Pràctica 5. Implementació de Protocols de transport. Control de Fluxe (*Stop & Wait*). Laboratori d'Aplicacions i Serveis Telemàtics

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

1 Introducció

En la sèrie de pràctiques realitzades fins ara hem usat un nivell de xarxa ideal en el que no es perden segments, els segments arriben sense errors i en el mateix ordre amb el que han sigut transmesos. Per ara, continuarem fent la mateixa suposició.

Ara bé, hi ha una altra suposició que hem fet fins ara i que és molt més irrealista: *tots els segments enviats per un extrem són acceptats i processats per l'altra extrem.* Aquesta suposició comporta una velocitat del receptor infinita o superior a la velocitat de l'emissor.

Què passa quan la velocitat del receptor no és prou alta per poder processar tots els segments que arriben ? En la pràctica 4 podem observar que la `ReceiverTask` de la classe `ProtocolRecv` és l'encarregada d'obtenir els segments que arriben i passar-los al *socket* corresponent. El processat en el *socket* (mètode `processReceivedSegment` de la classe `TSocketRecv`) consisteix en guardar el segment en memòria per a que les seves dades puguin ser consumides per l'aplicació; si no hi ha espai en memòria (la cua de recepció està plena) el segment es perd [1].

Per a evitar aquesta situació, incorporarem un control de fluxe que farà que l'emissor disminueixi la seva velocitat evitant que es perdin segments. El **control de fluxe** consisteix en usar un mecanisme de *feedback* que permeti notificar a l'emissor quan el receptor està disponible per al processat de nous segments. **L'emissor mai ha de transmetre segments si no està segur que el receptor els podrà acceptar**.

1.1 Exercicis

- Executeu la pràctica 4 i observeu com modificant els pràmetres que afecten les velocitats dels processos d'aplicació emissor i receptor, el protocol falla. Caldrà canviar els valors indicats en la construcció de les instàncies de `Sender` i `Receiver`.

2 Canal Full Duplex.

En aquesta pràctica hi ha per primer cop intercanvi de segments en els dos sentits. Per això, es necessita un canal full duplex que ja es dóna implementat. Aquesta implementació es composa de dues classes `FDuplexChannel` i `Peer`.

La classe `FDuplexChannel` encapsula 2 objectes de tipus `Peer`, que són precisament els atributs `left` i `right`. Intuitivament aquest dos objectes són un canal en cada sentit. Per exemple, invocar el mètode `send(TCPSegment seg)` sobre `left` implica enviar el segment `seg` des del `Host1` al `Host2`. El mateix passa amb el mètode `receive()`.

Per la seva banda, la classe `Peer`, encapsula en els atributs `txQueue` i `rxQueue`, dues cues de `TCPSegment`. En aquestes cues es guarden respectivament, els segments per enviar i els rebuts.

La classe `Peer`, disposa d'un `Thread` que executa la tasca `TxThreadTask`, aquest `Thread` es bloqueja en la cua d'emissió (`txQueue`) del `Peer`, n'agafa els segments i els posa en la cua de recepció (`rxQueue`) de l'altre `Peer`. Si

[1] Podriem pensar en aturar la `ReceiverTask` en lloc de perdre el segment. Ara bé, el nivell de xarxa simula el comportament d'IP en el que no hi ha control de fluxe i per tant en aquest cas seria la xarxa qui perdria el segment.

aquesta cua de recepció està plena, què fa? Perquè?.

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

    public static final int MAX_MSG_SIZE = 1480; // Link MTU - IP header
    public static final int QUEUE_SIZE = 8;

    protected Peer left, right;
    protected double t_rate;     // in segments per second
    protected double lossRatio; // in [0, 1)

    public FDuplexChannel(double rate, double lossRatio) {
        this.t_rate = rate;
        this.lossRatio = lossRatio;
        left = new Peer(1);
        right = new Peer(2);
    }

    public FDuplexChannel() {
        this(100.0, 0.0);
    }

    public Peer getLeft() {
        return left;
    }

    public Peer getRight() {
        return right;
    }


    public class Peer implements Channel {

      protected int addr;
      protected CircularQueue<TCPSegment> txQueue, rxQueue;
      protected Lock lk;
      protected Condition txNotFull, txNotEmpty, rxNotEmpty;
      protected Thread txThread;

      protected Peer(int addr) {
        this.addr = addr;
        txQueue = new CircularQueue<TCPSegment>(1);
        rxQueue = new CircularQueue<TCPSegment>(QUEUE_SIZE);
        lk = new ReentrantLock();
        txNotFull = lk.newCondition();
        txNotEmpty = lk.newCondition();
        rxNotEmpty = lk.newCondition();
        txThread = new Thread(new TxThreadTask(), "IP-" + addr + "-TX");
        txThread.start();
      }

      public int getAddr() {
        return addr;
      }

      /**
       * Get maximum transport message size
       */
      public int getMMS() {
            return MAX_MSG_SIZE;
      }
```

```
    public void send(TCPSegment seg) {
      if (Math.random() < lossRatio) {
          log.warn("------------>Channel.send: Segment to transmit is lost");
          return;
      }
      try {
          lk.lock();
          while (txQueue.full()) {
              txNotFull.await();
          }
          txQueue.put(seg);
          log.debug("channel send");
          txNotEmpty.signal();
      } catch (InterruptedException ex) {
          log.error(ex);
      } finally {
          lk.unlock();
      }
    }

    public TCPSegment receive() {
      TCPSegment resultat = null;
      try {
          lk.lock();
          while (rxQueue.empty()) {
              rxNotEmpty.await();
          }
          log.debug("channel receive");
          resultat = rxQueue.get();
      } catch (InterruptedException ex) {
          log.error(ex);
      } finally {
          lk.unlock();
      }
      return resultat;
    }

    class TxThreadTask implements Runnable {
      @Override
      public void run() {
          while (true) {
              try {
                  TCPSegment p;
                  lk.lock();
                  try {
                      while (txQueue.empty()) { txNotEmpty.await(); }
                      p = txQueue.get();
                      txNotFull.signal();
                  } finally {
                      lk.unlock();
                  }
                  Thread.sleep((int) (1000.0 / t_rate));
                  deliverPacket(p);
              } catch (InterruptedException e) {
                  // Interrupted when shutdown this node (see shutdown())
              }
          }
      }

      private void deliverPacket(TCPSegment p) {
          Peer dest = (Peer.this == left) ? right : left;
          if (dest != null) {
              dest.lk.lock();
              if (!dest.rxQueue.full()) {
```

```
                    log.debug("Peer(addr=%s)->deliverPacket(%s)", dest.addr, p);
                    dest.rxQueue.put(p);
                    dest.rxNotEmpty.signal();
                } else {
                    log.warn("------------>Peer(addr=%s)->Congestion: Segment to transmit is lost",
    addr);
                }
                dest.lk.unlock();
            }
        }

      } // End of class TxThreadTask

    } // End of class Peer

} // End of class FDuplexChannel
```

2.1 Exercicis

- Estudieu el codi de les classes `FDuplexChannel` i `Peer`.

3 Les classes `Protocol` i `TSocket`

Definim la classe `Protocol` que s'encarrega de la creació de sockets, el manteniment de la llista de sockets creats i la multiplexació dels segments rebuts, aspectes que ja s'han treballat en la pràctica 4.

Els sockets són instàncies de la classe *TSocket* amb les funcionalitats d'enviar dades des de l'aplicació, rebre dades a l'aplicació i processar els segments rebuts [2].

En la classe `TSocket` és per tant on haurem d'introduir els canvis corresponents al control de fluxe.

3.1 Exercicis

- Completeu el codi de la classe `Protocol`: mètodes `openWith`, `ipInput` i `getMatchingTSocket`. El codi d'aquests mètodes és similar o idèntic al que ja heu treballat en la pràctica 4 [3].

4 *Stop & Wait*

El control de fluxe usat en TCP és un algorisme de finestra lliscant similar al conegut *Go-Back-N*. Aquest algorisme tracta també el problema dels errors i pèrdues de segments, és força complicat i es deixa per més endavant el seu estudi detallat.

En aquesta pràctica realitzarem un altre algorisme molt més simple: el conegut com *Stop & Wait*. Ès el control de fluxe més simple que ens poguem imaginar.

L'algorisme *Stop & Wait* consisteix en:

- el receptor notifica que està preparat per acceptar un nou segment de dades (quan hi hagi espai per guardar el segment). Aquesta notificació es realitza enviant a l'emissor un segment especial (sense dades i amb el flag ACK) de reconeixement.

[2] En la pràctica 4, aquestes funcionalitats s'havien separat en classes diferents `TSocketSend` i `TSocketRecv`.

[3] En la pràctica 4, el mètode `openWith` s'havia anomenat diferent en cadascuna de les classes corresponents a la part emissora i part receptora: `openForOutput` en `ProtocolSend` i `openForInput` en `ProtocolRecv`.

- l'emissor, després d'enviar un segment de dades, espera rebre el segment de reconeixement corresponent. Enviarà el següent segment de dades només després d'haver rebut el reconeixement, la cual cosa indica que el segment podrà ser acceptat pel receptor.

En la pràctica 4 usavem un cua de segments en el receptor que mitigava una mica el problema de no tenir control de fluxe. En aquesta pràctica, per simplicitat, usarem una variable[4] `rcvSegment` que es pot veure com una cua de capacitat igual a 1. Per tant, el receptor haurà d'enviar el reconeixement només quan aquesta variable sigui igual a `null` (espai disponible).

Per a la part emisora, s'usa la variable *booleana* `sndIsUna` que indica si està pendent de rebre un reconeixement. El mètode `sendData` que és el que envia els segments (tal com es feia a la pràctica 4) haurà d'esperar `sndIsUna == false` abans d'enviar cadascún dels segments.

Inicialment el receptor té espai disponible i no ho notifica. Per tant l'emissor no espera cap reconeixemant abans d'enviar el primer segment.

4.1 Exercicis

- Completeu el codi de la classe `TSocket`.
- Executeu la pràctica i observeu com, independentment de les velocitats dels processos d'aplicació emissor i receptor, el protocol **no** falla.

5 Detalls d'implementació

Com es pot observar en el codi donat pels professors, en la classe `TSocket` es defineix una sola variable de condició `appCV` per aturar tots els processos d'aplicació, tant receptors com emissors. Això obliga a usar sempre `signalAll` quan es desperten els processos.

5.1 Exercicis

- Identifiqueu les diferents condicions que esperen els processos i reimplementeu la classe `TSocket` usant diferents variables de condició per a cada condició a esperar. Useu `signal` en lloc de `signalAll` en els punts adeqüats.

[4] Usar una cua amb capacitat > 1 sembla que no introduiria cap eficiència considerable en un protocol *Stop&Wait*