DCNP
Distributed Computing
Network Protocol

Versión 1.9

Pablo Martínez Sánchez

2 de agosto de 2017

Índice

1. Introducción **2**

2. Elementos del protocolo **3**

3. Diseño del protocolo **4**
3.1. Conductor - Problem 4
3.2. Node - Solver 6
3.3. Node - Conductor 7

4. Módulos del proyecto **10**

5. Servicios del protocolo **11**
5.1. Reparto justo del trabajo 11
5.2. Eficiencia del protocolo 11
5.3. Tolerancia a errores 11

6. Problemas **12**

7. Mejoras **12**

1. Introducción

La finalidad del protocolo *DCNP* es la de dar soporte genérico a la resolución de problemas en un sistema distribuido. Es decir, una solución genérica que puede adaptarse a cualquier problema que pueda resolverse de forma distribuida. *DCNP* proporciona un mecanismo para conseguir esto, basado en un modelo cliente-servidor.

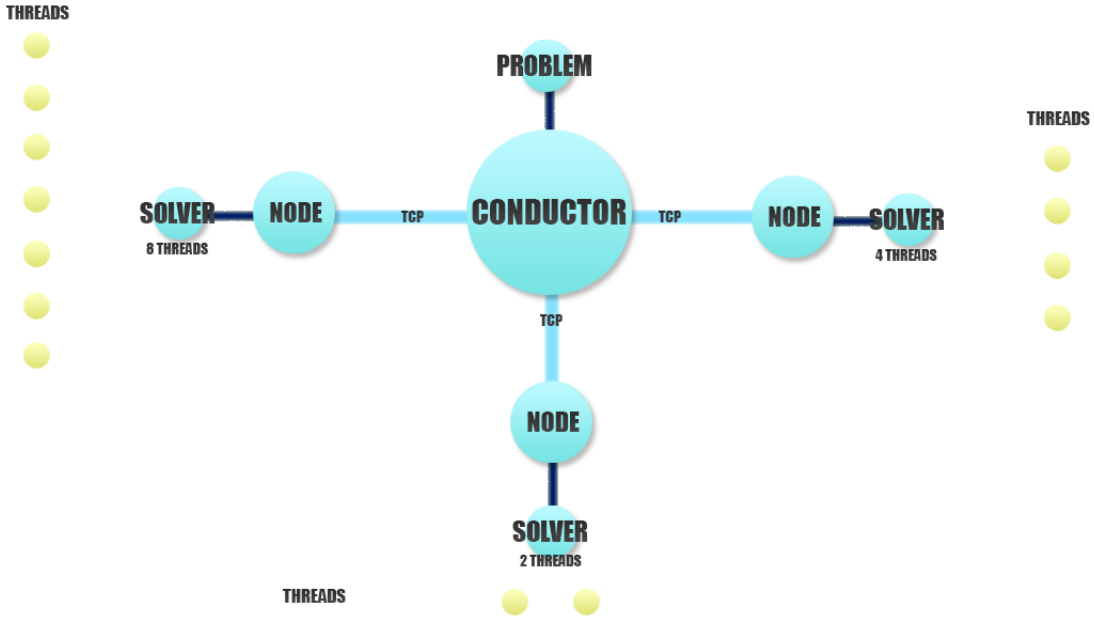


Figura 1: Esquema general del protocolo

El servidor, o *conductor* registrará todos los clientes(*nodes*), que se conectarán a él, esperando la orden de comienzo, que será enviada desde el *conductor* cuando el usuario así lo indique. Ambos están en comunicación con un módulo, que dependerá del problema, y cuya implementación es ajena al protocolo. El *conductor* se comunicará con el *problem*, de forma que éste último le proporcionará las entradas del problema que debe resolverse. El *conductor* enviará las entradas a los *nodes*, que a su vez se comunicarán con los *solver*, que proporcionarán la salida correspondiente, que se enviará de vuelta al *conductor*, hasta que todo el cómputo haya terminado y se haya encontrado una solución. Los módulos(*solver* y *problem*)son independientes del protocolo y tienen una serie de obligaciones y responsabilidades que se detallan más adelante.

2. Elementos del protocolo

Al ser un protocolo modular, consta de dos partes: Una de ellas es fija y común a cualquier problema que se quiera resolver y la otra, se acopla y desacopla del protocolo. La parte fija la componen los clientes y el servidor (*nodes y solver*), mientras que la parte que no es fija, la componen los módulos(*solver y problem*)cuya implementación es independiente a este protocolo. Dichos módulos definen el problema que se está resolviendo, de forma que, para desarrollar un nuevo problema, es necesario crear ambos módulos, adaptándolos a las necesidades del protocolo. Existen por lo tanto cuatro comunicaciones diferentes que se tienen que llevar a cabo:

- **Node - Conductor**: El servidor se comunica con cada uno de los nodos que participan en el cómputo del problema. Dicha comunicación se lleva a través de *sockets TCP*.
- **Conductor - Problem**: El servidor recibe entradas del *problem* y las reparte por los distintos *nodes*, que devolverán salidas, las cuales serán entregadas de vuelta al *problem* por el *conductor*. Dicha comunicación se realiza en memoria, sin ningún tipo de uso de la red. Se ejecuta el módulo *problem* y se abre una comunicación con el desde el *conductor*
- **Node - Solver**: Cada *node* recibirá las entradas desde el *conductor* y las entregará al módulo *solver*, que respondrá con la salida, que deberá entregarse al *conductor* por medio del *node*. Esta comunicación se realiza de igual manera que la del *conductor-problem*

3. Diseño del protocolo

Para cada comunicación, se detallan tanto los mensajes, como el autómata que define el comportamiento del intercambio de los mensajes. Los mensajes se representan como una secuencia de *bytes*, divididos en campos según convenga.

3.1. Conductor - Problem

- **NEXT_IN_REQ**(*Conductor* → *Problem*)
  El *conductor* enviará este mensaje al problem, pidiéndole la siguiente entrada. El *problem* sólo debe enviar una entrada, sin preocuparse de a qué *node* va a ir dirigida. Dicha consideración deberá llevarla a cabo el propio *conductor* . El mensaje se compone de los siguientes campos:

  - OpCode (1 byte): Código de operación

OpCode(1B)

- **NEXT_IN**(*Problem* → *Conductor*)
  Envía la siguiente entrada, cuyo formato y significado es totalmente indiferente para el protocolo, siendo algo que deben gestionar los módulos, siendo el protocolo sólo responsable de repartir el flujo de información de entradas y salidas. El mensaje se compone de los siguientes campos:

  - OpCode (1 byte): Código de operación
  - BytesOfField (4 bytes): Entero representando cuál es la longitud en *bytes* del siguiente campo
  - In(¿? bytes): Entrada, en *ASCII*, tal y como ha sido generada en el *problem* .

OpCode(1B)	BytesOfField(4B)	In (¿?)

- **NEW_OUT**(*Conductor* → *Problem*)
  El *conductor* recibe una nueva salida desde algún *node* y la envía mediante este mensaje al *problem* El mensaje se compone de los siguientes campos:

  - OpCode (1 byte): Código de operación
  - BytesOfField (4 bytes): Entero representando cuál es la longitud en *bytes* del siguiente campo
  - Output(¿? bytes): La salida, en *ASCII*

OpCode(1B)	BytesOfField(4B)	Output (¿?)

- **SOLUTION**(*Problem* → *Conductor*)
  En algún momento, el *problem* recibirá un mensaje *NewOut*, que será la última salida necesaria para encontrar la solución al problema que se está computando. El *problem* lo detectará, y usará este mensaje para enviar la solución, en texto, al *conductor* , que actuará en consecuencia(terminando la ejecución de todo el protocolo, mostrando por pantalla la solución). El mensaje se compone de los siguientes campos:

  - OpCode (1 byte): Código de operación
  - BytesOfField (4 bytes): Entero representando cuál es la longitud en *bytes* del siguiente campo
  - Solution(¿? bytes): La solución, en *ASCII*, que deberá mostrarse por pantalla directamente mediante el *conductor* .

OpCode(1B)	BytesOfField(4B)	Solution (¿?)

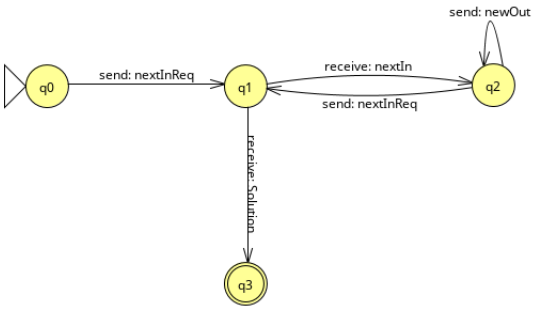


Figura 2: Protocolo Conductor - Problem

3.2. Node - Solver

- **NEW_IN**(*Node* → *Solver*)
  El *node* recibe una entrada, que debe hacer llegar al *solver* , que entenderá y utilizará la estructura del campo *In*(que habrá formateado el otro módulo, el *problem*) El mensaje se compone de los siguientes campos:

  - OpCode (1 byte): Código de operación
  - BytesOfField (4 bytes): Entero representando cuál es la longitud en *bytes* del siguiente campo
  - In(¿? bytes): La salida, en *ASCII*

OpCode(1B)	BytesOfField(4B)	In (¿?)

- **NEXT_OUT**(*Solver* → *Node*)
  El *solver* ha computado la entrada y devuelve esta salida al *node* , que deberá enviar al *conductor* . El mensaje se compone de los siguientes campos:

  - OpCode (1 byte): Código de operación
  - BytesOfField (4 bytes): Entero representando cuál es la longitud en *bytes* del siguiente campo
  - Output(¿? bytes): La solución, en *ASCII*, que deberá mostrarse por pantalla directamente mediante el *conductor* .

OpCode(1B)	BytesOfField(4B)	Output (¿?)

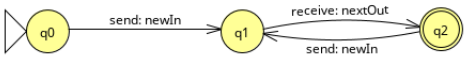


Figura 3: Protocolo Node - Solver

3.3. Node - Conductor

- **HAS_STARTED**(*Conductor* → *Node*)
  El *conductor* informa a un *node* de que el cómputo del problema ha comenzado o no, dependiendo del valor del campo *Answer*. El mensaje se compone de los siguientes campos:

  - OpCode (1 byte): Código de operación
  - Answer (1 byte): Si ha comenzado o no.

OpCode(1B)	Answer(1B)

- **START**(*Conductor* → *Node*)
  El *conductor* informa a un *node* que el cómputo del problema ha comenzado. El mensaje se compone de los siguientes campos:

  - OpCode (1 byte): Código de operación

OpCode(1B)

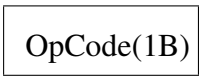
- **ADD_NODE**(*Node* → *Conductor*)
  El *node* quiere conectarse y participar en el cómputo del problema, enviándole información sobre su *hardware* y *software*. El mensaje se compone de los siguientes campos:

  - OpCode (1 byte): Código de operación
  - NºThreads (4 bytes): Entero que representa el número de *threads* disponibles en dicho *node*
  - BytesOfField (4 bytes): Entero representando cuál es la longitud en *bytes* del siguiente campo
  - CPU Arch (¿? bytes): Arquitectura de la cpu
  - BytesOfField (4 bytes): Entero representando cuál es la longitud en *bytes* del siguiente campo
  - OS (¿? bytes): Sistema Operativo

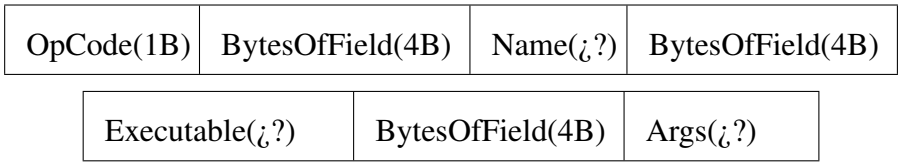
OpCode(1B)	NºThreads(4B)	BytesOfField(4B)	Arch(¿?B)	BytesOfField(4B)	OS(¿?)

- **BYE**(*Node* → *Conductor*/*Conductor* → *Node*)
  O *node* o *conductor* quieren cerrar la conexión con la otra parte(un mensaje *BYE* puede enviarlo cualquiera de los dos) El mensaje se compone de los siguientes campos:

- OpCode (1 byte): Código de operación



- **PROBLEM_MODULE**(*Conductor* → *Node*)
  El *conductor* envía el fichero ejecutable, el módulo *problem* a un *node* . El mensaje se compone de los siguientes campos:
  - OpCode (1 byte): Código de operación
  - BytesOfField (4 bytes): Entero representando cuál es la longitud en *bytes* del siguiente campo
  - Name (¿? bytes): Nombre del ejecutable
  - BytesOfField (4 bytes): Entero representando cuál es la longitud en *bytes* del siguiente campo
  - Executable (¿? bytes): Modulo ejecutable *problem*
  - BytesOfField (4 bytes): Entero representando cuál es la longitud en *bytes* del siguiente campo(puede ser 0)
  - Args (¿? bytes): Argumentos del ejecutable(opcional) *problem*



- **NEW_OUT**(*Node* → *Conductor*)
  Mismo formato que el *NEW_OUT* del protocolo *conductor - problem*
- **NEW_OUT_BYE**(*Node* → *Conductor*)
  Mismo formato que el *NEW_OUT* previamente descrito, pero con un significado especial. Aunque este mensaje contiene también una salida, servirá para, además de entregarla, terminar la conexión entre el *node* y el *conductor*
- **NEW_IN**(*Conductor* → *Node*)
  Mismo formato que el *NEW_IN* del protocolo *node - solver*

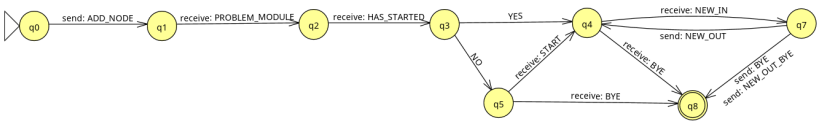


Figura 4: Protocolo Node - Conductor

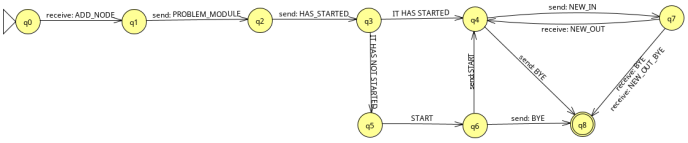


Figura 5: Protocolo Conductor - Node

4. Módulos del proyecto

Puesto que el protocolo sólo implementa el *node* y el *conductor* , el *problem* y *solver* deben proporcionarse de forma separada para resolver un problema concreto. A continuación de detalla de qué manera hay que estructurar dichos módulos. Debe existir un directorio con tres ficheros:

- 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 *conductor* , y será el quien gestione todo lo necesario para poner en marcha el protocolo.

5. Servicios del protocolo

5.1. Reparto justo del trabajo

El trabajo que realiza cada *node* es controlado directamente por la propia respuesta del mismo. Esto quiere decir que el *conductor* no gestiona de ninguna forma qué trabajo tiene qué *node* , sino que, estos serán servidos con más trabajo cuando terminen el que se les había asignado. Por ello, es muy importante que el trabajo que solicite el *problem* sea de un coste computacional bajo, para no tener a los *node* implicados mucho tiempo trabajando en una sola entrada, sino proporcionar entradas más pequeñas, teniendo cuidado de no ser demasiado pequeñas, lo que podría provocar una bajada del rendimiento debido a un exceso de intercambio de mensajes en la red para llevar a cabo el trabajo.

5.2. Eficiencia del protocolo

La clave del rendimiento del protocolo radica en la potencia de los *node* involucrados. Esto quiere decir, a más, mayor rendimiento y menor tiempo. Es importante destacar que es deseable implementar los módulos *solver* con la concurrencia en mente, pues esto mejorará enormemente el rendimiento. Como se veía en la Figura 1, lo ideal es que cada *solver* pueda aprovechar todos los núcleos de la CPU del *node* en el que se esté ejecutando, lo que mejoraría la velocidad sensiblemente.

5.3. Tolerancia a errores

Hay varios casos de error que se han contemplado, intentando actuar de la forma más correcta posible ante los mismos:

- **Desconexión de un elemento del protocolo**: Un *node* o el *conductor* puede desconectarse en cualquier momento(lo más probable es que sea un *node*) sin previo aviso, y podría provocar un fallo grave en el protocolo. Por ello, los *sockets TCP* deben establecer un *timeout* para asegurar que la conexión sigue estando activa. En caso contrario, debe actuarse en consecuencia, dependiendo de la situación.

6. Problemas

Problemas cuya aplicación es idónea en el protocolo:

- **Fuerza bruta**
- **Problema de las n-reinas**
- **Números primos**

7. Mejoras

El protocolo está lejos de ser un buen protocolo, y aún quedan muchas cosas por mejorar, de las que se consideran las siguientes:

- **Reparto del trabajo justo**: Gestionar dinámicamente qué cantidad de trabajo debe llevar a cabo cada nodo, de la forma más justa posible.
- **Desconexión del *node*** : Mejorar la situación en la que *node* está esperando el mensaje *START* del *conductor* ; si quiere salir, debe esperar a recibirlo, no puede salir inmediatamente.
- **Más información para mostrar en el *node*** : Añadir más mensajes para permitir mostrar más información en el *node* , como el porcentaje de trabajo computado, tiempo trabajando, tiempo estimado...etc