Practica 1.Documentació adicional

AST

1 Introducció

L'objectiu d'aquesta pràctica és repassar Java, presentar de forma simplificada l'estructura d'un protocol de transport i entendre bé l'estructura de dades *cua circular*.

2 Arquitectura d'un protocol de transport

Els protocols de transport controlen les transmissions de dades entre dues aplicacions (extrem a extrem). Les connexions de transport són bidireccionals (les dades viatgen en els dos sentits) i per tant els dos extrems són simètrics. En aquestes pràctiques però, per simplificar i fer més clar el disseny dels protocols, sempre es diferencia entre emisor (Sender) i receptor (Receiver) i es considera que només és l'emisor qui envia dades (més endavant es veurà que la informació de control del protocol sí que va en els dos sentits).

Els protocols de nivells inferiors (IP, ...) escapen de l'objectiu d'aquests exercicis, i de fet, aquí els protocols de transport accedeixen directament al canal (Channel), on es considera que hi hauria implementada tota la pila de protocols de nivells inferiors.

L'arquitectura general dels protocols de transport és la que es mostra a la figura 1. En les primeres pràctiques però s'utilitza una arquitectura una mica més simple, la que mostre la figura 2. Tot seguit es detallen i es fa una primera implementació dels diferents blocs.

2.1 Channel (Nivells inferiors)

El `Channel` representa el canal, i d'alguna manera, es pot suposar que engloba tots els protocols de nivells inferiors. Aquí, el `Channel` és simulat, i es fa utilitzant una cua circular.

Una cua circular és una estructura de dades on els elements es guarden en forma *circular*, és a dir, cadascun té un successor i un predecessor, veure figura 3.

2.1.1 Exercici 1: Implementar una cua cricular

El primer exercici demana crear una cua circular en una classe `CircularQueue` i que implementi la interfície `Queue` que es detalla en l'enunciat de la pràctica.

La notació `<E>` representa genèrics, és a dir, `E` és un tipus genèric (una classe qualsevol) que ja serà determinada en el moment de crear/instanciar un cua concreta. A

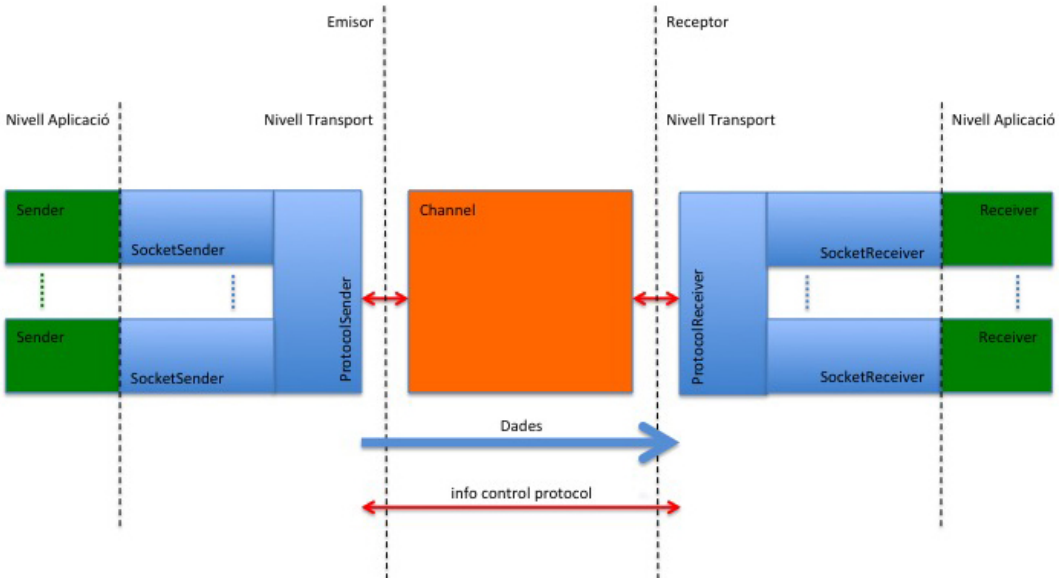

Figure 1: Arquitectura d'un protocol de transport

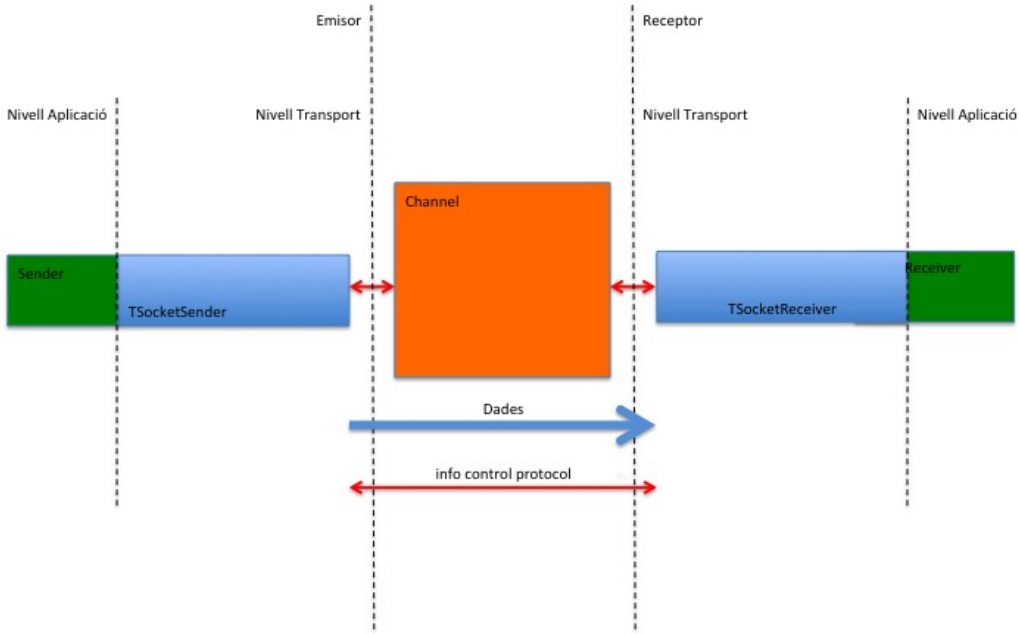

Figure 2: Arquitectura simplificada d'un protocol de transport

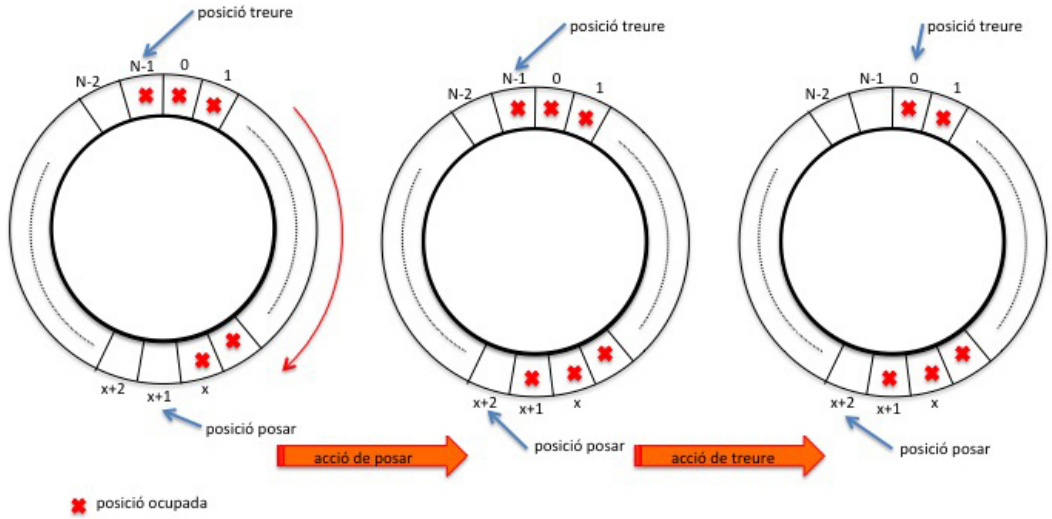

Figure 3: Cua circular de grandària N. Accions de treure i posar.

l'hora de implementar la `CircularQueue` s'haurà de mantenir els genèrics. Per crear un array usant genèrics es pot fer tal i com segueix:

```
private final T[] cua;
private final int N;

 public CircularQueue(int N) {
        this.N = N;
        cua = (T[]) (new Object[N]);
  }
```

I un esquelet de `CircularQueue` és:

```
public class CircularQueue<E> implements Queue<E> {
  private final T[] cua;
  private final int N; //Grandaria de la cua circular
    ...
  public CircularQueue(int N) {
      ...
   }
   ...
}
```

Recordar fer tots els tests que s'indiquen a l'enunciat de la pràctica.

2.1.2 Exercici 2: Implementar la classe `QueueChannel`

La classe `QueueChannel`, que representa el canal, té un atribut de classe `CircularQueue` i dos mètodes, un per posar elements a la cua, que representa l'acció d'enviar (recordar

que implementa la interfície `Channel`).

```
//Fa un put a la cua
public void send(TCPSegment s) {
  ...
}
```

i un per treure elements de la cua, que representa l'acció de rebre

```
//fa un get de la cua
public TCPSegment receive() {
   ...
}
```

Es fa notar que ja s'ha fixat el *genèric* que apareixia en la implementació de la classe `CircularQueue`, i en concret és la classe `TCPSegment`. Aquesta classe és proporciona a les llibreries de les pràctiques de l'assigantura `ast.protocols.tcp.TCPSegment`. Veure detalls a l'enunciat de la pràctica.

2.2 TSocketSender/TSocketReceiver (Nivell transport)

Aquestes classes són les pròpies del nivell de transport (és on s'implementarà bona part del protocol) i són les que utilitzen les aplicacions per tal de mantenir la transmissió, és a dir, enviar i rebre dades. Per tant han de proporcionar mètodes per fer-ho. Inicialment només es consideren dos mètodes per a cada classe, un per enviar i rebre segments i l'altre per tancar la connexió. En aquesta primera versió de fet no s'implementa cap protocol.

Per la classe `TSocketSender` el mètode per enviar té la següent capçalera:

```
public void sendData(byte[] data, int offset, int length) {
     ...
}
```

Rep com a paràmetre un array de bytes, `data`, un `offset`, que és la posició dins de `data` on es comencen a considerar els bytes, i un `length` que és el número de posicions, a partir de la posició `offset`, que es consideren útils. Veure la figura 4. A partir de les dades útils es construeix un segment (`TCPSegment`) i s'envia.

Per la classe `TSocketReceiver` el mètode per rebre té la següent capçalera:

```
public int receiveData(byte[] data, int offset, int length) {
       ...
}
```

Aquest mètode obté un `TCPSegment` i a partir de les dades del `TCPSegment` omple `length` posicions de l'array de bytes, `data`, a partir de la posició indicada per l'`offset`. Retorna un `int` que és el número de bytes que s'han llegit, ja que no sempre té que coincidir amb

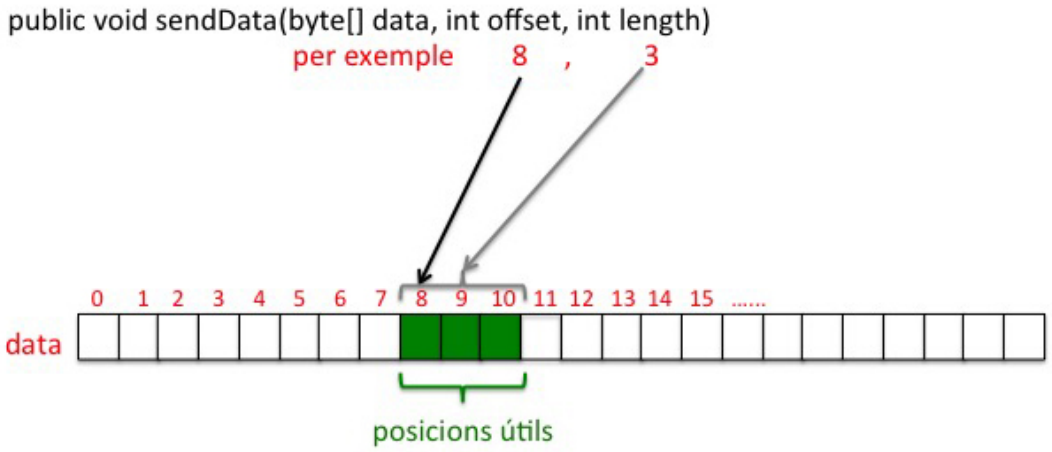

Figure 4: Offsets i lengths

`length` (el segment podria portar menys bytes per exemple). En el cas que el segment no conté dades, retorna -1.

Notem, que tal i com mostra l'esquema de la figura 2, aquestes classes tenen accés al `Channel` i per tant l'han d'obtenir com a paràmetre en el constructor.

```
public class TSocketSender {
    private final Channel ch;

    public TSocketSender(Channel ch) {
        this.ch=ch;
    }
    ...
}
```

2.2.1 Exercici 3: Implementar les classes `TsocketSender` i `TsocketReceiver`

Els mètodes per enviar i rebre dades ja s'han comentat, només queda el mètode `close()`. A l'hora de tancar una connexió TCP s'utilitza un protocol de desconnexió massa complex per aquesta pràctica. Per simplificar-ho, suposarem que per indicar el final de transmissió s'envia un segment sense dades. La implementació del mètode `close()` del `TSocketSender` seria:

```
public void close() {
        //envia un segment sense dades per indicar
        //final de transmissió
        TCPSegment s = new TCPSegment();
        ch.send(s);
}
```

2.3 Sender/Receiver (Nivell aplicació)

El `Sender` i el `Receiver` són les aplicacions que utilitzen el protocol. Aquí és on s'han d'implementar *tests* que demostrin que la implementació que s'ha fet del nivell de transport és satisfactòria i/o per mostrar les limitacions que té.

Un test pot ser el que es proposa a la pràctica, enviant línies introduïdes per teclat. Una altra possibilitat pot ser transmetre un fitxer de l'emisor cap al receptor. Per fer-ho s'utilitzen streams. Un Stream és un medi utilitzat per llegir dades d'una font i per escriure dades en un destí. La font i el destí poden ser fitxers, memòria, processos, el teclat, la pantalla, i també sockets. Els streams són unidireccionals. Per utilitzar-los s'han de construir relacionant-los directament amb la font (serà un input stream) o el distí (serà un output stream).

Un esquelet de la classe `Sender` pot ser el següent, tot i així, cadascú pot fer el seu propi sender i implementar els tests que cregui necessaris.

```
public class Sender {

    private final TSocketSender tss;
    private final int N = 10;
    private FileInputStream fr;

    //el fitxer poema.txt ha d'estar en la carpeta del projecte.
    public Sender(Channel ch) {
        tss = new TSocketSender(ch);
        try {
            fr = new FileInputStream("poema.txt");
        } catch (FileNotFoundException ex) {
            ex.printStackTrace();
        }
    }

    //llegeix N bytes del fitxer i els envia.
    //Retorna el número real de bytes enviats
    //-1 en cas de final de fitxer
    public int enviar() {
       ...
    }

    //Tanca l'stream al fitxer i la connexió
    public void close() {
       ...
    }
}
```

L'esquelet del `Receiver` seria equivalent:

```
public class Receiver {

    private final TSocketReceiver tsr;
    private final int N = 1000;
    private FileOutputStream fr;

    public Receiver(Channel ch) {
       ...
    }

    //Rep un segment i guarda els bytes rebuts al fitxer
    //retorna el número de bytes rebuts
    //-1 si no ha rebut cap byte (final)
    public int receive() {
        ...
    }

    //Tanca l'stream al fitxer i la connexió
    public void close() {
       ...
    }
}
```

Finalment és necessària una classe per executar tota la simulació. Aquí la simulació es fa des d'una mateixa màquina, i per tant el sender i el receiver s'executen en la mateixa màquina. I de fet, de moment, només sabem simular-ho de forma seqüencial, és a dir, es va enviant i rebent der forma alternada fins a acabar tota la transmissió. Un possible programa principal podria ser:

```
public static void main(String[] args) {
        Channel c = new QueueChannel(2);
        Sender s = new Sender(ch);
        Receiver r = new Receiver(ch);

        int ls = s.enviar();
        while (ls > 0) {
            int lr = r.receive();
            if (lr < 0) {
                System.out.println("Error en recepció!!");
            }
            ls = s.enviar();
        }
        s.close();
        r.close();
}
```