Pràctica 6. Implementació de Protocols de transport. Control d'Errors (*Stop & Wait - ARQ*). Laboratori d'Aplicacions i Serveis Telemàtics

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

1 Introducció

En les anteriors pràctiques hem suposat que el nivell de xarxa no introdueix errors ni pèrdues en la transmissió dels segments. En la realitat, en canvi, la capa d'IP pot perdre paquets i introduir errors en els *payloads*[1] que contenen els segments.

En aquesta pràctica, i amb l'objectiu de realitzar un protocol de transport fiable, tractarem el problema dels errors i pèrdues en la transmissió dels segments. Aquesta pràctica és la continuació de la pràctica 5 i consistirà en afegir al protocol *Stop & Wait* obtingut anteriorment el control d'errors i pèrdues.

Cal tenir en compte que en el protocol TCP s'utilitzen algorismes de control de fluxe i control d'errors molt més complexos que els que estem treballant en aquestes pràctiques 5 i 6. L'objectiu d'aquestes pràctiques és el d'introduir els aspectes més bàsics i experimentar amb les tècniques de programació d'aquests aspectes.

1.1 Exercicis

- Executeu la pràctica 5 i observeu com, modificant el pràmetre que afecta la tassa de pèrdues, el protocol falla. Caldrà canviar els valors indicats en la construcció de la instància de `FDuplexChannel`.

2 Protocols ARQ

Automatic Repeat Request (ARQ) és un mètode de control d'errors molt usat en protocols extrem a extrem i que ofereix una transferència fiable de la informació sobre un canal no fiable.

Consisteix en l'us dels següents mecanismes:

- Els missatges (segments en el nostre cas) incorporen un codi de detecció d'error (CRC o *checksum*), afegit per l'emissor i que permet al receptor determinar si s'han produit errors durant la transmissió.
- Quan el receptor reb un segment sense errors respon amb un reconeixement a l'emissor.
- Si l'emissor no reb el reconeixement abans d'un determinat temps de *timeout*, retransmet el segment. La retransmissió es repeteix fins que l'emissor rebi el reconeixement.
- Per evitar duplicats, els segments porten un número de seqüència. El receptor ignora les dades d'un segment duplicat però ha de respondre amb el reconeixement al segment ja rebut.
- Per evitar ambigüitats en l'emissor, els reconeixements també porten un numero de seqüència.

[1] La capçalera d'IP incorpora un *checksum* però no protegeix les dades del paquet

Existeixen 3 tipus de protocols ARQ: *Stop & Wait ARQ*, *Go-Back-N ARQ* i *Selective Repeat ARQ*. Es distingeixen bàsicament per la qüantitat de bits dels números de seqüència i el tamany de les finestres d'emissió i recepció.

El protocol TCP és un *Go-Back-N ARQ* modificat i adaptat als requerimets i condicions que imposa el nivell de xarxa IP. En aquesta pràctica realitzarem, en canvi, un protocol *Stop & Wait ARQ*.

3 Implementació

3.1 Detecció d'error

En la pràctica no tractarem la generació en l'extrem emissor del *checksum* (com es fa en TCP) i suposarem que el nivell de xarxa s'encarrega d'aquesta funció.

La comprobació del *checksum* en l'extrem receptor tampoc la farem en el nivell de transport (com es fa en TCP) i suposarem que el nivell de xarxa simplement descarta els segments erronis.

Per tant la tassa de pèrdues simulada en la classe `FDuplexChannel` inclou la probabilitat d'errors en la xarxa.

3.2 Retransmissions

Considereu el següent escenari: l'emissor transmet un segment i la xarxa perd o descarta aquest segment o el segment de reconeixement del receptor. En qualsevol cas, el reconeixement no arriba a l'emissor. Aleshores, després d'un temps suficientment gran que permeti assumir que el segment s'ha perdut, l'emissor haurà de retransmetre el segment.

En l'emissor caldrà guardar en memòria tots els segment enviats i pendents de reconeixement. El tamany d'aquesta memòria en un protocol *Stop & Wait* és 1, i per tant usarem la variable `sndUnackedSegment` (que es pot veure com una cua de capacitat igual a 1). L'emissor estarà esperant reconeixement quan aquesta variable sigui diferent de `null` i quan rebi el corresponent reconeixement posarà `sndUnackedSegment` a `null`.

3.2.1 Temporitzadors

El sistema de retransmissions requereix un servei de temporitzacions. La variable `timerService` de la classe `TSocket` és un servei de creació de temporitzacions amb la següent interfície (classe `ast.util.Timer`):

```
public class Timer {

    /**
     * Creates and executes a one-shot action that becomes
     * enabled after the given delay in given time units.
     */
    public Task startAfter(Runnable task, long delay, TimeUnit unit) { ... }

    public void shutdown() { ... }

    public class Task {
        /**
         * Attempts to cancel future execution of given task.
         * This attempt will fail if the task has already completed,
         * has already been canceled, or could not be canceled for some other reason.
         * If successful, and this task has not started
         * when cancel is called, this task should never run.
         */
        public void cancel() { ... }
    }
}
```

Observeu com la variable `sndRtTimer` de `TSocket` és la referència a la temporització de retransmissió en curs. El control d'aquesta temporització i la corresponent tasca a realitzar quan es produeix el *timeout* es realitza en els següents mètodes de `TSocket`:

```
protected void startRTO() {
    if (sndRtTimer != null) sndRtTimer.cancel();
    sndRtTimer = timerService.startAfter(
        new Runnable() {
            @Override public void run() { timeout(); }
        },
        SND_RTO, TimeUnit.MILLISECONDS);
}

protected void stopRTO() {
    if (sndRtTimer != null) sndRtTimer.cancel();
    sndRtTimer = null;
}

/**
 * Timeout elapsed.
 */
protected void timeout() {
    lk.lock();
    try{
        // Actions to perform after timeout expires
        log.debug("%s->timeout()", this);
        if (sndUnackedSegment != null) {
            sendSegment(sndUnackedSegment);
        }
    } finally {
        lk.unlock();
    }
}
```

3.2.2 Números de seqüència

En aquest protocol usarem números de seqüència a nivell de segments. El primer segment enviat tindrà el número 0, el següent 1, i així successivament. Cal observar que en un protocol *Stop & Wait* és suficient tenir números de seqüència amb un sol bit (aritmètica mòdul 2).

Es disposa de la variable `sndNxt` que manté el número de seqüència del següent segment encara no transmés. En la part receptora, es disposa de la variable `rcvNxt` que manté el número de seqüència del segment esperat.

Els segments, tant de dades com de reconeixement, hauran de portar el corresponent número de seqüència. Per això usarem els corresponents camps de la capçalera TCP, i que podrem fixar i consultar amb els mètodes `setSeqNum`/`getSeqNum` i `setAckNum`/`getAckNum` respectivament de la classe `TCPSegment`.

4 Exercicis

- Completeu el codi de la classe `Protocol`. El codi d'aquests mètodes és similar o idèntic al que ja heu treballat en la pràctica 5.
- Completeu el codi de la classe `TSocket`.
- Executeu la pràctica i observeu com, independentment de la tassa de pèrdues en la xarxa, el protocol **no** falla.