Pràctica 7. Implementació de Protocols de transport.
Pràctica d'establiment i alliberament de connexions en comunicacions de transport.
Laboratori d'Aplicacions i Serveis Telemàtics

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

1. Introducció

Continuant amb la sèrie de pràctiques per aprendre a dissenyar un protocol de transport fiable, ara tractarem el problema d'establir i alliberar connexions en protocols de transport orientats a connexió.

De manera informal podriem dir que un protocol orientat a connexió és aquell en el qual abans d'intercanviar dades d'aplicació, s'estableixi una connexió lògica entre els dos extrems de la connexió. Normalment, un dels extrems comença enviant una petició per obrir una connexió, a la qual l'altre extrem respon. A aquest intercanvi inicial de peticions, que anomenarem *handshake*, es passa informació de control per determinar si i com la connexió s'ha d'establir. Si el handshake té exit, llavors es podrà procedir a l'intercanvi de dades.

Quan entre els dos extrems de la comunicació ja no hi ha més dades a intercanviar, la connexió s'ha d'alliberar. Normalment, cada extrem acaba la connexió per separat enviant un segment especial.

Cal destacar des del principi que en aquesta pràctica en centrarem en establir i alliberar una connexió, i per això, **no hi haurà intercanvi de dades d'aplicació**.

Un exemple de protocol orientat a connexió és el celebrat TCP (Transmission Control Protocol). Un exemple de protocol no orientat connexió és l'UDP (User Datagram Protocol).

En general, en els protocols orientats a connexió, els extrems de la connexió es mouen per diferents estats perque han de saber per quina part de la "conversa" d'intercanvi de missatges passen. En aquesta pràctica, hem dissenyat un diagrama d'estats del protocol que, per qüestions didàctiques, és una simplificació del diagrama d'estats del protocol TCP. Aquest diagrama es mostra en la Figura 1 .

2. Establiment d'una connexió

A l'hora d'establir una connexió els dos extrems tenen papers diferents. Un dels dos extrems, que anomenarem *extrem passiu*, espera a que l'altre extrem, l'*extrem actiu* comenci a establir la connexió. Els dos extrems es representaran per una instància de la classe `TSocket`. Normalment, en un mateix host de la xarxa hi haurà extrems actius i passius alhora, és per això que en la classe `Protocol` hi ha dues cues:

```
public class Protocol {
    . . .

    protected ArrayList<TSocket> listenTSocks;     /* All unbound TSockets (in state LISTEN) */
    protected ArrayList<TSocket> activeTSocks;     /* All bound TSockets (in active states) */

    . . .
}
```

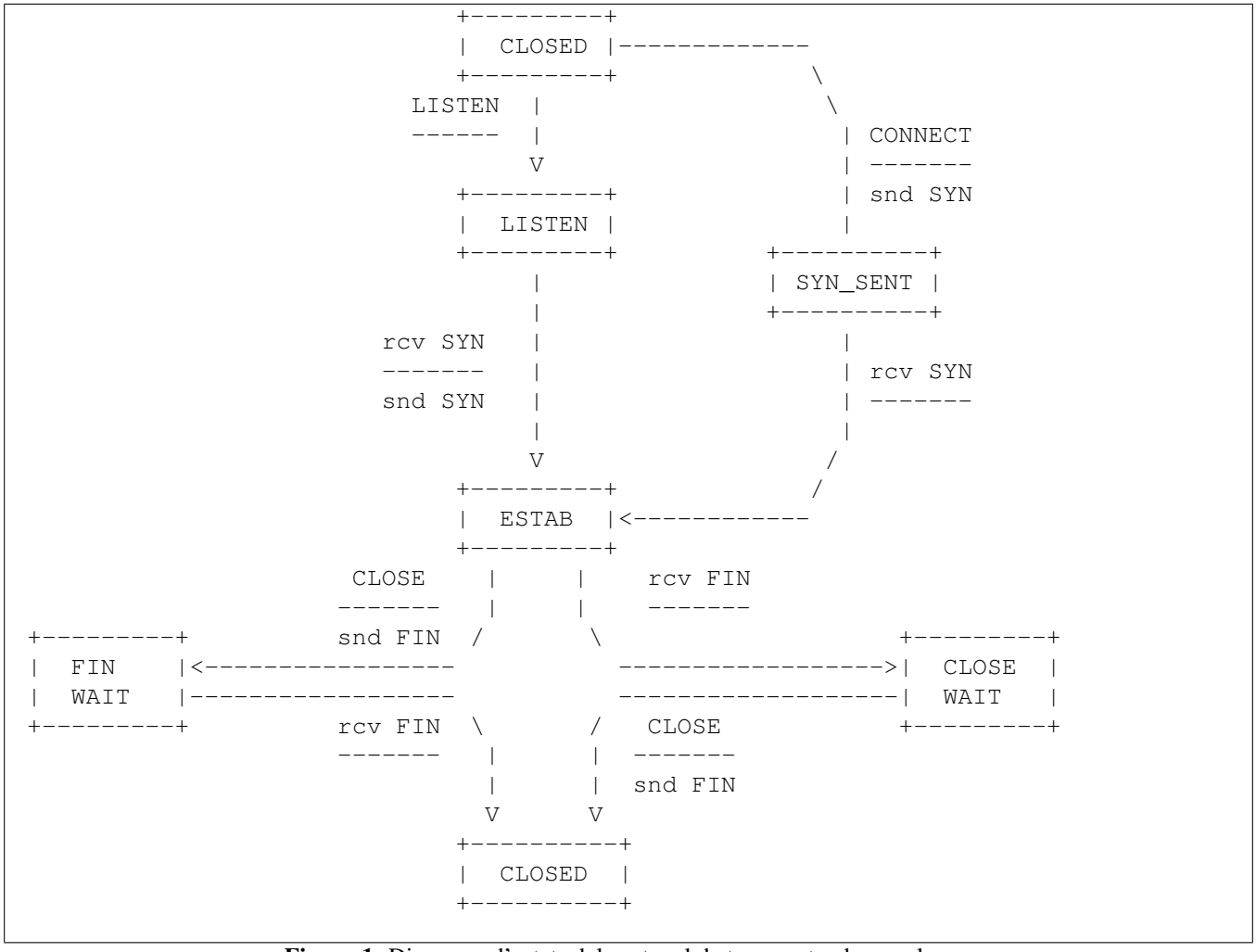


Figura 1: Diagrama d'estats del protocol de transport a desenvolupar.

2.1. L'extrem passiu

L'*extrem passiu*, utilitza els mètodes `listen()` i texttt accept(). Per obtenir una instància d'quest tipus d'objecte s'invoca al mètode `openListen(int localPort)` de la classe `Protocol`. Això es pot veure en mètode `run()` de la classe `Host1`:

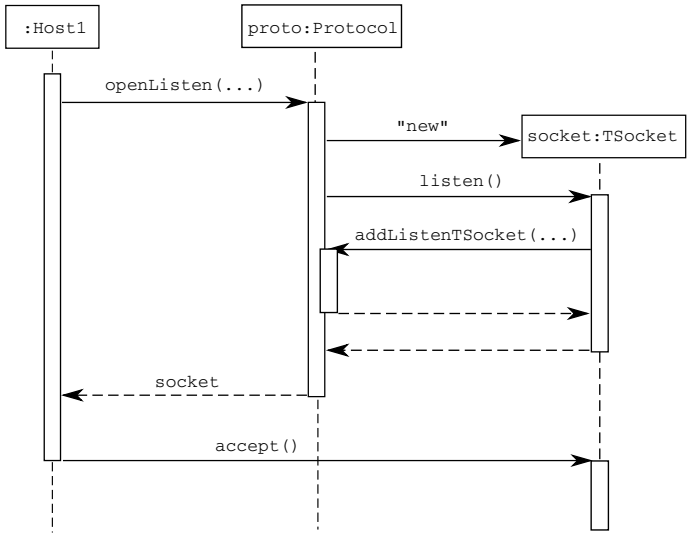
```
class Host1 implements Runnable {

    ...

    public void run() {
        log.info("Server started");
        TSocket serverTSocket = proto.openListen(Host1.PORT);
        while (true) {
            TSocket sc = serverTSocket.accept();
            new Thread(new Service(sc)).start();
        }
    }
}
```

En aquest mètode `run()` també es pot observar com s'utilitza el mètode `accept()` que es bloqueja en espera d'una nova connexió i retorna un objecte de tipus `TSocket` que manté una connexió virtual amb l'extrem remot que és qui ha iniciat aquesta connexió.

Si ens fixem en el codi de la classe `Protocol` veurem que en el mètode `openListen(int localPort)` hi ha una crida al mètode `listen()` de la classe `TSocket`:

```
public TSocket openListen(int localPort) {
    // Comprobar que el port no esta ocupat
    if (portInUse(localPort, listenTSocks) || portInUse(localPort, activeTSocks)) {
        log.error("openListen: port %d is in use", localPort);
        return null;
    }
    TSocket socket = new TSocket(this, localPort);
    socket.listen();
    return socket;
}
```

En el següent diagrama es pot veure una explicació gràfica del que hem dit:



És a dir, una crida a `openListen(int localPort)`, primer mira si el port al qual volem associar el nou `TSocket` està disponible, si és així, crea una nova instància de `TSocket` i invoca el mètode `listen()` sobre aquesta nova instància. Aquest mètode es pot veure a continuació:

```
    protected void listen() {
      lk.lock();
      try {
        log.debug("%s->listen()", this);
        acceptQueue = new CircularQueue<TSocket>(5);
        state = LISTEN;
        proto.addListenTSocket(this);
        logDebugState();
      } finally {
        lk.unlock();
      }
    }
```

La necessitat de la cua `acceptQueue`, es veurà més endavant.

2.2. L'extrem actiu

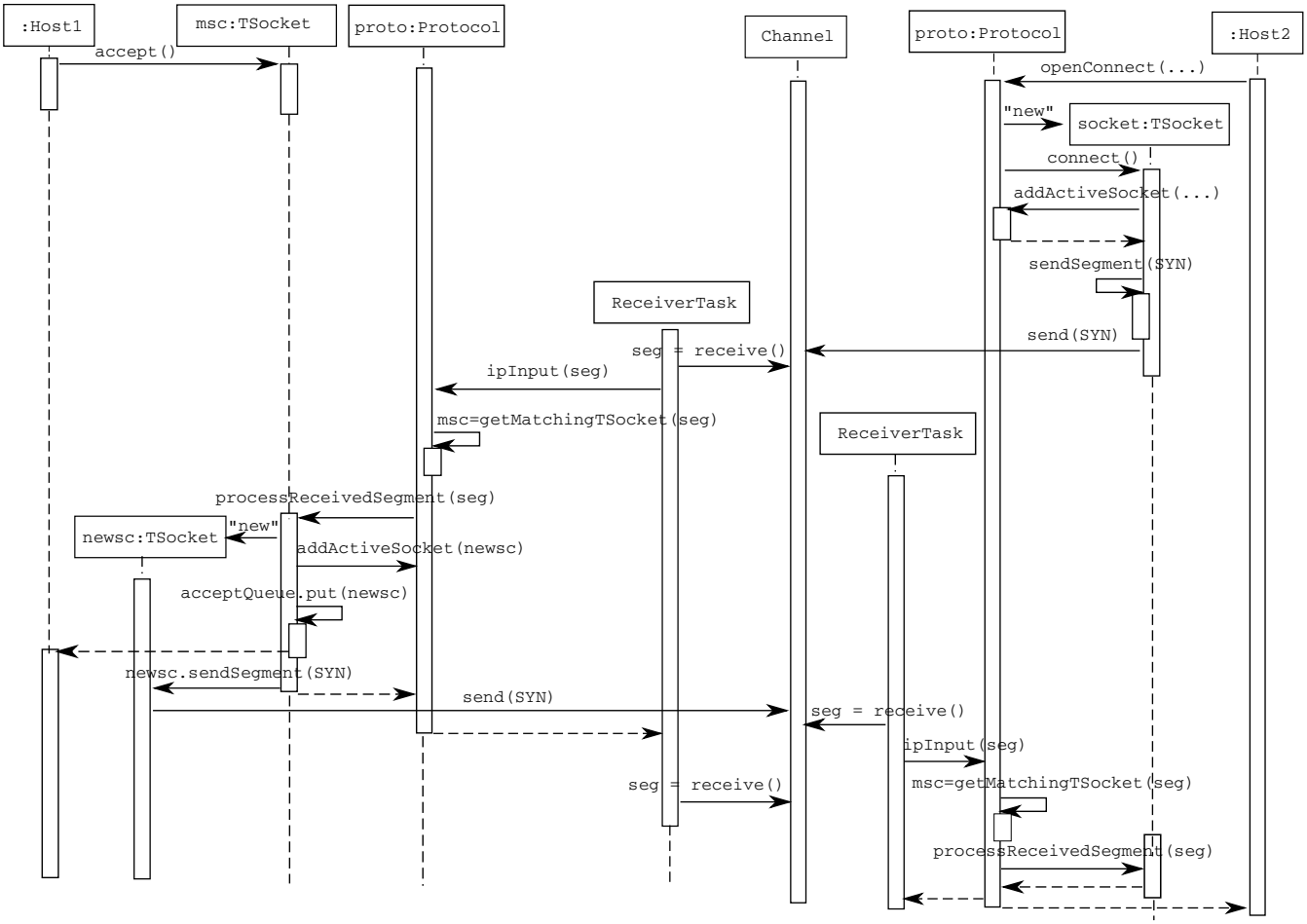
Com hem dit, l'extrem que inicia la connexió, s'acostuma a anomenar *extrem actiu*. A continuació es pot veure com obtenir un objecte d'aquest tipus, que és invocant el mètode `openConnect(int remotePort)` de la classe `Protocol`:

```
class Host2 implements Runnable {
    . . .

    class Client implements Runnable {
        public void run() {
            log.info("Client started");
            TSocket ts = proto.openConnect(Host1.PORT);
            log.info("Client connected");
            ts.close();
            log.info("Client disconnected");
        }
    }
}
```

Si ens fixem en aquest mètode `openConnect(int remotePort)`, podem observar que hi ha una crida al mètode `connect()`, per iniciar l'establiment d'una connexió.

```
public TSocket openConnect(int remotePort) {
    int localPort = newPort();
    TSocket sock = new TSocket(this, localPort);
    sock.connect(remotePort);
    return sock;
}
```

En la següent figura es mostren de manera gràfica les accions a partir d'una crida a `openConnect(int remotePort)`:



És a dir, a partir d'una crida a `openConnect(int remotePort)`:

1. Es crea un nou objecte `TSocket`
2. S'invoca el mètode `connect(int remPort)` sobre aquest nou objecte. En aquest mètode es realitzen les següents accions:
   a) S'afegeix el `TSocket` a la llista de sockets actius del protocol mitjançant el mètode `addActiveTSocket`.
   b) Crear un nou segment de tipus `SYN` i enviar-lo.
   c) Posar el `TSocket` en estat `SYN_SENT`.
   d) Esperar a que l'estat del `TSocket` sigui `ESTABLISHED`. (Això passarà quan es rebi un segment de `SYN` de l'extrem remot).
3. L'extrem remot, que haurà invocat `accept()` rep el segment de `SYN`. El `Protocol` decidirà quin `TSocket` l'ha de processar. Aquest processament consisteix en:
   a) Es crea un nou `TSocket` que mantindrà la connexió lògica en l'extrem local, i s'inicialitza correctament (ports local i remot, estat).
   b) S'afegeix aquest nou `TSocket` a la llista de sockets actius del protocol.

c) També s'afegeix aquest nou `TSocket` a la cua `acceptQueue`. Algun thread està esperant a que aquesta cua no estigui buida?.

d) Es crea un nou segment de tipus `SYN`, que és enviat pel **nou** `TSocket` creat.

4. Aquest segment de tipus `SYN` es rep a l'altre extrem. El seu processament per part del `TSocket` , (que és el que ha iniciat la connexió), implica canviar l'estat d'aquest a `ESTABLISHED`. Hi ha algun thread que esperi a que l'estat d'aquest `TSocket` passi a `ESTABLISHED`?

2.3. Exercicis

1. Canviar el mètode `getMatchingTSocket(...)` de la classe `Protocol` que vàreu fer en anteriors pràctiques, de manera que busqui tant en la cua `activeTSockets` com en la cua `listenTSockets`. Perquè?
2. Implementar el mètode `accept()`. Aquest mètode espera fins que la cua `acceptQueue`, i un cop no ho està en treu i retorna el primer element.
3. Implementar el mètode `connect()`.
4. Completar el mètode `processReceivedSegment`, en els casos `LISTEN` i `SYN_SENT`.

3. Alliberament d'una connexió

De la mateixa manera que abans d'iniciar l'intercanvi de dades s'ha d'establir una connexió, un cop ha finalitzar l'intercanvi la connexiò s'ha de tancar. D'aquesta manera s'alliberaran els recursos del sistema utilitzats per la connexió.

Per tancar una connexió, la classe `TSocket` disposa del mètode `close()`. Aquest mètode, a més a més de canviar l'estat del `TSocket` segons el diagrama d'estats de la Figura 1 (el `TSocket` passarà de `ESTABLISHED` a `FIN_WAIT`), envia un segment de tipus `FIN` a l'altre extrem.

Quan el `TSocket` de l'altre extrem rebi el segment de tipus `FIN`, en processar-lo canviarà d'estat, i despertarà al thread que està esperant per rebre dades (més sobre això en properes pràctiques). Eventualment, també es cridarà al mètode `close()` des d'aquest extrem, de manera que els dos extrems acabaran en estat `CLOSED`.

3.1. Exercicis

1. Implementar el mètode `close`.
2. Completar el mètode `processReceivedSegment`, en els casos `ESTABLISHED`, `FIN_WAIT` i `CLOSE_WAIT`.
3. Entendre que fa la classe `Main` i provar el codi realitzat. Com ja s'ha dit, en aquesta pràctica no hi ha intercanvi de dades d'aplicació.