Implementando los métodos de la librería **3**
3.1. Módulo problem 3
3.2. Módulo solver 5

4. Ejemplos **6**

5. Recomendaciones **9**

6. Cómo probar y ejecutar módulos **9**

1. Introducción

Este manual describe el proceso de adaptación de un código cualquiera en el lenguaje de programación Java al protocolo *DCNP* .Dicho protocolo permite distribuir el trabajo a realizar entre tantos ordenadores como se desee, mejorando sensiblemente el tiempo de ejecución del mismo. Todo el código y documentación del protocolo se encuentra en github *DCNP* , aunque se referenciará más concretamente conforme vaya haciendo falta.

Para ello, será necesario adaptar el código al protocolo, usando la librería Java para *DCNP* . Es decir, es posible(y recomendable) programar primero aquello que se desea portar al protocolo de forma separada y, una vez comprobada la corrección del mismo, adaptarse, en cuyo caso conviene saber que para ello será muy aconsejable que el código cumpla una serie de reglas, sin las cuales su conversión será realmente complicado(y posiblemente, imposible). Este manual esta orientado y elaborado con el *IDE Eclipse*, pero puede ser válido en caso de usar cualquier otra herramienta.

2. Antes de comenzar

Para programar y compilar cualquier módulo para *DCNP* , es necesario tener importado la librería Java para *DCNP* , que puede descargarse desde el enlace(*github.com*). Para importar la librería en *Eclipse*, debemos seleccionar el proyecto donde queramos incluirla e ir a: *Proyecto* $\rightarrow$ *Propiedades* $\rightarrow$ *Java Build Path* $\rightarrow$ *Anadir JARs externos* y por último, *Aplicar*.

Es altamente recomendable disponer del código que se quiere portar, en vez de intentar hacerlo directamente cuando se implemente lo necesario para hacerlo funcionar en el protocolo. Puesto que el protocolo distribuirá el trabajo, dicho código debe de cumplir las condiciones necesarias para ser paralelizable, de lo contrario, será muy difícil adaptarlo al protocolo. También es altamente recomendable leer con cautela el apartado de recomendaciones de este documento.

3. Implementando los métodos de la librería

Para una mejor comprensión del funcionamiento del protocolo, es recomendable ver el documento explicativo del mismo, aunque este manual explica las partes estrictamente necesarias lo suficientemente en detalle como para poder programar los módulos correctamente;

El funcionamiento del protocolo es el siguiente: Hay un servidor al que se conectan los clientes que quieren participar en el cálculo. Si hay N clientes conectados, el servidor repartirá el trabajo entre esos N clientes. Para ello, el servidor se comunica con el módulo *problem* y cada cliente, con el módulo *solver* (cada cliente se comunica con el suyo, de forma independiente). Esto quiere decir que el protocolo es totalmente modular; el protocolo es independiente de los módulos. Dichos módulos deberán compilarse como *JAR ejecutable* y proporcionárselos al servidor.

Es necesario implementar dos clases(módulos) independientes. Cada una llevará a cabo una función ,implementará métodos de la librería y dispondrá de distintos métodos auxiliares; Por una parte, está el módulo *problem* , cuya función principal es proveer de entradas y tratar o procesar salidas. Por otro lado, está el módulo *solver* , cuya función es, mediante las entradas proporcionadas, devolver las salidas(dicho intercambio no es responsabilidad del programador). El *problem* debe estar preparado para situaciones como la de enviar varias entradas sin haber recibido ninguna salida, como se verá más adelante. Las entradas y salidas, que se intercambiarán entre ambos módulos, irán en el tipo de datos *String* , pudiendo el programador modificar su tipo libremente, después de haber recibido las entradas y salidas como argumentos de los métodos. A continuación se detalla cómo implementar los dos módulos.

3.1. Módulo problem

Primero, crea una clase con el método

```
public static void main(String[] args)
```

en el que se cree una clase nueva(que será la que implemente la librería) y lánzala como un hilo;

```
package ejemplo;

public class Problem {

  public static void main(String[] args) {

    EjemploProblem e = new EjemploProblem();
    e.start();

    try {
```

```
            e.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

}
```

Si fuera necesario datos de entrada para hacer funcionar el programa, sería necesario pasarlos como argumentos del *main*, y de ahí pasarlos a la clase *Ejemplo*, que es la que vamos a ver ahora, que implementará los métodos y que hará realmente el trabajo. Dentro de dicha clase, para importar la librería, es necesario usar *import*;

```
import dcnp.Problem;
```

y ahora, en la declaración de la clase, haz que la clase ejemplo herede de la clase *Problem*;

```
public class EjemploProblem extends Problem {
...
}
```

Ahora es el momento de implementar los métodos obligatorios de la clase *Problem*, de los que se detallan cuál es su objetivo, qué deben hacer y cómo hacerlo, a continuación:

- **String nextIn()**: Devuelve una nueva entrada para ser calculada. Es importante recalcar que debe ser una entrada nueva.
- **void newOut(String out, String in**: Recibe una nueva salida, representada por el argumento *out* del método(cada programador deberá evaluar las necesidades de lo que esté haciendo, y hacer con la salida lo que necesite) El argumento *in* es la entrada que devolvió anteriormente el método *nextIn()*, pudiendo así saber, para las entradas que se enviaron, cuales fueron sus correspondientes salidas.
- **boolean isSolution(String output)**: Devuelve *true* si, en el momento de la llamada del método, se ha encontrado la solución, *false* si no. Como argumento se pasa una salida que acaba de ser recibida. De esta forma, es posible que sea necesario evaluar dicha salida para comprobar si se ha encontrado solución, o disponer de estructuras de datos para dicho objetivo y obviar el argumento que se recibe. Si se dispone de varias soluciones, este método sólo debe devolver *true* cuando se hayan encontrado todas ellas, o cuando no se vayan a encontrar más. Es decir, cuando el método devuelva *true* , significará que el cálculo ha finalizado, y todos los nodos conectados terminarán.
- **String solutionMessage()**: Cuando *isSolution* devuelva *true* , se imprimirá por pantalla la solución, en el servidor. Dicho mensaje de solución es el que debe devolver, en un *String* , dicho método. Esto quiere decir que el mensaje de solución es dependiente de la implementación que se haga en este método. En el caso de

haber diversas soluciones, una posible aproximación sería almacenarlas y, en este método, devolver *String* en el que estén todas ellas.

Es importante destacar que los argumentos que se proporcionan en los métodos no son de uso obligatorio; Se proporcionan por si el programador los necesitara, pero no es necesario que sean usados. Además de estos, se proporcionan otros métodos auxiliares;

- **void print(String msg)**: Imprime en el servidor el mensaje que se le pasa como argumento(útil para *debug*)
- **String INVALID_INPUT()**: Devuelve la constante de *entrada inválida*. Útil en el método *nextIn()* cuando debes devolver una entrada, pero no hay más disponibles por cualquier motivo. En el caso de que un nodo reciba dicha entrada, se desconectará del servidor(esto puede suceder cuando dicho nodo ha terminado, pero otros aún están calculando otras entradas, porque necesitan más tiempo para ello)

Para ver un ejemplo de un *problem* real, échale un vistazo al apartado Ejemplos

3.2. Módulo solver

Para esta implementación, se sigue la misma filosofía de crear una clase estática que lance el hilo de una clase que implementará los métodos del *solver* . Dicho módulo es, en realidad, el que va a hacer el trabajo. Se ejecutará en cada cliente de forma separada y,hará los cálculos necesarios. La clase debe ser algo así;

```
import dcnp.Solver;

public class EjemploSolver extends Solver {
...
}
```

Sólo es obligatorio implementar el siguiente método;

- **String getOut(String input)**: Para la entrada denotada por *input*, devuelve otro string, que corresponde con su salida.

Y como métodos auxiliares;

- **void print(String msg)**: Imprime en el servidor el mensaje que se le pasa como argumento(útil para *debug*)

Si bien es cierto que, al repartir el trabajo entre varios nodos, el tiempo de ejecución se reduce notablemente, es importante destacar que dicha mejora será mucho mayor si la ejecución de éste módulo es concurrente. En un ideal, debería ser así, para que se aprovechara al máximo la capacidad de cómputo de todos los nodos. En definitiva, el hecho de que este protocolo funcione concurrentemente es una tarea que debe llevar a cabo el programador de éste módulo.

Para ver un ejemplo de un *solver* real, échale un vistazo al apartado Ejemplos

4. Ejemplos

A continuación se incluye el código de un ejemplo que implementa los métodos de la librería. Su finalidad es encontrar el número definido en la variable *FIND_NUM* del *problem* . El módulo *solver* espera un tiempo aleatorio, que es mayor cuantos más núcleos tiene el procesador donde se ejecuta, para dar la falsa sensación de que se está haciendo un cálculo real(este ejemplo esta hecho solo para mostrar cómo funcionaría el protocolo en condiciones reales). El código se encuentra también en github.

```
//Problem.java
package app;

import problem.*;

public class Problem {

  public static void main(String[] args) {

    Numeros problem = new Numeros();
    problem.start();

    try {
      problem.join();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }

  }

}


//Solver.java
package app;

import solver.*;

public class Solver {

  public static void main(String[] args) {

    Numeros solver = new Numeros();
    solver.start();

    try {
      solver.join();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
```

```
    }

}

//Numeros.java ==> Modulo Problem
package problem;

import dcnp.Problem;

public class Numeros extends Problem {

    private final int FIND_NUM = 50;
    private int num;

    public Numeros() {
        this.num = 0;
    }

    @Override
    public boolean isSolution(String input) {
        return Integer.parseInt(input) == FIND_NUM;
    }

    @Override
    protected String nextIn() {
        this.num++;
        return String.valueOf(this.num-1);
    }

    @Override
    protected void newOut(String out, String input) {

    }

    @Override
    protected String solutionMessage() {
        return "The number was found: " + this.FIND_NUM;
    }

}

//Numeros.java ==> Modulo solver
package solver;

import dcnp.Solver;

public class Numeros extends Solver {
```

```
  public Numeros() {
  }

  protected String getOut(String input) {
    try {
      Thread.sleep((long) ((Math.random()*4000)/
      Runtime.getRuntime().availableProcessors()));
    } catch (InterruptedException e) {
      this.print("Thread interrupted");
    }
    return String.valueOf(Integer.parseInt(input)+1);
  }

}
```

Si bien es cierto que esto es sólo un ejemplo, en github se muestra un ejemplo real, en el que se calcula la solución al famoso *problema de las N-Reinas*, en el que se calcula de cuántas formas pueden colocarse N reinas en un tablero de tamaño NxN. Dicho ejemplo es concurrente y aprovecha dinámicamente los todos los núcleos del nodo en el que se ejecute.

5. Recomendaciones

El protocolo es limitado en ciertos aspectos, el *debug* puede ser complejo y es sencillo cometer ciertos errores si no se tiene precaución;

- **Importante:No hacer uso *System.out.print* ni *System.err.print* ni *System.in***: Usar cualquiera de ellos provocaría un comportamiento incorrecto, por lo que su uso está completamente restringido(dentro de los módulos). Si fuera necesario imprimir por pantalla, usa el método de la libería *print(msg)*. Si necesitas información de entrada, hazlo mediante el archivo *dcnp.txt*, como se explica a continuación, en el apartado cómo probar y ejecutar módulos.

- **Tratar todas las excepciones**: Sobre todo aquellas excepciones *runtime*. Por ejemplo, si se trabaja con enteros, como es el caso del ejemplo, es necesario pasar de *String* a entero, o a *float*, lo que podría fallar en tiempo de ejecución si no se hace correctamente. Una vez sabemos que se hace correctamente, no hay problema en eliminar el *catch*, pero al principio pueden ser de mucha utilidad.

- **Utilizar el método *print(msg)***: Para *debug* puede resultar múy util, y al igual que tratar las excepciones, cuando sabemos que va todo bien, podemos eliminarlo.

- **Asegurarse de que se especifican los ficheros correctamente**: Si aparecen muchos errores nada más empezar el cálculo por el cliente y el servidor, es probable que no se hayan especificado los ejecutables(en el fichero *dcnp.txt*) adecuadamente. Por ejemplo, que en el fichero se diga que el módulo *solver* es el fichero *fich_solver.jar* y en realidad ese sea el *problem* y esté al revés. En ese caso, el protocolo fallará al intentar comunicarse con los módulos.

- **Portar el programa previamente testeado**: Lo mejor es programar aquello que quiera hacerse y, una vez se sepa que funciona bien, pasarlo al protocolo *DCNP* . Intentar hacerlo directamente es muy difícil por las escasas formas que tiene el protocolo para testear el funcionamiento de los módulos

- **Asegurarse de que la librería esta incluida en el proyecto *eclipse***: Podría suceder(por experiencia) que *eclipse* permitiera compilar/exportar el *JAR* sin tener incluida la librería dentro del proyecto, por lo que conviene asegurarse de que la librería esta correctamente incluida. Si no fuera así, el comportamiento del protocolo sería incorrecto.

6. Cómo probar y ejecutar módulos

Primero, compilar desde el código en github, o bien descargar los ficheros *JAR* ya compilados(cliente y servidor). Después, compilar el módulo *solver* y *problem* . Con estos 4 ficheros ya podemos probar el protocolo. El *JAR* cliente debe estar en cada cliente y los otros tres, en el servidor, que necesita ejecutarse pasándole la carpeta donde se encuentran los siguientes elementos:

- fichero1
- fichero2
- dcnp.txt

Los nombres de los ficheros es libre, excepto el del fichero *dcnp.txt*. Dicho fichero de texto, debe seguir el formato especificado. Cada campo debe estar en una línea diferente, y en una misma línea no puede haber más de un campo. El nombre del campo debe estar seguido del carácter igual('=') y, a después, su valor. A continuación se especifican los posibles campos, siendo los campos en cursiva opcionales y en negrita, obligatorios:

- **problem**: Nombre del fichero del ejecutable *problem* (debe estar dentro del directorio)
- *problem_args*: Argumentos para pasar al *problem*
- **solver**: Nombre del fichero del ejecutable *solver* (debe estar dentro del directorio)
- *solver_args*: Argumentos para pasar al *solver*
- **name**: Nombre del problema a resolver.

Dicho directorio será proporcionado y será leído en el servidor, y será el quien gestione todo lo necesario para poner en marcha el protocolo. Un ejemplo de *dcnp.txt*:

```
name = Numeros
problem = problem_numeros.jar
solver = solver_numeros.jar
```

Dicho fichero de texto corresponde al del ejemplo, que no recibe argumentos, pero sería posible especificarlos con las opciones indicadas más arriba. Ejecutamos en servidor y cliente con:

```
java -jar Conductor.jar ruta/a_la_carpeta
java -jar Node.jar -ip IP_SERVIDOR -folder CARPETA
```

La carpeta del *Node*(cliente) solo especifica dónde se descargará el módulo *solver* que será enviado por el servidor. Para empezar el cálculo, es necesario introducir el comando *start* en el servidor(el comando *help* muestra los comandos disponibles). Dicho comando provocará que cualquier nodo conectado empiece el cálculo.