Pràctica 3. Implementació de Protocols de Transport. Fragmentació de Segments. Laboratori d'Aplicacions i Serveis Telemàtics

Josep Cotrina, Marcel Fernandez, Jordi Forga, Juan Luis Gorricho, Francesc Oller

1. Introducció

Continuarem amb el sistema introduït a la pràctica anterior. Com es va veure és un sistema dividit en 3 capes o nivells: nivell de xarxa, nivell de transport i nivell d'aplicació. En aquesta pràctica, ens centrarem principalment en els següents aspectes:

- A l'extrem emissor en la fragmentació de les dades d'aplicació en diferents segments.
- A l'extrem receptor en l'entrega de dades al nivell d'aplicació des del nivell de transport.
- En el nivell de xarxa, implementarem un canal mitjançant mecanismes de monitors. A més a més, hi afegirem la possibilitat de que una fracció dels paquets enviats es perdin i la restricció a una mida màxima de paquet.

Començarem pel nivell de transport.

2. Nivell de Transport. Transmissió de segments.

Paradoxalment, la primera cosa que farem serà començar dissenyant un protocol que perd segments en cas que aquests arribin massa "*ràpid*" al receptor.

Volem a més a més, fer una presentació el més didàctica possible, per això en l'esquema que es presenta la comunicació és simplex, on només l'emissor, que anomenarem `Sender`, envia dades d'aplicació al receptor, que anomenarem `Receiver`. Com ja s'ha vist en la pràctica anterior, el protocol de comunicació s'implementarà mitjançant les classes `TSocketSend` and `TSocketRecv`, però ara aquestes dues classes hereten de la classe `TSocketBase`:

```
public class TSocketBase {
    public static Log log = LogFactory.getLog(TSocketBase.class);

    protected Lock lk;
    protected Condition appCV;
    protected Channel channel;

        protected TSocketBase(Channel ch) {
        lk = new ReentrantLock();
        appCV = lk.newCondition();
        channel = ch;
    }
}
```

2.1. L'extrem emissor, `TSocketSend`.

Està clar que ens hem de fixar en la mida màxima dels segments. Acotar superiorment la mida màxima dels segments millora l'eficiència perque evita fragmentació de paquets en nivells inferiors en la torre de protocols. En aquesta part de la pràctica afegirem un nou mètode a l'interfície `Channel`.

El mètode `int getMMS()` retorna la mida màxima del camp de dades dels paquets de nivell de xarxa, perque no hi hagi fragmentació.

La realització d'aquesta classe es farà en el següent apartat.

```
public interface Channel {

    public int getMMS();

    public void send(TCPSegment seg);

    public TCPSegment receive();
}
```

El protocol en l'extrem emissor disposa del mètode:

```
void sendData(byte[] data, int offset, int length)
```

com ja s'ha vist en la Pràctica 2. Per evitar fragmentació en nivells inferiors, ara s'agafarà una seqüència de bytes provinents de les dades de l'aplicació i es posaran els bytes en **un o més** `TCPSegments` de mida màxima `sndMSS` i s'enviaran el o els segments pel nivell de xarxa.

2.1.1. Exercicis

Acabar d'implementar la classe `TSocketSend`. Inicialment poseu un valor petit a l'atribut `sndMSS`. Fer un test amb les classes de la Pràctica 2 i veure que no hi ha errors.

```
public class TSocketSend extends TSocketBase {

    protected int sndMSS;       // Send maximum segment size

    public TSocketSend(Channel ch) {
        super(ch);
        sndMSS = . . . // IP maximum message size - TCP header size
                       // Provar primer amb un valor petit
    }

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

        // Amb l'ajuda del mètode segmentize,
        // dividir la seqüència de bytes de data
        // en segments de mida màxima sndMSS
        // i enviar-los pel canal

    }

    protected TCPSegment segmentize(byte[] data, int offset, int length) {

        // Crea un segment que en el seu payload (camp) de dades
        // conté length bytes que hi ha a data a partir
        // de la posició offset.

    }

    protected void sendSegment(TCPSegment segment) {
```

```
        channel.send(segment);
    }
}
```

Fer un test amb les classes de la Pràctica 2 i veure que no hi ha errors.

2.2. L'extrem receptor, `TSocketRecv`.

La classe que implementa el protocol de transport en l'extrem receptor l'anomenem `TSocketRecv`. A diferència de la pràctica anterior, la classe `TSocketRecv` conté un objecte de tipus `Thread` que executa la següent tasca:

```
class ReceiverTask implements Runnable {
    public void run() {
        while (true) {
            TCPSegment rseg = channel.receive();
            processReceivedSegment(rseg);
        }
    }
}
```

Les accions de la tasca són autoexplicatives, agafa un segment del nivell de xarxa i el passa al mètode

`void processReceivedSegment(TCPSegment rseg).`

En l'extrem receptor, a part del thread que executa aquesta tasca hi ha el thread `Receiver` que és qui invoca el mètode `int receiveData(byte[] buf, int offset, int length)`. L'intercanvi de dades entre aquests dos threads es fa a traves d'una cua `CircularQueue` de `TCPSegments` que anomenarem `rcvQueue`. La sincronització entre threads per accedir a aquesta cua es realitza mitjançant mecanismes de monitors amb els objectes definits a la classe `TSocketBase` que hem presentat més amunt. En concret, mentre `rcvQueue` sigui buida, el fil que invoca `receiveData` s'haurà d'esperar. Aquest fil podrà continuar en el moment que un segment rebut del canal sigui introduït a `rcvQueue`.

En `processReceivedSegment(TCPSegment rseg)` posarem el segment rebut a la cua `rcvQueue`, però si la cua `rcvQueue` està plena aquest segment s'ha de descartar.

Aquesta classe disposa d'un únic mètode public:

`int receiveData(byte[] buf, int offset, int length).`

Aquest mètode que és invocat pel fil de l'aplicació, agafa dades del payload dels segments rebuts que hi ha a `rcvQueue` i omple l'array `buf`, a partir de la posició `offset`, amb `length` bytes. Com que la quantitat de bytes demanada per l'aplicació no ha de coincidir amb el número de bytes continguts en un o més segments, es proporciona el mètode:

`int consumeSegment(byte[] buf, int offset, int length)`

per ajudar a extreure bytes d'un segment. Observeu que l'atribut `rcvSegConsumedBytes` indica la quantitat de bytes que s'han consumit del primer segment que hi ha a la cua `rcvQueue`.

2.2.1. Exercicis

Es demana:

1. Entendre què fa el mètode `int consumeSegment(byte[] buf, int offset, int length).`

2. Implementar el mètode `void processReceivedSegment(TCPSegment rseg)`. Si la cua `rcvQueue` no està plena, aquest mètode hi posa el segment rebut. Si està plena descarta el segment rebut. Per què?

3. Implementar el mètode `int receiveData(byte[] buf, int offset, int length)`. Aquest mètode "consumeix" un màxim de `length` bytes de dades dels segments que hi ha a `rcvQueue` i els posa a l'array `buf` a partir de la posició `offset`. Per això s'ajuda del mètode `consumeSegment`. El valor que retorna és el número de bytes que realment ha posat a `buf`, que pot ser menor que `length`. Per què?

4. Fer un test amb les classes de la Pràctica 2 i veure que no hi ha errors.

```
public class TSocketRecv extends TSocketBase {

    protected Thread thread;
    protected CircularQueue<TCPSegment> rcvQueue;
    protected int rcvSegConsumedBytes;
        // invariant: rcvQueue.empty() || rcvQueue.peekFirst().getDataLength() > rcvSegConsumedBytes

    public TSocketRecv(Channel ch) {
        super(ch);
        rcvQueue = new CircularQueue<TCPSegment>(20);
        rcvSegConsumedBytes = 0;
        thread = new Thread(new ReceiverTask());
        thread.start();
    }

    /**
     * Places received data in buf
     * Veure descripció detallada en Exercici 3!!
     */
    public int receiveData(byte[] buf, int offset, int length) {
        lk.lock();
        try {
            // wait until receive queue is not empty

              . . .

            // fill buf with bytes from segments in rcvQueue
            // Hint: use consumeSegment!

              . . .

        } finally {
            lk.unlock();
        }
    }

    protected int consumeSegment(byte[] buf, int offset, int length) {
        TCPSegment seg = rcvQueue.peekFirst();
        // get data from seg and copy to receiveData's buffer
        int n = seg.getDataLength() - rcvSegConsumedBytes;
        if (n > length) {
            // receiveData's buffer is small. Consume a fragment of the received segment
            n = length;
        }
        // n == min(length, seg.getDataLength() - rcvSegConsumedBytes)
        System.arraycopy(seg.getData(), seg.getDataOffset() + rcvSegConsumedBytes, buf, offset, n);
        rcvSegConsumedBytes += n;
        if (rcvSegConsumedBytes == seg.getDataLength()) {
            // seg is totally consumed. Remove from rcvQueue
            rcvQueue.get();
            rcvSegConsumedBytes = 0;
        }
```

```
        return n;
    }

    /**
     * TCPSegment arrival.
     * @param rseg segment of received packet
     */
    protected void processReceivedSegment(TCPSegment rseg) {
        lk.lock();
        try {

            . . .

        } finally {
            lk.unlock();
        }
    }


    class ReceiverTask implements Runnable {
        public void run() {
            while (true) {
                TCPSegment rseg = channel.receive();
                processReceivedSegment(rseg);
            }
        }
    }
}
```

3. Nivell de xarxa. Model d'un canal de comunicacions.

El nivell de xarxa és l'encarregat d'encaminar paquets en una xarxa de comunicacions. Com que el nostre objectiu és dissenyar un nivell de transport, per nosaltres el nivell de xarxa serà una "caixa negra" que disposarà de dos mètodes `send(TCPSegment seg)` i `TCPSegment receive()`, per enviar i rebre segments respectivament.

3.1. Modelat d'un nivell de xarxa ideal.

Per començar farem una implementació d'un model de xarxa *ideal*. Per ideal entenem que no introdueix errors, no perd segments, ni els reordena. Resulta que un mecanisme útil per implementar aquest model són els monitors.

El model a desenvolupar serà una classe `MonitorChannel`, que implementa l'interfície `channel` i per tant disposa dels mètodes `send(TCPSegment seg)` i `TCPSegment receive()`. El mètode `send(TCPSegment seg)` haurà de ser bloquejant si en el moment d'enviar un `TCPSegment` el canal està "al seu nivell màxim d'ocupació". El mètode `TCPSegment receive()` haurà de ser bloquejant mentre no hi hagi segments en l'extrem receptor del canal.

3.1.1. Exercicis

Implementeu la següent classe `MonitorChannel`, mitjançant els mecanismes de monitor que heu vist a classe de teoria. Tant amb monitors generals com amb monitors natius.

```
public class MonitorChannel {

    . . .

    public MonitorChannel() {  . . .  }

    public void send(TCPSegment seg) {  . . .  }
```

```
    public TCPSegment receive(){   . . .   }
}
```

Afegir el següent codi a la classe `MonitorChannel`

```
    . . .

    public static final int MAX_MSG_SIZE = 1480; // Link MTU - IP header

    . . .

    public int getMMS() {
        return MAX_MSG_SIZE;
    }
```

3.2. Modelat d'un nivell de xarxa amb pèrdues.

Ara passarem a implementar el model d'un canal una mica més real, introduint el concepte de *pèrdua* de segments.

3.2.1. Exercicis

Per això afegirem el següent constructor a la classe `MonitorChannel` que acabeu de fer:

```
public MonitorChannel(double lossRatio) {
    . . .
}
```

El paràmetre `lossRatio` representa la fracció de segments que perd el canal. Una manera de realitzar aquest canal amb pèrdues és modificant el mètode `send(TCPSegment seg)` de la següent manera. Es genera un nombre aleatòri entre 0 i 1. Si aquest nombre és menor que la fracció de segments que perd el canal llavors el mètode retorna, o sinó el segment es transmet.

4. Nivell d'aplicació

El nivell d'aplicació es dóna tot implementat.

4.1. Exercicis

1. Entendre què fan els mètodes `run()` de les classes `Sender` i `Receiver`.
2. Entendre quants threads intervenen i com es sincronitzen **entre ells i en els diferents nivells**.
3. Executar l'aplicació. Canviar els valors dels atributs `sendNum`, `sendSize`, `sendInterval` i `recvBuf`, `recvInterval` per tal que es perdin dades.

```
public class Main {
    public static void main(String[] args){
            Channel c = new MonitorChannel();
            new Thread(new Sender(c)).start();
            new Thread(new Receiver(c)).start();
```

```
    }
}

class Sender implements Runnable {
    public static Log log = LogFactory.getLog(Sender.class);

    protected TSocketSend output;
    protected int sendNum, sendSize, sendInterval;


    public Sender(Channel c, int sendNum, int sendSize, int sendInterval) {
        this.output = new TSocketSend(c);
        this.sendNum = sendNum;
        this.sendSize = sendSize;
        this.sendInterval = sendInterval;
    }

    public Sender(Channel c) {
        this(c, 20, 50, 100);
    }

    public void run() {
        try {
            byte n = 0;
            byte[] buf = new byte[sendSize];
            for (int i = 0; i < sendNum; i++) {
                Thread.sleep(sendInterval*10);
                // stamp data to send
                for (int j = 0; j < sendSize; j++) {
                    buf[j] = n;
                    n = (byte) (n + 1);
                }
                output.sendData(buf, 0, buf.length);
            }
            log.info("Sender: transmission finished");
        } catch (Exception e) {
            log.error("Excepcio a Sender: %s", e);
            e.printStackTrace(System.err);
        }
    }

}

class Receiver implements Runnable {
    public static Log log = LogFactory.getLog(Receiver.class);

    protected TSocketRecv input;
    protected int recvBuf, recvInterval;

    public Receiver(Channel c, int recvBuf, int recvInterval) {
        this.input = new TSocketRecv(c);
        this.recvBuf = recvBuf;
        this.recvInterval = recvInterval;
    }

    public Receiver(Channel c) {
        this(c, 25, 10);
    }

    public void run() {
        try {
            byte n = 0;
            byte[] buf = new byte[recvBuf];
            while (true) {
```

```
                int r = input.receiveData(buf, 0, buf.length);
                // check received data stamps
                for (int j = 0; j < r; j++) {
                    if (buf[j] != n) {
                        throw new Exception("ReceiverTask: Received data is corrupted");
                    }
                    n = (byte) (n + 1);
                }
                log.info("Receiver: received %d bytes", r);
                Thread.sleep(recvInterval);
            }
        } catch (Exception e) {
            log.error("Excepcio a Receiver: %s", e);
            e.printStackTrace(System.err);
        }
    }
}
```