Pràctiques d'AST
Curs 2024T

Pràctica 1

Introducció

Els objectius d'aquesta pràctica són refrescar java, programar l'estructura de dades cua, *circular* i *enllaçada*, utilitzades en pràctiques posteriors, i programar l'arquitectura (molt simplificada) d'un protocol de transport, que anirà evolucionant al llarg del curs.

1.1 Cua circular

Una cua circular és una estructura de dades, concretament una llista on cada element té un successor i un predecessor. Per a definir l'**estat** d'una cua circular són importants: la posició d'eliminació (extracció) d'elements, G (head), la posició d'inserció d'elements, P (tail), i el número d'elemnts, n, que conté la cua. Satisfan $P = (G + n) \mod N$, on N és la capacitat de la cua. Veure la figura 1.1 per a una representació gràfica.

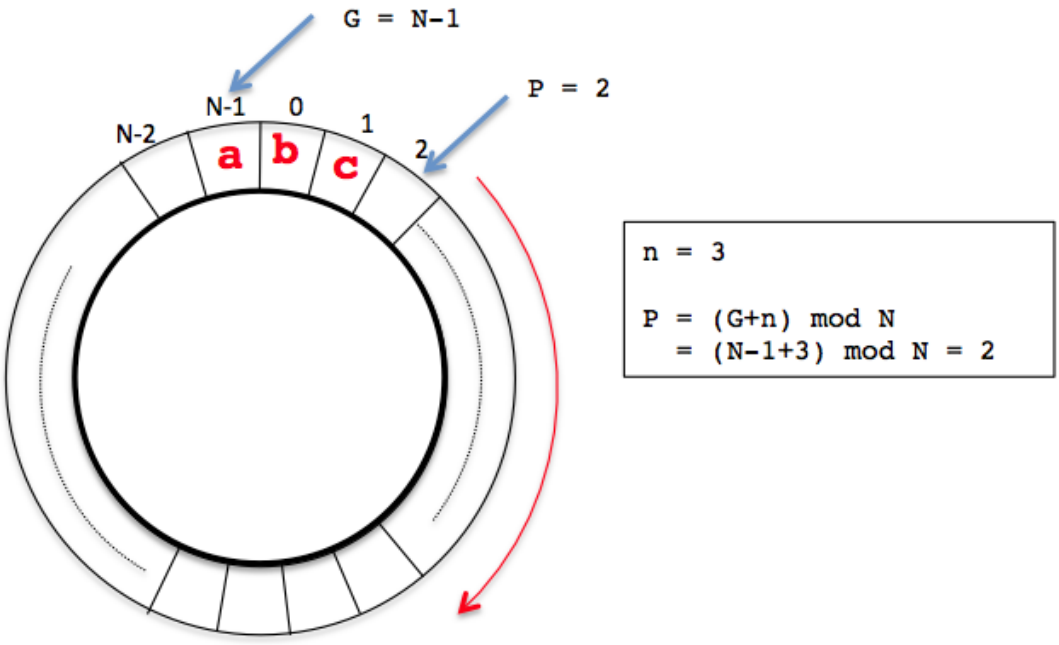

Figura 1.1: Exemple d'estat (no inicial) d'una cua circular.

Exercici 1 [Cua circular]

La classe `CircularQueue<E>` implementa la interfície `Queue`, que es proporciona (comentada) en el paquet `util`.

En el codi de la cua circular apareix l'identificador `E` que correspon al que s'anomena *tipus genèric*, i que representa una classe qualsevol, que es concreta més endavant, en el moment d'instanciar un objecte de la classe `CircularQueue`.

Exemple de definició d'un array de genèrics

```
public class CircularQueue<E> implements Queue<E> {
   private final E[] cua;
   private final int N;
   ...
   public CircularQueue(int N) {
        this.N = N;
        cua = (E[]) (new Object[N]);//<-------- array de genèrics
        ...
   }
   ...
}
```

Es demana implementar la classe `CircularQueue.java` (els mètodes estan comentats a la interfície `Queue`). El mètode `toString()` ha de retornar un `String` mostrant el contingut de la cua, de forma ordenada, des de la posició corresponent al *head* fins a la posició prèvia al *tail*. En aquest exercici no s'han d'implementar els mètodes de l'iterador.

Completar també la classe `TestCQ.java`. S'ha d'implementar un test de la cua circular, que en verifiqui el bon funcionament. A més de comprovar els mètodes `put(...)` i `get(...)`, també és important verificar els altres mètodes `size()`, `free()`, `empty()`, `full()` i `peekFirst()`, especialment per als casos de cua plena i de cua buida.

Exercici 2 [Cua circular: iterador]

Es demana completar la classe `MyIterator`, que és una classe interna de `CircularQueue.java`. Un iterador ha de permetre recórrer els elements de la cua.

- El mètode `boolean hasNext()` retorna `true` si encara queden elements per iterar.
- El mètode `E next()` retorna l'element actual i posiciona el cursor d'iteració a la següent posició.

Notar que aquests dos mètodes no modifiquen l'**estat** de la cua, són mètodes només de consulta.

- El darrer mètode, `void remove()`, sí que modifica l'**estat** de la cua, ja que elimina el darrer element consultat (el que ha retornat el mètode `next()`). Per tant, és important notar que abans d'invocar `remove()` s'ha d'haver invocat `next()`.

Es proporciona el test `TestCQ_ite.java`. La sortida ha de ser:

```
run:
Queue content: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
we iterate over the queue elements to take 0,4,5 and 9, if present:
taken: 0
taken: 4
taken: 5
taken: 9
present content of the queue: [1, 2, 3, 6, 7, 8]
Queue content: [1, 2, 3, 6, 7, 8, 10, 11, 12, 13, 14]
BUILD SUCCESSFUL (total time: 0 seconds)
```

1.2 Cua enllaçada

Una altra opció a l'hora d'implementar la interfície `Queue` és fer-ho en forma de cua enllaçada. Una cua enllaçada és una llista de nodes, on cada node té un atribut per a guardar les dades i un atribut per a referenciar el següent node de la llista (veure la figura 1.2).

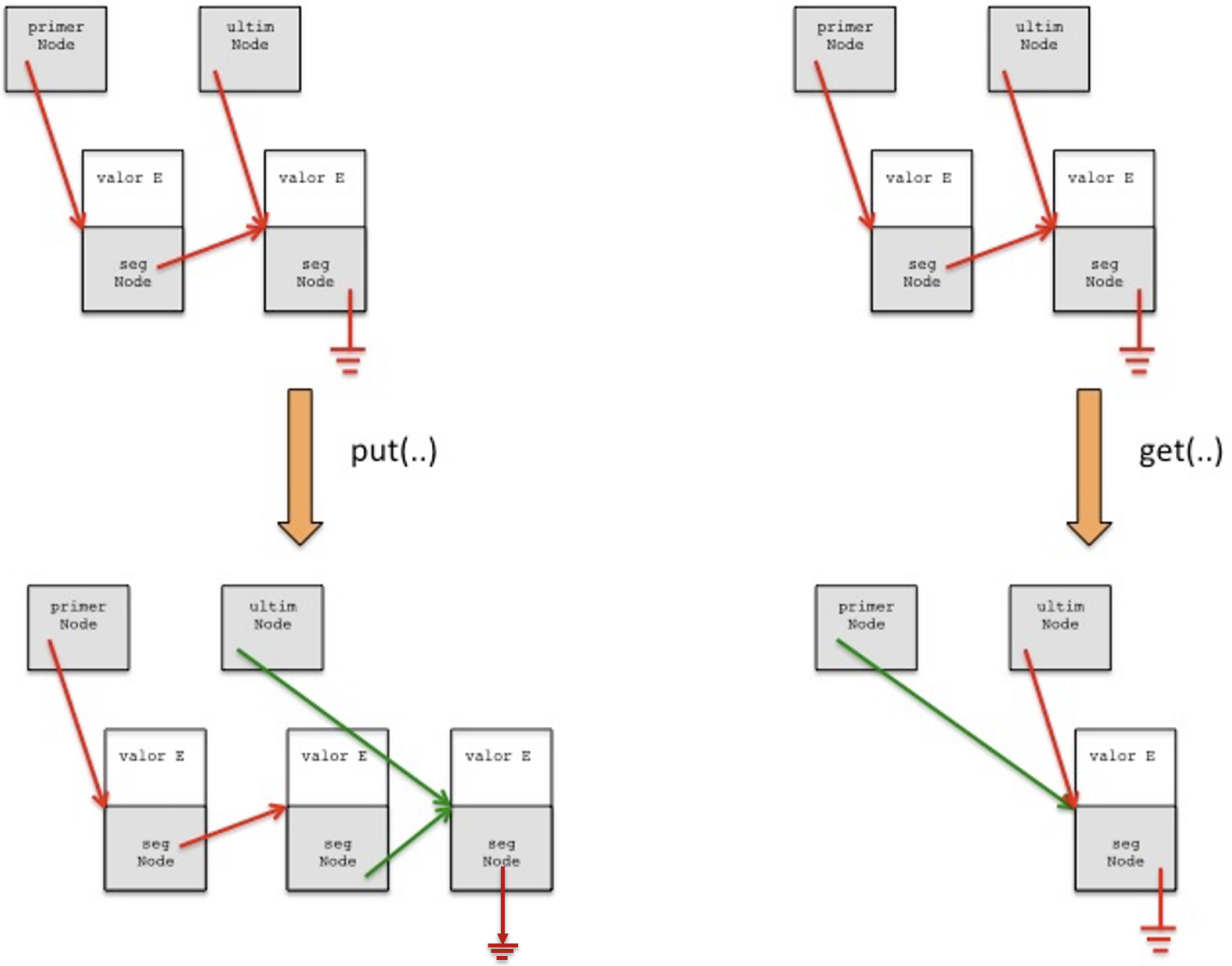

Figura 1.2: Representació gràfica d'una cua enllaçada i les operacions associades.

Per a matenir l'estructura (i de fet l'**estat**) de la cua, tal i com mostra la figura 1.2, a més del número d'elements de la cua, s'utilitza un atribut `primer` que conté una referència al primer element (node) de la cua i un atribut `ultim` que conté una referència al darrer element (node) de la cua. El procés de treure es fa a partir del primer element i afegir es fa *després* de l'últim.

Com a exemple es descriu breument el procés d'afegir un nou element a la cua, el procés de treure un element és similar. A l'hora d'afegir un element a la cua s'ha de crear un nou node, guardar-hi les dades i incrementar en una unitat el comptador del número d'elements de la cua. A més, per a mantenir l'estructura de la cua, s'han d'actualitzar les referències, `primer`, `ultim` i l'atribut `seg` del darrer node de la cua. S'ha de distingir el cas en que la cua està buida de la resta de casos. Si la cua està buida, el nou node és el primer de la cua (i també l'últim), per tant s'ha d'actualitzar `primer`. En altre cas, el nou node serà el següent de l'últim *actual*, per tant, en tots els casos, el nou node passa a ser l'últim.

Exercici 3 [Cua enllaçada]

Implementar la classe `LinkedQueue.java`. Implementar també la classe `TestLQ.java`. En aquesta classe s'ha d'implementar un test de la cua enllaçada, que verifiqui el seu bon funcionament (es pot aprofitar bona part del test de l'exercici 1, observant però que els mètodes `full()` i `free()` en aquest cas no tenen massa sentit).

Finalment s'ha d'implementar l'iterador, amb la mateixa funcionalitat que en el cas de la cua circular. També es proporciona un test per a aquest cas `TestLQ_ite.java`.

1.3 Protocol de transport

La figura 1.3 mostra l'arquitectura del protocol de transport que s'implementa en aquesta pràctica. Es comenta seguint els diferents nivells.

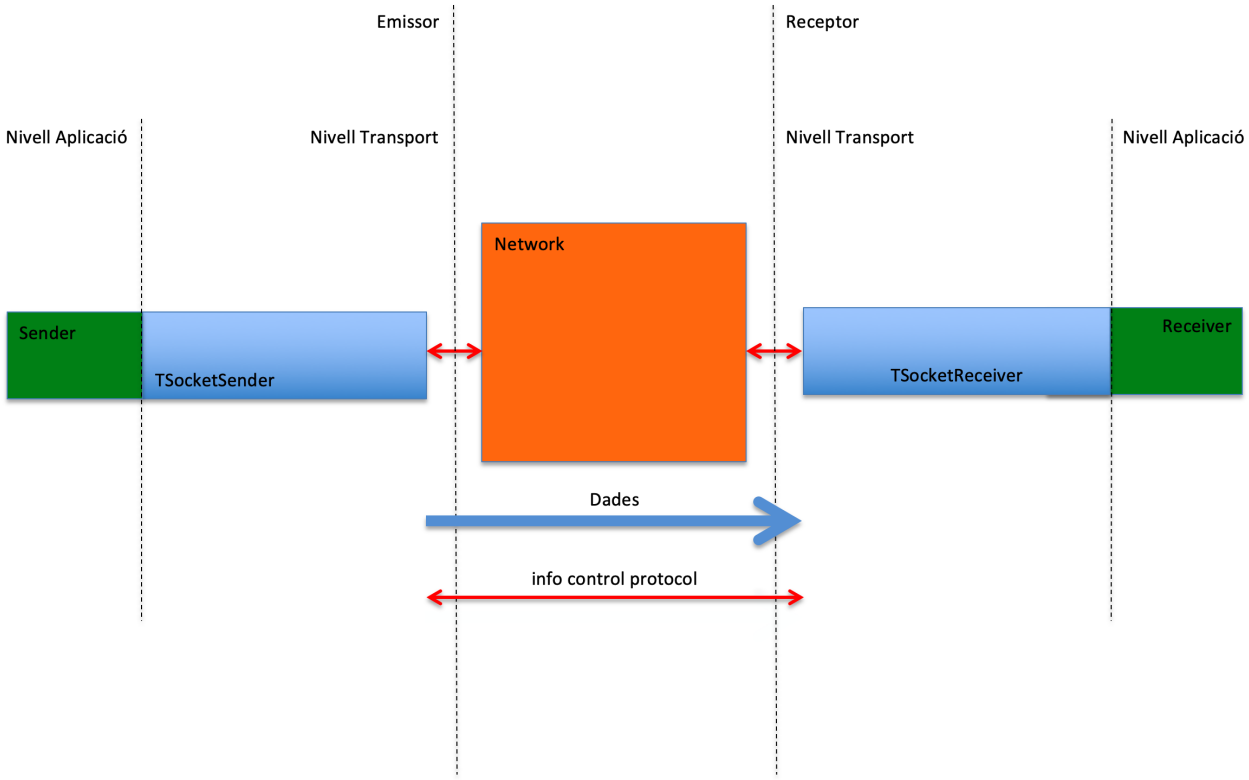

Figura 1.3: Arquitectura simplificada d'un protocol de transport.

1.3.1 Nivells inferiors

El bloc *Network* de la figura 1.3 representa tots els nivells que hi ha per sota del nivell de transport i que no s'estudien en aquest curs. Per a simular-ho es programa la classe `SimNet_Queue.java`, que implementa la interfície `SimNet.java`, proporcionada en el paquet `util`.

Exercici 4 [Xarxa]

Programar la classe `SimNet_Queue.java`. Té un atribut tipus `CircularQueue`, que és on es guarden les dades enviades i d'on s'obtenen les dades rebudes. Notar que és aquí on es concreta el genèric `<E>` de la cua circular.

1.3.2 Nivell de transport

El nivell de transport està format per dues classes `TSocketSend` (emissor) i `TSocketRecv` (receptor). En aquestes pràctiques es considera que les dades només van de l'emissor cap al receptor (del receptor cap a l'emissor, en pràctiques posteriors s'enviaran segments de control, però mai de dades).

Aquestes dues classes proporcionen a les aplicacions (nivell superior) els serveis necessaris per a mantenir una comunicació, és a dir, enviar i rebre dades i per tant han de proporcionar els mètodes per a fer-ho. Aquí només es considera un mètode a cada classe, un per a enviar i un per a rebre arrays de bytes.

Les dades que s'envien de l'emissor cap al receptor s'han d'encapsular en segments, i són aquests segments els que es passen a nivells inferiors, que en el nostre cas equival a enviar-los a la xarxa. Aquests segments corresponen a la classe `TCPSegment` del paquet `util`.

Exercici 5 [TSocketSend]

Completar el mètode per a enviar dades de `TSocketSend`, que té la següent capçalera:

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

Rep com a paràmetres un array de bytes, `data`, un `offset`, que és la posició dins de `data` on comencen els bytes útils (els que s'han d'enviar), i una `length` que és el número de posicions útils comptant a partir de la posició `offset` (veure la figura 1.4). Aquest mètode haurà de **crear un nou** segment, posar-hi les dades (bytes) útils (no és eficient posar-hi sempre tot l'array), activar el flag de `push` del segment (ja que és un segment de dades) i finalment enviar el segment a la xarxa (`network`).

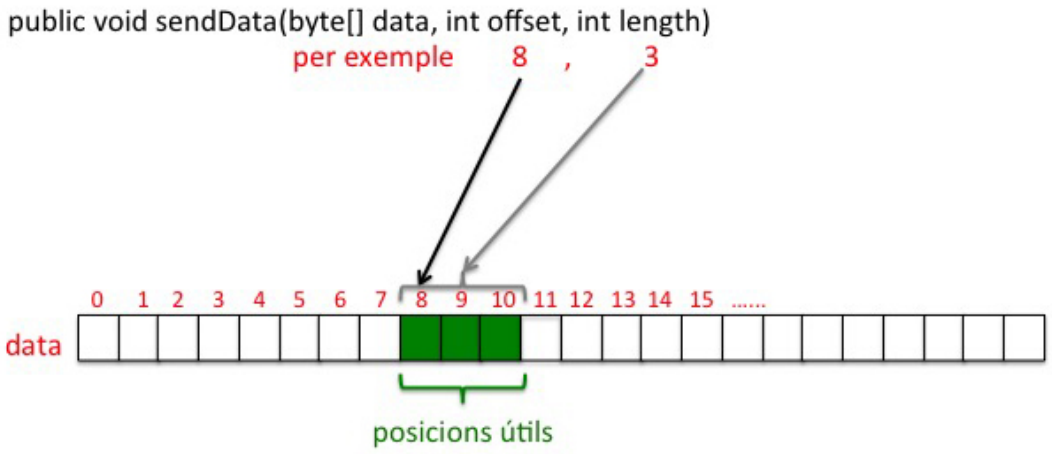

Figura 1.4: Exemple d'array de dades (`data,offset,length`).

Exercici 6 [TSocketRecv]

Completar el mètode per a obtenir dades de la classe `TSocketRecv`, que té la següent capçalera:

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

Aquest mètode obté un `TCPSegment` de la xarxa (`network`) i a partir de les dades del `TCPSegment` intenta omplir `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 posat realment a l'array, que no sempre coincideix amb `length` (el segment pot portar menys bytes que els `length` demanats per l'aplicació receptora).

1.3.3 Nivell d'aplicació

En aquestes pràctiques el nivell d'aplicació es dóna fet. Correspon sempre a la classe `Test.java`. Estudiar la classe i veure com es crea la xarxa, com es crea un `Sender` que utilitza un `TSocketSend` per a enviar dades, i com es crea un `Receiver` que utilitza un `TSocketRecv` per a rebre dades (revisar també aquestes dues classes per a entendre exactament quina és la seva lògica: què s'envia? què s'espera rebre?...).

Un cop implementades les classes `TSocketSend` i `TSocketRecv`, executar el test `Test.java`. La sortida que s'ha d'obtenir ha de ser semblant a la següent, on es mostra els segments enviats i rebuts a cada extrem (a cada `Socket`), així com les dades rebudes pel receptor. Això implica que s'han de posar els `println(...)` necessaris allà on calgui de les classes `TSocket...`. Per aconseguir les tabulacions que es mostren es pot fer servir el mètode `printRcvSeg(..)` de la classe

TSocket_base.java (tots els sockets hereten d'aquesta classe). També es poden utilitzar els mètodes proporcionats a la classe Log.java.

```
run:
snd --> [PSH, src = 0, dst = 0, seqNum = 0, data = {0}]
snd --> [PSH, src = 0, dst = 0, seqNum = 0, data = {1}]
snd --> [PSH, src = 0, dst = 0, seqNum = 0, data = {2}]
snd --> [PSH, src = 0, dst = 0, seqNum = 0, data = {3}]
snd --> [PSH, src = 0, dst = 0, seqNum = 0, data = {4}]
snd --> [PSH, src = 0, dst = 0, seqNum = 0, data = {5}]
snd --> [PSH, src = 0, dst = 0, seqNum = 0, data = {6}]
snd --> [PSH, src = 0, dst = 0, seqNum = 0, data = {7}]
snd --> [PSH, src = 0, dst = 0, seqNum = 0, data = {8}]
snd --> [PSH, src = 0, dst = 0, seqNum = 0, data = {9}]
Sender: transmission finished
                                        received: [PSH, src = 0, dst = 0, seqNum = 0, data = {0}]
                                        Receiver: received 1 bytes
                                        received: [PSH, src = 0, dst = 0, seqNum = 0, data = {1}]
                                        Receiver: received 1 bytes
                                        received: [PSH, src = 0, dst = 0, seqNum = 0, data = {2}]
                                        Receiver: received 1 bytes
                                        received: [PSH, src = 0, dst = 0, seqNum = 0, data = {3}]
                                        Receiver: received 1 bytes
                                        received: [PSH, src = 0, dst = 0, seqNum = 0, data = {4}]
                                        Receiver: received 1 bytes
                                        received: [PSH, src = 0, dst = 0, seqNum = 0, data = {5}]
                                        Receiver: received 1 bytes
                                        received: [PSH, src = 0, dst = 0, seqNum = 0, data = {6}]
                                        Receiver: received 1 bytes
                                        received: [PSH, src = 0, dst = 0, seqNum = 0, data = {7}]
                                        Receiver: received 1 bytes
                                        received: [PSH, src = 0, dst = 0, seqNum = 0, data = {8}]
                                        Receiver: received 1 bytes
                                        received: [PSH, src = 0, dst = 0, seqNum = 0, data = {9}]
                                        Receiver: received 1 bytes
                                        Receiver: reception finished
BUILD SUCCESSFUL (total time: 7 seconds)
```

Pràctica 2

Concurrència

L'objectiu d'aquesta pràctica és estudiar la concurrència i la problemàtica que porta associada. També es millora la simulació del protocol de transport, utilitzant les noves eines de concurrència.

2.1 Execució de processos concurrents

L'execució del protocol de la Pràctica 1 és una execució seqüencial. Només hi ha **un** procés, el procés *principal*, que executa el mètode `main()` de la classe `Test.java`. Simula l'acció d'un emissor i d'un receptor sobre una xarxa, però els dos depenen de l'únic procés que hi ha, i per tant, sempre s'executen de forma seqüencial i **ordenada**. Observant la classe `Test.java` de la pràctica anterior, es veu el codi

```
...
s.run();
r.run();
```

on primer l'emissor envia totes les dades a la xarxa i després són rebudes i tractades pel receptor.

Una simulació més realista s'aconsegueix si l'emissor (`Sender`) i el receptor (`Receiver`) són processos (*threads* o *fils*) diferents i independents, evitant així el comportament anterior, és a dir, l'emissor envia quan vol i el receptor rep també quan vol. Per aconseguir aquest nou comportament s'ha de saber com crear nous processos, ja que com a mínim se'n necessiten dos per a executar independentment el `Sender` i el `Receiver`.

Per a crear i executar un nou procés Java proporciona la classe `Thread.java`

```
class Thread implements Runnable{
  void run(){}
  void start(){...}
  static void sleep(long milis){...}
  void join(){...}
  ...
}
```

El mètode `start()` conté el codi necessari per a crear i executar un nou procés. El codi, o punt d'inici, on comença l'execució aquest nou procés és el mètode `run()`, que a la classe `Thread` és un mètode buit, sense codi. Llavors, si es vol un procés que executi un codi determinat, es crea una subclasse de la classe `Thread` i es sobreescriu el mètode `run()` amb el codi desitjat.

Exemple

```
class MyFil extends Thread{
  public void run(){
    //codi que executa el nou procés...
  }
}
...
```

```
MyFil mf = new MyFil(); //crea l'objecte (no el procés).
mf.start();//crea i arranca un nou procés que executa el mètode run().
```

També es pot indicar el codi que executarà el nou procés a partir de la implementació de la interfície `Runnable`, és a dir, creant classes que obligatòriament tindran un mètode `run()`. Llavors passar com a paràmetre un objecte que implementa `Runnable` al constructor d'un `Thread` és una alternativa per a indicar quin mètode `run()` ha d'executar el nou procés.

Exemple

```
class MyIFil implements Runnable{
  public void run(){
    //codi que executa el nou procés...
  }
}
...

MyIFil mif = new MyIFil();
Thread t = new Thread(mif);//crea un objecte de classe Thread
t.start();//crea i arranca un nou procés que executa el mètode run() de mif.
```

2.2 Problemàtica de la concurrència

Abans de millorar la simulació del protocol és interessant estudiar els problemes bàsics de la concurrència.

2.2.1 Zona crítica

Comencem amb un exercici que mostra un dels problemes de la concurrència: les zones crítiques.

Exercici 7 [paquet practica2.P0CZ]

Completar la classe `TestSum.java` de manera que arrenqui dos processos a partir de la classe `CounterThread.java` i al final, un cop han acabat els dos processos, imprimeixi el valor de la variable `x` (usar el mètode `join()`). Determinar el valor d'`I` pel qual comencen a aparèixer resultats inesperats.

L'exercici 7 reprodueix un dels problemes inherents de la concurrència, el problema de la **zona crítica**. A `TestSum` es creen i s'executen dos processos a partir de classe `CounterThread.java`. Aquests dos processos comparteixen la variable `static int x`, i el que fan és incrementar I cops el seu valor. Executant `TestSum` (amb `I=10`) s'obté:

```
run:
x: 20
BUILD SUCCESSFUL (total time: 0 seconds)
```

Ara bé, incrementant el valor d'`I`, arriba un moment que s'obtenen sortides inesperades com la següent (en l'exemple per a `I=10000`).

```
run:
x: 16375  //<-- el resultat esperat és 20000!!
BUILD SUCCESSFUL (total time: 0 seconds)
```

La causa del comportament evidenciat a l'exercici 7 està a

```
for (int i = 0; i < I; i++) {
    x = x + 1;
}
```

concretament a la sentència `x = x + 1;`. Cal observar que `x` és l'única variable compartida, la resta són variables locals de cada procés (i no porten problemes). A més, la sentència `x = x + 1;` no és una *acció atòmica*. Això vol dir (de forma molt simple) que la versió compilada de `x = x + 1;` correspon a vàries sentències de llenguatge màquina. Per tant és raonable considerar una codificació a nivell màquina com

$$x = x + 1 \longrightarrow \begin{cases} I_1. & R \leftarrow x & \text{//<— passar el valor de } x \text{ al registre } R \\ I_2. & inc(R) & \text{//<— incrementar el valor del registre } R \\ I_3. & x \leftarrow R & \text{//<— passar el valor de } R \text{ a la posició de memòria } x \end{cases}$$

on R representa un registre de la UAL. Si dos processos executen concurrentment el codi anterior, pot passar que es perdin increments, tal i com mostra el següent entrellaçat (pels canvis de context, es pot considerar el registre R *local* per a cada procés, i es representa per R_A i R_B).

temps	procés `fA`	R_A	procés `fB`	R_B	x
inici		0		0	0
(1)	$executa : R \leftarrow x$	0		0	0
(2)		0	$executa : R \leftarrow x$	0	0
(3)		0	$executa : inc(R)$	1	0
(4)		0	$executa : x \leftarrow R$	1	1
(5)	$executa : inc(R)$	1			1
(6)	$executa : x \leftarrow R$	1			1

Aquest comportament es pot reproduir en Java. Per a forçar que el procés `fA` deixi d'executar després de (1) es pot afegir un `sleep(1)` tal i com es mostra a continuació. El mètode `sleep(long milis)` adorm el procés que l'executa durant `milis` mil·lisegons.

```
@Override
public void run() {
    int R;//<-- representa el registre (notar que és local a cada procés)
    for (int i = 0; i < I; i++) {
        //x = x + 1;
        R = x;
        try {
            sleep(1);
        } catch (InterruptedException ex) {

        }
        R = R + 1;
        x = R;
    }
}
```

Executant aquest codi s'observa com es perden molts increments (l'exemple s'ha fet amb `I=100`).

```
run:
x: 100 //<-- el resultat esperat és 200!!
BUILD SUCCESSFUL (total time: 0 seconds)
```

Exercici 8 [paquet practica2.P0CZ]

Fer les modificacions anteriors a la classe `CounterThread.java` i observar que s'obtenen els resultats comentats.

Per tant, s'ha vist com la sentència `x = x + 1;` és problemàtica si s'executa concurrentment, és a dir, el resultat final de l'execució és indeterminat, i depèn de l'entrellaçat dels fils. Aquestes zones de codi s'anomenen **zones crítiques**. Normalment les zones crítiques són les zones de codi on es modifiquen recursos compartits per més d'un procés. En el nostre cas, el recurs compartit és la variable `x`.

Una opció per tal d'eliminar la incertesa en les execucions concurrents de les zones crítiques és precisament limitar la concurrència en aquestes zones. Només un únic procés podrà executar simultàniament la zona crítica. Una forma habitual de dir-ho és que les zones crítiques s'executen en **exclusió mútua**.

Actualment la majoria de llenguatges de programació ofereixen eines que permeten solucionar els problemes de concurrència de forma eficient. Java ofereix els **locks** i les **variables de condició**. Els locks permeten implementar l'accés en exclusió mútua a les zones crítiques.

La classe `ReentrantLock` de Java proporciona els mètodes `lock()` i `unlock()`. El mètode `lock()` actua com a barrera, deixant avançar un únic procés, és a dir, si dos o més processos executen el mètode `lock()`, només un el podrà superar. La resta de processos que executen el `lock()` i que no el poden superar perquè algun altre procés ja ho ha fet, s'adormen (deixen d'executar) i no utilitzen temps de CPU. Quan un procés surt de la ZC i executa el mètode `unlock()` desperta algun procés que està adormit (en el `lock()`) esperant entrar a la ZC, i per tant, ara que la ZC està lliure sí que tindrà l'opció d'avançar.

Exercici 9 [paquet practica2.POCZ.Monitor]

Conegut el problema es demana solucionar-lo. Es proporciona la classe `MonitorCZ.java` que és usada pels processos creats a partir de `CounterThreadCZ.java` per tal de fer els increments. S'ha de modificar la classe `MonitorCZ.java` utilitzant locks, de manera que la zona crítica s'executi en exclusió mútua. Completar també la classe `TestSumCZ.java`, que permet fer una simulació (crear dos processos concurrents a partir de `CounterThreadCZ.java`).

2.2.2 Sincronització de processos

Un altre aspecte important de la concurrència és poder sincronitzar els diferents processsos de manera que les possibles execucions sempre siguin correctes, o equivalentment, que l'aplicació sempre es comporti de forma correcta sigui quin sigui l'ordre d'execució (entrellaçat) dels processos.

Al paquet `Practica2.P1Sync` hi ha la classe `TestID.java` on es creen `N` (en el nostre cas `N=2`) processos a partir de la classe `CounterThreadID.java`. Aquests processos mostren per pantalla 10 cops el seu identificador (que és un número). Executant s'obté una sortida per consola equivalent a la següent:

```
run:
00001000111100011111
Simulation end.
BUILD SUCCESSFUL (total time: 0 seconds)
```

Notem que aquesta sortida no té perquè ser sempre la mateixa, doncs depèn de l'entrellaçat dels processos. De fet, qualsevol sortida amb deu 1's i deu 0's és possible (per al cas `N=2`).

Ara es vol sincronitzar l'execució d'aquests processos de manera que la sortida sigui sempre la mateixa i sigui:

```
run:
01010101010101010101
Simulation end.
BUILD SUCCESSFUL (total time: 0 seconds)
```

Una possible solució per aconseguir aquest comportament és assignar torns als processos, és a dir, un procés només pot mostrar el seu identificador quan és el seu torn, i un cop fet, ha de passar el torn a l'altre procés.

La dificultat està en com aturar el procés que no pot escriure perquè no és el seu torn. Per a fer-ho Java proporciona les **variables de condició**.

Variables de condició

Per aturar processos Java proporciona la classe `Condition` (variable de condició). Una variable de condició es pot interpretar com una classe que gestiona una cua de processos aturats, proporcionant mètodes per afegir/treure processos de la cua. Una variable de condició sempre està associada a un `Lock` i disposa dels mètodes següents:

- `await()`: Quan un procés executa l'`await()`, sempre dins d'una ZC controlada per un lock, allibera la ZC i passa a la cua de processos adormits associada a la variable de condició.
- `signal()`: Treu un procés de la cua de processos adormits de la variable de condició i el passa a la cua associada al lock, ja que la ZC està ocupada pel procés que ha executat el `signal()`. El procés que ha executat el `signal()` segueix normalment la seva execució dins la ZC.

- `signalAll()`: Equivalent al `signal()`, però afecta a tots els processos de la cua.

Exercici 10 [paquet practica2.P1Sync.Monitor]

Completar les classes del paquet `practica2.P1Sync.Monitor` per tal d'aconseguir el comportament desitjat. Fer-ho per a `N=2`. Pensar si la solució que heu proposat funciona per a `N>2` i també en el cas que hi hagués més d'un procés amb el mateix identificador.

2.2.3 Monitors

Amb l'ús de **locks** i **variables de condició** es pot resoldre qualsevol problema de concurrència. L'ús ordenat de totes aquestes eines porta al concepte de **monitor**. Els monitors són utilitzats per processos per tal d'accedir en EM a recursos, i també per a establir sincronització i comunicació entre processos.

Un monitor és una classe que representa una abstracció de dades, conjuntament amb les operacions de manipulació d'aquestes dades. Les dades poden representar qualsevol recurs compartit entre els processos.

Les operacions de manipulació de dades (els mètodes) s'executen en exclusió mútua, és a dir, tots els mètodes determinen una única zona crítica.

A més, un monitor ofereix sincronització condicional a través de les variables de condició (a vegades anomenades cues d'esdeveniments). Veure la figura 2.1 que representa la semàntica d'un monitor.

Els monitors són processos passius (no executen) sinó que ofereixen mètodes per a ser utilitzats per altres processos (actius). Un dels avantatges dels monitors és que es poden programar/dissenyar en un nivell important d'isolació.

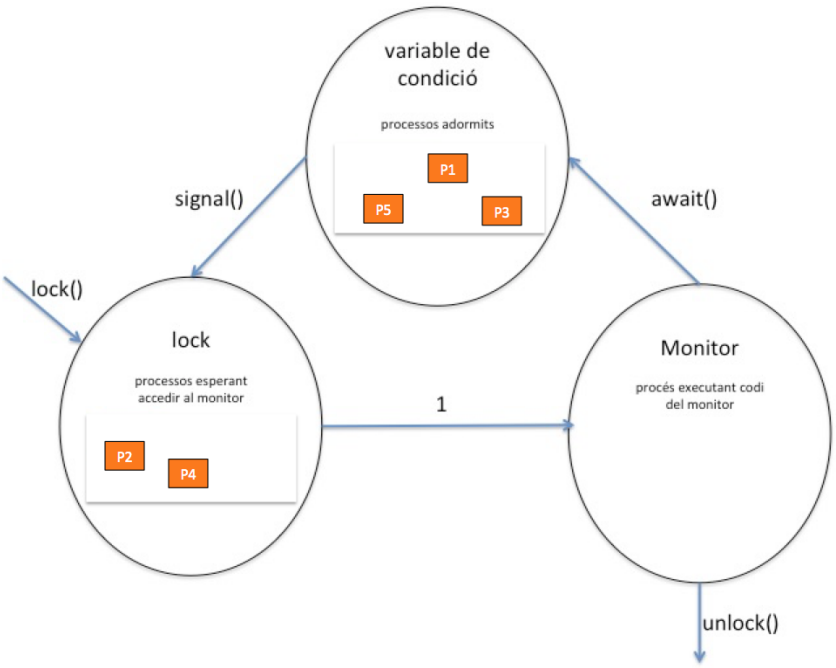

Figura 2.1: Semàntica de locks i variables de condició en un monitor

2.3 Simulació del protocol

Tal i com s'ha comentat al principi d'aquesta pràctica, l'execució de la pràctica 1 és seqüencial, és a dir, el mateix procés executa el codi de l'emissor i del receptor. Per aconseguir una simulació més realista, el codi de l'emissor i del receptor l'han d'executar processos diferents, ja que habitualment emissor i receptor estan en màquines diferents.

Exercici 11 [paquet practica2.Protocol]

Modificar la classe `Test.java`, del paquet `Practica2.Protocol`, fent que l'emissor i el receptor corresponguin a processos diferents. Intentar forçar l'execució per tal d'obtenir errors del tipus que es mostren a sota, i per tant, demostrar que al fer-ho concurrent apareixen problemes.

```
run:
Receiver exception: queue empty
...
BUILD SUCCESSFUL (total time: 0 seconds)
```

El problema en aquest cas és a la xarxa (`SimNet_Queue.java`). Ara la xarxa és un objecte compartit per dos processos (`Sender` i `Receiver`), però no hi ha cap sincronització, ni control de zona crítica. Per tant, s'ha de crear una nova xarxa (en forma de monitor), on a més de la zona crítica es sincronitzin els processos de manera que:

1. Si un procés vol posar un element a la cua i aquesta està plena s'ha d'esperar fins que hi hagi lloc.
2. Si un procés vol treure un element de la cua i aquesta està buida s'ha d'esperar fins que hi hagi un element disponible.

Exercici 12 [paquet practica2.Protocol]

Completar la classe `SimNet_Monitor.java` de manera que satisfaci les especificacions anteriors. Utilitzant la nova xarxa a la classe `Test.java`, una execució ha de mostrar per consola un resultat equivalent al següent:

```
rrun:
snd --> [PSH, src = 0, dst = 0, seqNum = 0, data = {0}]
                                          received: [PSH, src = 0, dst = 0, seqNum = 0, data = {0}]
snd --> [PSH, src = 0, dst = 0, seqNum = 0, data = {1}]
                                          Receiver: received 1 bytes
snd --> [PSH, src = 0, dst = 0, seqNum = 0, data = {2}]
snd --> [PSH, src = 0, dst = 0, seqNum = 0, data = {3}]
                                          received: [PSH, src = 0, dst = 0, seqNum = 0, data = {1}]
                                          Receiver: received 1 bytes
snd --> [PSH, src = 0, dst = 0, seqNum = 0, data = {4}]
snd --> [PSH, src = 0, dst = 0, seqNum = 0, data = {5}]
                                          received: [PSH, src = 0, dst = 0, seqNum = 0, data = {2}]
                                          Receiver: received 1 bytes
snd --> [PSH, src = 0, dst = 0, seqNum = 0, data = {6}]
snd --> [PSH, src = 0, dst = 0, seqNum = 0, data = {7}]
                                          received: [PSH, src = 0, dst = 0, seqNum = 0, data = {3}]
                                          Receiver: received 1 bytes
snd --> [PSH, src = 0, dst = 0, seqNum = 0, data = {8}]
snd --> [PSH, src = 0, dst = 0, seqNum = 0, data = {9}]
Sender: transmission finished
                                          received: [PSH, src = 0, dst = 0, seqNum = 0, data = {4}]
                                          Receiver: received 1 bytes
                                          received: [PSH, src = 0, dst = 0, seqNum = 0, data = {5}]
                                          Receiver: received 1 bytes
                                          received: [PSH, src = 0, dst = 0, seqNum = 0, data = {6}]
                                          Receiver: received 1 bytes
                                          received: [PSH, src = 0, dst = 0, seqNum = 0, data = {7}]
                                          Receiver: received 1 bytes
                                          received: [PSH, src = 0, dst = 0, seqNum = 0, data = {8}]
                                          Receiver: received 1 bytes
                                          received: [PSH, src = 0, dst = 0, seqNum = 0, data = {9}]
                                          Receiver: received 1 bytes
                                          Receiver: reception finished
BUILD SUCCESSFUL (total time: 4 seconds)
```

Pràctica 3

Fragmentació de les dades i encapsulament en segments

Continuant amb l'arquitectura de protocol introduïda a la Pràctica 1 (figura1.3), tal i com es va veure, és una arquitectura dividida en tres capes o nivells: nivell de xarxa, nivell de transport i nivell d'aplicació. En aquesta pràctica ens centrarem en els següents aspectes:

- A l'extrem emissor, en la divisió de les dades d'aplicació en fragments i el seu encapsulament en segments.
- A l'extrem receptor, en l'entrega de dades al nivell d'aplicació des del nivell de transport mitjançant la recuperació dels fragments de dades continguts en cadascun dels segments rebuts.
- En el nivell de xarxa, afegirem la possibilitat de que una fracció dels paquets enviats pel canal es perdin. A més, es tindrà en compte la restricció de la mida màxima dels paquets a transmetre per la xarxa.

3.1 Nivell de transport. Transmissió de segments

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

Volem desenvolupar la pràctica de la forma més didàctica possible, per això, en l'esquema de treball proposat, la comunicació és simplex, on només l'emissor, que anomenem `Sender`, envia dades d'aplicació al receptor, que anomenem `Receiver`. Com ja s'ha vist a la Pràctica 1, el protocol de transport s'implementa mitjançant les classes `TSocketSend` i `TSocketRecv`, classes que hereten de la classe `TSocket_base` (reviseu aquesta darrera classe per veure quins atributs i funcionalitats ofereix).

3.1.1 L'extrem emissor, `TSocketSend`

Acotar superiorment la mida dels segments del nivell de transport millora l'eficiència de qualsevol comunicació perquè evita la fragmentació del segment en varis paquets del nivell de xarxa. Els paquets de xarxa no poden ser fragmentats pel nivell d'enllaç, ja que una trama de la capa d'enllaç ha de contenir sempre un paquet de xarxa sencer. En aquesta part de la pràctica és on utilitzarem el mètode de la interfície `SimNet`, `int getMTU()`, que retorna la mida màxima del camp de dades d'una trama del nivell d'enllaç, aquest és el valor clau a tenir en compte perquè no es produeixi fragmentació dels segments del nivell de transport. Les dades a incloure en un segment han de ser tal que afegint les capçaleres de transport i xarxa, el paquet de xarxa resultant càpiga al camp de dades de la trama d'enllaç. Es demana modificar el mètode corresponent a la classe `SimNet_Monitor` per a proveir aquesta informació de mida màxima (el valor real el trobareu a `Const.java`).

El protocol en l'extrem emissor disposa del mètode:

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

implementat a la Pràctica 1. El següent exercici proposa tornar a implementar aquest mètode de manera que eviti la fragmentació del segment al nivell inferior. Ara s'agafarà la seqüència de bytes provinents de les dades de l'aplicació i es fragmentarà i encapsularà en **un o més** `TCPSegments` de mida màxima condicionada pel valor MTU. Finalment s'enviaran el/s segment/s pel nivell de xarxa.

Exercici 13 [practica3.TSocketSend]

Implementar la classe `TSocketSend`, afegint-hi el procés de fragmentació. Inicialment, per a fer proves, es pot posar un valor petit a l'atribut `MSS`. Es pot fer un test usant les classes de la Pràctica 2 i veure que no hi ha errors (o esperar a tenir-ho tot implementat al final d'aquesta pràctica).

```
public class TSocketSend extends TSocket_base {

  protected int MSS;        // Maximum Segment Size

  public TSocketSend(SimNet net) {
    super(net);
    MSS = net.getMTU() - Const.IP_HEADER - Const.TCP_HEADER;
  }

  @Override
  public void sendData(byte[] data, int offset, int length) {
    // Amb l'ajuda del mètode segmentize,
    // dividir la seqüència de bytes de dades
    // en fragments de mida màxima MSS
    // i enviar-los encapsulats en segments per la xarxa
    //...
  }

  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.
    //...
  }
}
```

Al final de l'enunciat d'aquesta pràctica es mostra una possible traça d'execució.

3.2 L'extrem receptor, TSocketRecv

La classe que implementa el nivell de transport en l'extrem receptor és `TSocketRecv`. A diferència de la Pràctica 1, la classe `TSocketRecv` conté un objecte de tipus `Thread` que executa la següent tasca:

```
class ReceiverTask extends Thread {

  @Override
  public void run() {
    while (true) {
      TCPSegment rseg = network.receive();
      processReceivedSegment(rseg);
    }
  }
}
```

Les accions de la tasca són autoexplicatives, agafa un segment provinent 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 través d'una cua `CircularQueue` de `TCPSegments` que anomenem `rcvQueue`. La sincronització entre threads per a accedir a aquesta cua es realitza mitjançant l'eina de monitors amb els objectes (locks/variables de condició) definits a la classe `TSocket_base`. En concret, la sincronització consisteix en:

- mentre `rcvQueue` sigui buida, el fil que invoca `receiveData` s'haurà d'esperar. Aquest fil podrà continuar en el moment que un segment rebut pel canal sigui introduït a `rcvQueue`.
- A `processReceivedSegment(TCPSegment rseg)` es posa el segment rebut a la cua `rcvQueue`, però si la cua `rcvQueue` està plena, aquest segment es descarta. Noteu que això implica que el mètode no és bloquejant.

Aquesta classe disposa d'un únic mètode public:

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

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

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

per a 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`.

Exercici 14 [practica3.TSocketRecv]

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 i veure que no hi ha errors.

```
public class TSocketRecv extends TSocket_base {

  protected Thread thread;
  protected CircularQueue<TCPSegment> rcvQueue;
  protected int rcvSegConsumedBytes;

  public TSocketRecv(SimNet net) {
    super(net);
    rcvQueue = new CircularQueue<>(Const.RCV_QUEUE_SIZE);
    rcvSegConsumedBytes = 0;
    new ReceiverTask().start();
  }


  @Override
  public int receiveData(byte[] buf, int offset, int length) {
    lock.lock();
    try {
      // wait until receive queue is not empty
        ...
```

```
        // fill buf with bytes from segments in rcvQueue
        // Hint : use consumeSegment !
          ...
      } finally {
        lock.unlock();
      }
    }

    protected int consumeSegment(byte[] buf, int offset, int length) {
      TCPSegment seg = rcvQueue.peekFirst();
      int a_agafar = Math.min(length, seg.getDataLength() - rcvSegConsumedBytes);
      System.arraycopy(seg.getData(), rcvSegConsumedBytes, buf, offset, a_agafar);
      rcvSegConsumedBytes += a_agafar;
      if (rcvSegConsumedBytes == seg.getDataLength()) {
        rcvQueue.get();
        rcvSegConsumedBytes = 0;
      }
      return a_agafar;
    }

    @Override
    public void processReceivedSegment(TCPSegment rseg) {
      lock.lock();
      try {
        //...
      } finally {
        lock.unlock();
      }
    }

    class ReceiverTask extends Thread {

      @Override
      public void run() {
        while (true) {
          TCPSegment rseg = network.receive();
          processReceivedSegment(rseg);
        }
      }
    }
}
```

3.3 Nivell de xarxa. Modelat d'una xarxa amb pèrdues

El nivell de xarxa és l'encarregat d'enrutar paquets en una xarxa de conmutació de paquets. Com que el nostre objectiu és dissenyar el nivell de transport, per a nosaltres el nivell de xarxa és una *caixa negra* que disposa de dos mètodes `send(TCPSegment seg)` i `TCPSegment receive()`, per a enviar i rebre segments respectivament.

Per a implementar un model de xarxa *ideal*, és a dir, que no introdueix errors, no perd segments, ni els reordena, s'ha utilitzat la classe `SimNet_Monitor` de la Pràctica 2.

Recordeu que el mètode `send(TCPSegment seg)` és bloquejant si en el moment d'enviar un `TCPSegment` el canal està *al seu nivell màxim d'ocupació*. El mètode `TCPSegment receive()` és bloquejant mentre no s'hagin rebut segments de dades en l'extrem receptor del canal.

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

Exercici 15 [practica3.SimNet_Loss]

Per a modelar pèrdues, s'afegeix el següent constructor a la classe `SimNet_Loss`:

```
public SimNet_Loss(double lossRate) {
    this.lossRate = lossRate;
    rand = new Random(Const.SEED);
    log = Log.getLog();
}
```

El paràmetre `lossRate` 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 `lossRate` el segment es perd, sinó, el segment es transmet.

3.4 Nivell d'aplicació

El nivell d'aplicació es dóna tot implementat.

Exercici 16 [util]

Es demana:

1. Entendre què fa el mètode `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. Fer diferents proves canviant els valors dels atributs `sendNum`, `sendSize`, `sendInterval` i `recvBuf`, `recvInterval` per tal que es perdin dades.

Una traça d'execució d'aquesta pràctica, sense errors a la xarxa, ha de ser equivalent a la següent:

```
run:
snd --> [PSH, src = 0, dst = 0, seqNum = 0, payload = 1460]
snd --> [PSH, src = 0, dst = 0, seqNum = 0, payload = 1460]
snd --> [PSH, src = 0, dst = 0, seqNum = 0, payload = 80]
                              received: [PSH, src = 0, dst = 0, seqNum = 0, payload = 1460]
                              received: [PSH, src = 0, dst = 0, seqNum = 0, payload = 1460]
                              received: [PSH, src = 0, dst = 0, seqNum = 0, payload = 80]
                              Receiver: received 2000 bytes
snd --> [PSH, src = 0, dst = 0, seqNum = 0, payload = 1460]
snd --> [PSH, src = 0, dst = 0, seqNum = 0, payload = 1460]
snd --> [PSH, src = 0, dst = 0, seqNum = 0, payload = 80]
                              received: [PSH, src = 0, dst = 0, seqNum = 0, payload = 1460]
                              received: [PSH, src = 0, dst = 0, seqNum = 0, payload = 1460]
                              received: [PSH, src = 0, dst = 0, seqNum = 0, payload = 80]
                              Receiver: received 2000 bytes
                              Receiver: received 2000 bytes
snd --> [PSH, src = 0, dst = 0, seqNum = 0, payload = 1460]
snd --> [PSH, src = 0, dst = 0, seqNum = 0, payload = 1460]
snd --> [PSH, src = 0, dst = 0, seqNum = 0, payload = 80]
                              received: [PSH, src = 0, dst = 0, seqNum = 0, payload = 1460]
                              received: [PSH, src = 0, dst = 0, seqNum = 0, payload = 1460]
                              received: [PSH, src = 0, dst = 0, seqNum = 0, payload = 80]
                              Receiver: received 2000 bytes
                              Receiver: received 1000 bytes
snd --> [PSH, src = 0, dst = 0, seqNum = 0, payload = 1460]
snd --> [PSH, src = 0, dst = 0, seqNum = 0, payload = 1460]
snd --> [PSH, src = 0, dst = 0, seqNum = 0, payload = 80]
                              received: [PSH, src = 0, dst = 0, seqNum = 0, payload = 1460]
                              received: [PSH, src = 0, dst = 0, seqNum = 0, payload = 1460]
                              received: [PSH, src = 0, dst = 0, seqNum = 0, payload = 80]
                              Receiver: received 2000 bytes
```

```
                                Receiver: received 1000 bytes
snd --> [PSH, src = 0, dst = 0, seqNum = 0, payload = 1460]
snd --> [PSH, src = 0, dst = 0, seqNum = 0, payload = 1460]
snd --> [PSH, src = 0, dst = 0, seqNum = 0, payload = 80]
                                received: [PSH, src = 0, dst = 0, seqNum = 0, payload = 1460]
Sender: transmission finished
                                received: [PSH, src = 0, dst = 0, seqNum = 0, payload = 1460]
                                received: [PSH, src = 0, dst = 0, seqNum = 0, payload = 80]
                                Receiver: received 2000 bytes
                                Receiver: received 1000 bytes
                                Receiver: reception finished
BUILD SUCCESSFUL (total time: 1 second)
```

Pràctica 4

Multiplexat / demultiplexat

En aquesta pràctica s'implementa la funcionalitat de multiplexat/demultiplexat de les comunicacions a la capa de transport. A diferència de la pràctica anterior, on només es treballa amb una única comunicació de transport entre dos nodes concrets, ara es vol treballar alhora amb més d'una comunicació de transport entre aquests dos nodes i per tant s'han de poder identificar les diferents comunicacions.

Per aconseguir aquest objectiu s'introdueix el concepte de **port** associat a una comunicació, així, cada comunicació queda completament identificada mitjançant la quàdrupla:

`<IP_origen, port_origen, IP_destí, port_destí>`.

Comunicacions establertes entre els mateixos dos nodes extrems tindran les mateixes adreces `IP_origen` i `IP_destí`. Fins i tot, més d'una comunicació pot tenir el mateix `port_origen` o el mateix `port_destí` (aquest últim és el cas habitual quan s'està accedint al mateix servidor web des de diferents navegadors). Però dues comunicacions establertes entre els mateixos dos nodes, mai podran tenir simultàniament els mateixos `port_origen` i `port_destí`, doncs serien indistingibles perquè les quàdruples de totes dues serien idèntiques.

Conseqüentment, el demultiplexat dels paquets IP que arriben a destinació, per a lliurar el contingut d'aquests (segment de transport) a la comunicació de transport corresponent, es fa en base a la comparació de la quàdrupla definida anteriorment i obtinguda del paquet arribat, amb totes les quàdruples guardades en local al node de destí, quàdruples corresponents a totes les comunicacions de transport obertes en aquell moment.

En el nostre cas en particular, no fem servir adreces IP per a identificar els nodes, ja que la pràctica es desenvolupa mitjançant l'ús d'un canal simulat d'enviament de segments de transport entre dos nodes únics i predefinits a priori. Per això a la pràctica es demana el demultiplexat dels segments de transport que arriben pel canal simulat, pertanyents a comunicacions diferents, fent servir únicament la dupla: `<port_origen, port_destí>`.

A la pràctica anterior només es treballava amb una única comunicació; i per tant, tota la funcionalitat de la capa de transport quedava recollida dins de la classe *TSocket*, en les seves dues modalitats: `TSocketSend` i `TSocketRecv`. Ara, amb la necessitat de multiplexar diverses comunicacions de transport entre dos nodes, es diferencien els conceptes de **socket** i de **protocol**. El **socket** és propi d'una comunicació de transport en particular, mentre que el **protocol** ha de gestionar l'operativa conjunta de més d'una comunicació de transport oberta al mateix node. La implementació de la classe `Protocol.java` haurà de gestionar les llistes de sockets oberts, implementant la funcionalitat de multiplexat/demultiplexat de les comunicacions de transport.

Exercici 17 [practica4.Protocol]

Analitzeu la classe `Protocol_base` del paquet `util`. Primer noteu que la tasca de recepció, `ReceiverTask`, ara s'ha traslladat del socket a aquesta classe, ja que és a través del protocol, i amb el procés de demultiplexat, que els segments es distribueixen als sockets destinataris.

Per a poder realitzar aquest procés de demultiplexat, el protocol necessita conèixer els sockets oberts, per això hi ha definida la llista `activeSockets`, que conté referències a tots els sockets actius, i per tant, possibles candidats a rebre segments.

El procés de demultiplexat es realitza a la classe `Protocol.java` i és la part de codi que s'ha de completar. El mètode `protected void ipInput(TCPSegment seg)` rep un segment provinent de la xarxa, i a partir dels ports del segment, i invocant el mètode `getMatchingTSocket(...)`, dedueix el socket (en cas d'existir) destinatari a qui es lliurarà el segment invocant el mètode `processReceivedSegment(...)`.

El mètode `TSocket_base getMatchingTSocket(int localPort, int remotePort)` rep dos ports i fa una cerca a la llista de sockets actius per a determinar si hi ha coincidència amb els ports d'algun dels sockets oberts.

Exercici 18 [practica4.TSocket]

En les pràctiques anteriors s'han considerat dos tipus de sockets, un amb rol d'emissor i l'altre amb rol de receptor. En aquesta pràctica i posteriors ja només es considera una única classe `TSocket.java` que assumeix tots dos rols.

De moment a la nova classe `TSocket.java` no s'hi afegeix quasi cap funcionalitat que no existís en els sockets previs, per tant, la implementació només consisteix en fer *copy* i *paste* del codi de les pràctiques anteriors. L'únic nou que s'ha de tenir en compte és:

- tots els segments que s'envien porten els ports d'origen i destí,
- per a començar a practicar l'ús de segments de control es demana que per a cada segment de dades acceptat, s'enviï un segment de control de tornada amb el flag d'`ack` actiu. En aquesta pràctica el receptor dels segments d'`ack` no en farà res d'aquests (els ignora).

Una possible traça d'execució d'aquesta pràctica, sense errors a la xarxa, ha de ser equivalent a la següent:

```
run:
                                    received: [PSH, src = 20, dst = 80, seqNum = 0, payload = 1460]
                                    received: [PSH, src = 20, dst = 80, seqNum = 0, payload = 1460]
  received: [ACK, src = 80, dst = 20, ackNum = 0, wnd = 0]
  received: [ACK, src = 80, dst = 20, ackNum = 0, wnd = 0]
                                    received: [PSH, src = 20, dst = 80, seqNum = 0, payload = 80]
                                    received: [PSH, src = 30, dst = 80, seqNum = 0, payload = 1460]
  received: [ACK, src = 80, dst = 20, ackNum = 0, wnd = 0]
  received: [ACK, src = 80, dst = 30, ackNum = 0, wnd = 0]
                                    received: [PSH, src = 30, dst = 80, seqNum = 0, payload = 1460]
                                    received: [PSH, src = 30, dst = 80, seqNum = 0, payload = 80]
  received: [ACK, src = 80, dst = 30, ackNum = 0, wnd = 0]
  received: [ACK, src = 80, dst = 30, ackNum = 0, wnd = 0]
                                    Receiver: received 2000 bytes
                                    Receiver: received 2000 bytes
                                    received: [PSH, src = 20, dst = 80, seqNum = 0, payload = 1460]
                                   received: [PSH, src = 30, dst = 80, seqNum = 0, payload = 1460]
                                    received: [PSH, src = 30, dst = 80, seqNum = 0, payload = 1460]
  received: [ACK, src = 80, dst = 20, ackNum = 0, wnd = 0]
  received: [ACK, src = 80, dst = 30, ackNum = 0, wnd = 0]
  received: [ACK, src = 80, dst = 30, ackNum = 0, wnd = 0]
                                    received: [PSH, src = 30, dst = 80, seqNum = 0, payload = 80]
                                    received: [PSH, src = 20, dst = 80, seqNum = 0, payload = 1460]
  received: [ACK, src = 80, dst = 30, ackNum = 0, wnd = 0]
  received: [ACK, src = 80, dst = 20, ackNum = 0, wnd = 0]
                                    Receiver: received 2000 bytes
                                    received: [PSH, src = 20, dst = 80, seqNum = 0, payload = 80]
  received: [ACK, src = 80, dst = 20, ackNum = 0, wnd = 0]
                                    Receiver: received 2000 bytes
                                    Receiver: received 2000 bytes
                                    Receiver: received 2000 bytes
                                    received: [PSH, src = 30, dst = 80, seqNum = 0, payload = 1460]
                                    received: [PSH, src = 30, dst = 80, seqNum = 0, payload = 1460]
  received: [ACK, src = 80, dst = 30, ackNum = 0, wnd = 0]
  received: [ACK, src = 80, dst = 30, ackNum = 0, wnd = 0]
                                    received: [PSH, src = 20, dst = 80, seqNum = 0, payload = 1460]
                                    received: [PSH, src = 20, dst = 80, seqNum = 0, payload = 1460]
                                    Receiver: received 2000 bytes
  received: [ACK, src = 80, dst = 20, ackNum = 0, wnd = 0]
  received: [ACK, src = 80, dst = 20, ackNum = 0, wnd = 0]
```

```
                                    received: [PSH, src = 20, dst = 80, seqNum = 0, payload = 80]
                                    received: [PSH, src = 30, dst = 80, seqNum = 0, payload = 80]
  received: [ACK, src = 80, dst = 20, ackNum = 0, wnd = 0]
  received: [ACK, src = 80, dst = 30, ackNum = 0, wnd = 0]
                                    Receiver: received 1000 bytes
                                    Receiver: received 2000 bytes
                                    received: [PSH, src = 30, dst = 80, seqNum = 0, payload = 1460]
                                    Receiver: received 1000 bytes
                                    received: [PSH, src = 30, dst = 80, seqNum = 0, payload = 1460]
  received: [ACK, src = 80, dst = 30, ackNum = 0, wnd = 0]
  received: [ACK, src = 80, dst = 30, ackNum = 0, wnd = 0]
                                    received: [PSH, src = 20, dst = 80, seqNum = 0, payload = 1460]
                                    received: [PSH, src = 20, dst = 80, seqNum = 0, payload = 1460]
                                    received: [PSH, src = 20, dst = 80, seqNum = 0, payload = 80]
                                    Receiver: received 2000 bytes
  received: [ACK, src = 80, dst = 20, ackNum = 0, wnd = 0]
  received: [ACK, src = 80, dst = 20, ackNum = 0, wnd = 0]
  received: [ACK, src = 80, dst = 20, ackNum = 0, wnd = 0]
                                    received: [PSH, src = 30, dst = 80, seqNum = 0, payload = 80]
  received: [ACK, src = 80, dst = 30, ackNum = 0, wnd = 0]
                                    Receiver: received 1000 bytes
                                    Receiver: received 2000 bytes
Sender: transmission finished
                                    received: [PSH, src = 30, dst = 80, seqNum = 0, payload = 1460]
Sender: transmission finished
                                    received: [PSH, src = 30, dst = 80, seqNum = 0, payload = 1460]
  received: [ACK, src = 80, dst = 30, ackNum = 0, wnd = 0]
  received: [ACK, src = 80, dst = 30, ackNum = 0, wnd = 0]
                                    Receiver: received 1000 bytes
                                    received: [PSH, src = 30, dst = 80, seqNum = 0, payload = 80]
                                    received: [PSH, src = 20, dst = 80, seqNum = 0, payload = 1460]
  received: [ACK, src = 80, dst = 30, ackNum = 0, wnd = 0]
                                    Receiver: received 2000 bytes
                                    received: [PSH, src = 20, dst = 80, seqNum = 0, payload = 1460]
  received: [ACK, src = 80, dst = 20, ackNum = 0, wnd = 0]
                                    received: [PSH, src = 20, dst = 80, seqNum = 0, payload = 80]
  received: [ACK, src = 80, dst = 20, ackNum = 0, wnd = 0]
  received: [ACK, src = 80, dst = 20, ackNum = 0, wnd = 0]
                                    Receiver: received 1000 bytes
                                    Receiver: reception finished
                                    Receiver: received 2000 bytes
                                    Receiver: received 1000 bytes
                                    Receiver: reception finished
BUILD SUCCESSFUL (total time: 3 seconds)
```

Pràctica 5

Mecanismes Stop & Wait, control d'errors i pèrdues (ARQ) i control de flux

5.1 Introducció

A les pràctiques anteriors s'ha suposat que el nivell de xarxa no introduïa ni errors ni pèrdues a la transmissió dels segments. Però en un funcionament real, la capa IP pot perdre paquets i introduir errors en els *payloads* que contenen els segments de transport.

En aquesta pràctica, es programaran diversos mecanismes de control amb l'objectiu d'implementar un canal de comunicació fiable, i així dur a terme el tractament del problema dels errors de transmissió i les pèrdues de segments. Però també es considerarà la freqüencia amb la que s'injectaran segments a la xarxa i la dependència envers la velocitat de consum de dades per part del receptor. En particular, s'implementaran els mecanismes de:

- Injecció de segments a la xarxa anomenat Stop & Wait.
- Control d'errors i pèrdues de segments, anomenat ARQ (Automatic Repeat reQuest).
- Control de flux basat en l'ús de l'anomenada *finestra de recepció*.

5.2 Stop & Wait

En aquesta pràctica implementarem un algorisme d'injecció de segments a la xarxa anomenat Stop & Wait. L'algorisme Stop & Wait comporta:

- Cada vegada que l'emissor envia un nou segment de dades, aquest ha d'esperar a rebre del receptor el reconeixement d'acceptació d'aquest segment, abans d'enviar el següent segment de dades.
- El receptor, per la seva part, notifica a l'emissor la recepció de cada nou segment de dades, sempre i quan el pugui guardar a la cua de recepció. Aquesta notificació es realitza enviant un segment de control anomenat segment de reconeixement (segment sense dades, amb número de seqüència de reconeixement i amb el flag `ACK` activat).

Exercici 19 [Stop & Wait]

S'han de modificar els mètodes `sendData` i `processReceivedSegment` de la classe `TSocket` implementant el mecanisme Stop & Wait descrit anteriorment. Feu les proves utilitzant una xarxa sense pèrdues, ja que encara no s'ha programat el mecanisme d'ARQ.

Tota traça d'execució ha de mostrar com s'intercalen els segments de `PUSH` i els d'`ACK`:

```
run:
                            received: [PSH, src = 10, dst = 80, seqNum = 0, payload = 1460]
  received: [ACK, src = 80, dst = 10, ackNum = 1, wnd = 49]
                            received: [PSH, src = 10, dst = 80, seqNum = 1, payload = 1460]
  received: [ACK, src = 80, dst = 10, ackNum = 2, wnd = 48]
```

```
                            received: [PSH, src = 10, dst = 80, seqNum = 2, payload = 80]
received: [ACK, src = 80, dst = 10, ackNum = 3, wnd = 47]
                            Receiver: received 2000 bytes
                            received: [PSH, src = 10, dst = 80, seqNum = 3, payload = 1460]
received: [ACK, src = 80, dst = 10, ackNum = 4, wnd = 47]
                            received: [PSH, src = 10, dst = 80, seqNum = 4, payload = 1460]
received: [ACK, src = 80, dst = 10, ackNum = 5, wnd = 46]
...
```

5.3 Mecanisme ARQ

Automatic Repeat reQuest (ARQ) és un mecanisme de retransmissió del segment de dades, per manca de confirmació de la seva rebuda a destinació en un temps predeterminat. En comunicacions extrem a extrem és l'única tècnica possible per a garantir el lliurament de les dades a destinació; i per tant, per aconseguir una transferència fiable de la informació sobre un canal no fiable. Consisteix en la implementació del següent mecanisme:

- Els missatges (segments en el nostre cas) incorporen un codi de detecció d'error (CRC o *checksum*) i un número de seqüència, afegits per l'emissor, i que permeten al receptor determinar si el segment rebut no conté errors i és l'esperat.
- Quan el receptor rep el segment esperat i sense errors, si el pot guardar a la cua de recepció (hi ha espai lliure), aquest respon a l'emissor amb un segment de reconeixement (confirmació d'arribada a destinació i acceptació del segment de dades).
- Si l'emissor no rep el reconeixement de segment abans d'un temps determinat RTO (Retransmission Time Out), aquest retransmetrà el segment de dades de nou. La retransmissió es repetirà fins que l'emissor rebi el reconeixement del segment de dades per part del receptor.
- Un cas particular de retransmissió de dades es dóna com a conseqüència de la pèrdua del segment de reconeixement. En aquest cas es produirà una duplicitat de dades en recepció com a resultat de la inevitable retransmissió. Això és el que fa necessari amb Stop & Wait que els segments de dades portin un **número de seqüència** per a detectar la duplicitat del segment de dades arribat al receptor, si no es perdesin segments de reconeixement fent servir Stop & Wait, no caldria numerar els segments de dades. El receptor ignorarà les dades d'un segment duplicat, és a dir, d'un segment prèviament rebut correctament. En aquest cas el receptor entendrà que el segment de reconeixement enviat prèviament no ha arribat a l'emissor i per tant l'haurà de tornar a enviar.
- Els segments de reconeixement poden no arribar, o arribar desordenats a destinació (l'emissor en aquest cas), per tant, aquests segments han d'indicar quines dades estan reconeixent, i es fa posant com a número d'Ack el número de seqüència del proper segment de dades que s'espera rebre, per tal que l'emissor pugui detectar i descartar reconeixements no esperats. D'altra banda, els segments de reconeixement no contenen número de seqüència propi, ja que mai s'ha de reconèixer la rebuda d'un segment de reconeixement, això seria una incongruència de funcionament.

5.3.1 Detecció d'errors

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

Tampoc farem la comprovació del checksum a l'extrem receptor de la capa de transport (a diferència de com es fa al TCP), suposarem que el nivell de xarxa simplement descarta els segments rebuts amb dades errònies. Per tant, la taxa de pèrdues simulada a la classe `SimNet_FullDuplex` inclou la probabilitat de rebre de la xarxa segments amb dades errònies.

5.3.2 Retransmissions

Fent servir el mecanisme ARQ explicat anteriorment, a l'emissor caldrà fer servir un temporitzador sobre el que es programarà l'execució d'una tasca futura per cada segment de dades enviat.

El mecanisme de retransmissió ARQ requereix l'ús d'un servei de temporització. En el nostre cas utilitzarem el temporitzador que proporciona Java.

A la classe `TSocket_base` es pot veure els mètodes que es proporcionen per a l'ús del temporitzador. La variable `sndRtTimer` fa referència a la futura tasca a executar pel temporitzador. La programació d'execució d'aquesta tasca es fa mitjançant els següents mètodes de `TSocket_base`:

```
protected void timeout(TCPSegment seg) {
      throw new RuntimeException("Not supported yet.");
  }

  protected void startRTO(TCPSegment seg) {
      TimerTask sndRtTimer = new TimerTask() {
          @Override
          public void run() {
              timeout(seg);
          }
      };
      timerService.schedule(sndRtTimer, Const.SND_RTO);
  }
```

A la classe `TSocket` és on es sobreescriu el mètode `timeout()`, que l'únic que hauria de fer és retransmetre el segment de dades en cas de que estigui pendent de confirmació i complementariament hauria de reprogramar de nou el temporitzador.

5.3.3 Números de seqüència

Ja s'ha comentat la necessitat de numerar els segments de dades i els reconeixements. Fent-ho de forma generalista es consideraran seqüències que comencen per 0 i s'incrementen en 1 a cada enviament. Al protocol TCP, la fase d'establiment de la connexió té entre els seus propòsits acordar la numeració inicial de cada interlocutor de forma pseudo-aleatoria.

A la nostra pràctica, la variable `snd_sndNxt` de l'emissor guardarà el número de seqüència del proper segment de dades a enviar. La variable `rcv_rcvNxt` del receptor guardarà el número de seqüència del proper segment de dades que espera rebre.

Tots els segments de dades hauran de portar el corresponent número de seqüència i els segments de reconeixement el corresponent número d'ack. Per això usarem els corresponents camps de la capçalera TCP, i que podrem fixar i consultar amb els mètodes de la classe `TCPSegment`.

Exercici 20 [ARQ]

Afegiu el mecanisme de retransmissió ARQ a la classe `TSocket`. A més, afegiu el número de seqüència als segments i, evidentment, descarteu els segments que arribin amb un número de seqüència no esperat. Recordeu posar una taxa de pèrdues positiva a la xarxa. Una possible traça d'execució ha de ser equivalent a la següent:

```
...
                            received: [PSH, src = 10, dst = 80, seqNum = 4, payload = 1460]
  received: [ACK, src = 80, dst = 10, ackNum = 5, wnd = 46]
           +++++++++ SEGMENT LOST: [PSH, src = 10, dst = 80, seqNum = 5, payload = 80] +++++++++

                            Receiver: received 2000 bytes
retrans: [PSH, src = 10, dst = 80, seqNum = 5, payload = 80]
                            received: [PSH, src = 10, dst = 80, seqNum = 5, payload = 80]
           +++++++++ SEGMENT LOST: [ACK, src = 80, dst = 10, ackNum = 6, wnd = 47] +++++++++

                            Receiver: received 2000 bytes
retrans: [PSH, src = 10, dst = 80, seqNum = 5, payload = 80]
                            received: [PSH, src = 10, dst = 80, seqNum = 5, payload = 80]
  received: [ACK, src = 80, dst = 10, ackNum = 6, wnd = 50]
                            received: [PSH, src = 10, dst = 80, seqNum = 6, payload = 1460]
...
```

5.4 Control de flux

A la solució implementada fins ara el receptor envia un segment de reconeixement cada vegada que rep un segment de dades i el pot guardar a la cua de recepció. Aquest reconeixement, a més a més del que s'ha comentat en apartats anteriors, ara portarà informació de l'espai lliure que queda a la cua del receptor, l'anomenada **finestra de recepció**. Observeu que el que s'està fent en aquest cas, indicant la mida de l'espai lliure en recepció, és introduir un mecanisme de control de flux aprofitant els reconeixements del mecanisme Stop & Wait. Per a indicar la mida d'aquest espai lliure, al segment de reconeixement s'utilitza el mètode `setWnd(...)` (mitjançant el mètode `getWnd()` es pot consultar aquest valor).

Ara doncs, l'emissor, a més de les consideracions fetes en apartats anteriors, també ha de tenir en compte el valor de la finestra de recepció. Un cas particular es dona quan el valor de la finestra de recepció és zero. En aquesta situació l'extrem emissor hauria d'aturar temporalment l'emissió ordinària de segments de dades i començar a **sondejar** periòdicament l'extrem receptor, enviant un segment de dades ordinari però de només 1 byte de dades, per tal d'esbrinar si ja hi torna a haver espai lliure a la cua del receptor.

Exercici 21 [Flux]

Afegiu el valor de la finestra de recepció als segments d'ACK, i tracteu la situació de finestra zero en l'emissor, enviant periòdicament el segment de sondeig corresponent.

Per a veure que el mecanisme funciona, s'ha definit una cua de recepció de segments de grandària 2:

```
//rcv_Queue = new CircularQueue<>(Const.RCV_QUEUE_SIZE);
rcv_Queue = new CircularQueue<>(2);
```

i executant s'han d'obtenir traces equivalents a la següent:

```
run:
                            received: [PSH, src = 10, dst = 80, seqNum = 0, payload = 1460]
  received: [ACK, src = 80, dst = 10, ackNum = 1, wnd = 1]
                            received: [PSH, src = 10, dst = 80, seqNum = 1, payload = 1460]
  received: [ACK, src = 80, dst = 10, ackNum = 2, wnd = 0]
----- zero-window probe ON -----
                            Receiver: received 2000 bytes
0-wnd probe: [PSH, src = 10, dst = 80, seqNum = 2, payload = 1]
                            received: [PSH, src = 10, dst = 80, seqNum = 2, payload = 1]
  received: [ACK, src = 80, dst = 10, ackNum = 3, wnd = 0]
----- zero-window probe OFF -----
----- zero-window probe ON -----
                            Receiver: received 921 bytes
0-wnd probe: [PSH, src = 10, dst = 80, seqNum = 3, payload = 1]
                            received: [PSH, src = 10, dst = 80, seqNum = 3, payload = 1]
  received: [ACK, src = 80, dst = 10, ackNum = 4, wnd = 1]
----- zero-window probe OFF -----
                            received: [PSH, src = 10, dst = 80, seqNum = 4, payload = 78]
  received: [ACK, src = 80, dst = 10, ackNum = 5, wnd = 0]
                            Receiver: received 79 bytes
...
```

Pràctica 6

Finestra de congestió de mida superior a la unitat

Aquesta pràctica és la continuació de la pràctica anterior, afegint-hi una finestra de congestió de mida superior a 1, és a dir, ja no s'utilitza el mecanisme d'injecció Stop & Wait. Ara l'emissor, a l'hora d'enviar segments, està limitat pel mínim (`snd_minWnd`) entre dues finestres: la finestra de congestió `snd_cngWnd` i la finestra de recepció `snd_rcvWnd`. El cas particular de finestra de recepció zero es tracta igual que en la pràctica anterior.

Exercici 22 [TSocket]

Implementeu a la classe `TSocket` tots els mecanismes de la pràctica anterior excepte el control de congestió Stop & Wait, aquest darrer s'ha de substituir per un control de congestió que permeti tenir més d'un segment de dades pendent de reconeixement en treballar amb una finestra de congestió de mida superior a la unitat.

Com a conseqüència d'aquesta característica, s'han de guardar temporalment els segments de dades rebuts fora d'ordre, per introduir-los posteriorment (quan es pugui) en ordre a la cua circular del receptor, per això es farà servir un `Map<Integer, TCPSegment> out_of_order_segs` que guarda temporalment en una taula: els números de seqüència i segments associats dels segments rebuts fora d'ordre. Una traça d'execució ha de ser equivalent a la següent:

```
run:
                                        received: [PSH, src = 10, dst = 80, seqNum = 0, payload = 10]
                                        receiver - introduit el: 0
                                        received: [PSH, src = 10, dst = 80, seqNum = 1, payload = 10]
                                        receiver - introduit el: 1
  received: [ACK, src = 80, dst = 10, ackNum = 1, wnd = 49]
                                        received: [PSH, src = 10, dst = 80, seqNum = 2, payload = 10]
                                        receiver - introduit el: 2
  received: [ACK, src = 80, dst = 10, ackNum = 2, wnd = 48]
  received: [ACK, src = 80, dst = 10, ackNum = 3, wnd = 47]
         +++++++++ SEGMENT LOST: [PSH, src = 10, dst = 80, seqNum = 5, payload = 10] +++++++++

                                        received: [PSH, src = 10, dst = 80, seqNum = 3, payload = 10]
                                        receiver - introduit el: 3
                                        received: [PSH, src = 10, dst = 80, seqNum = 4, payload = 10]
                                        receiver - introduit el: 4
  received: [ACK, src = 80, dst = 10, ackNum = 4, wnd = 46]
  received: [ACK, src = 80, dst = 10, ackNum = 5, wnd = 45]
                                        received: [PSH, src = 10, dst = 80, seqNum = 6, payload = 10]
                                        receiver - guardat fora d'ordre: 6
          +++++++++ SEGMENT LOST: [ACK, src = 80, dst = 10, ackNum = 5, wnd = 45] +++++++++

                                        received: [PSH, src = 10, dst = 80, seqNum = 7, payload = 10]
                                        receiver - guardat fora d'ordre: 7
  received: [ACK, src = 80, dst = 10, ackNum = 5, wnd = 45]
                                        Receiver: received 50 bytes
retrans: [PSH, src = 10, dst = 80, seqNum = 5, payload = 10]
                                        received: [PSH, src = 10, dst = 80, seqNum = 5, payload = 10]
                                        receiver - introduit el: 5
```

```
                                    receiver - introduit en ordre: 6
                                    receiver - introduit en ordre: 7
received: [ACK, src = 80, dst = 10, ackNum = 8, wnd = 47]
                                    received: [PSH, src = 10, dst = 80, seqNum = 8, payload = 10]
                                    receiver - introduit el: 8
                                    received: [PSH, src = 10, dst = 80, seqNum = 9, payload = 10]
                                    receiver - introduit el: 9
received: [ACK, src = 80, dst = 10, ackNum = 9, wnd = 46]
...
```

Pràctica 7

Establiment i alliberament de connexions

7.1 Introducció

En aquesta pràctica s'estudien les fases d'establiment i alliberament d'una connexió per als protocols de transport orientats a connexió.

De manera informal podríem dir que un protocol orientat a connexió és aquell en el qual, abans d'intercanviar dades d'aplicació, s'estableix una connexió lògica entre els dos extrems comunicants. Normalment, un dels extrems comença enviant una petició per a obrir la connexió, s'acostuma a dir que això es fa des de l'element que juga el rol de **client**, a la qual cosa l'altre extrem respon, aquest últim jugant el rol de **servidor**. En aquest intercanvi inicial de missatges, anomenat **handshake**, es passa informació de control per a determinar sí i com s'ha d'establir la connexió. Si el handshake té èxit, llavors es produeix un canvi de fase i 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 de forma individual tanca la connexió del seu costat (no té més dades a transmetre) enviant un segment especial.

Cal destacar que en aquesta pràctica ens centrarem en les fases d'establiment i alliberament de la connexió, i per això, no hi haurà intercanvi de dades d'aplicació.

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

En general, en els protocols orientats a connexió, les fases d'establiment i alliberament s'implementen mitjançant una màquina d'estats. Cadascun dels extrems transita entre diferents estats com a resultat d'enviar o rebre un determinat missatge producte de la *conversa* que estan mantenint tots dos interlocutors. En aquesta pràctica s'ha dissenyat una màquina d'estats del protocol que, per qüestions didàctiques, és una simplificació del diagrama d'estats del protocol TCP. Aquest diagrama es mostra a la Figura 7.1.

7.2 Establiment d'una connexió

A l'hora d'establir una connexió, com s'ha comentat abans, els dos extrems tenen papers diferents. Un dels dos extrems, que anomenarem extrem *passiu* (servidor), espera a que l'altre extrem, l'extrem *actiu* (client), prengui la iniciativa i demani establir la connexió. A l'extrem passiu es fa servir la classe `TServerSocket` i a l'actiu es fa servir la classe `TSocket`. Normalment, en un mateix host hi poden haver extrems passius i actius alhora, és per això que en la classe `Protocol_base` hi ha definides dues cues:

```
public abstract class Protocol_base {

  //...
  protected ArrayList<TSocket_base> listenSockets;
  protected ArrayList<TSocket_base> activeSockets;
  //...
}
```

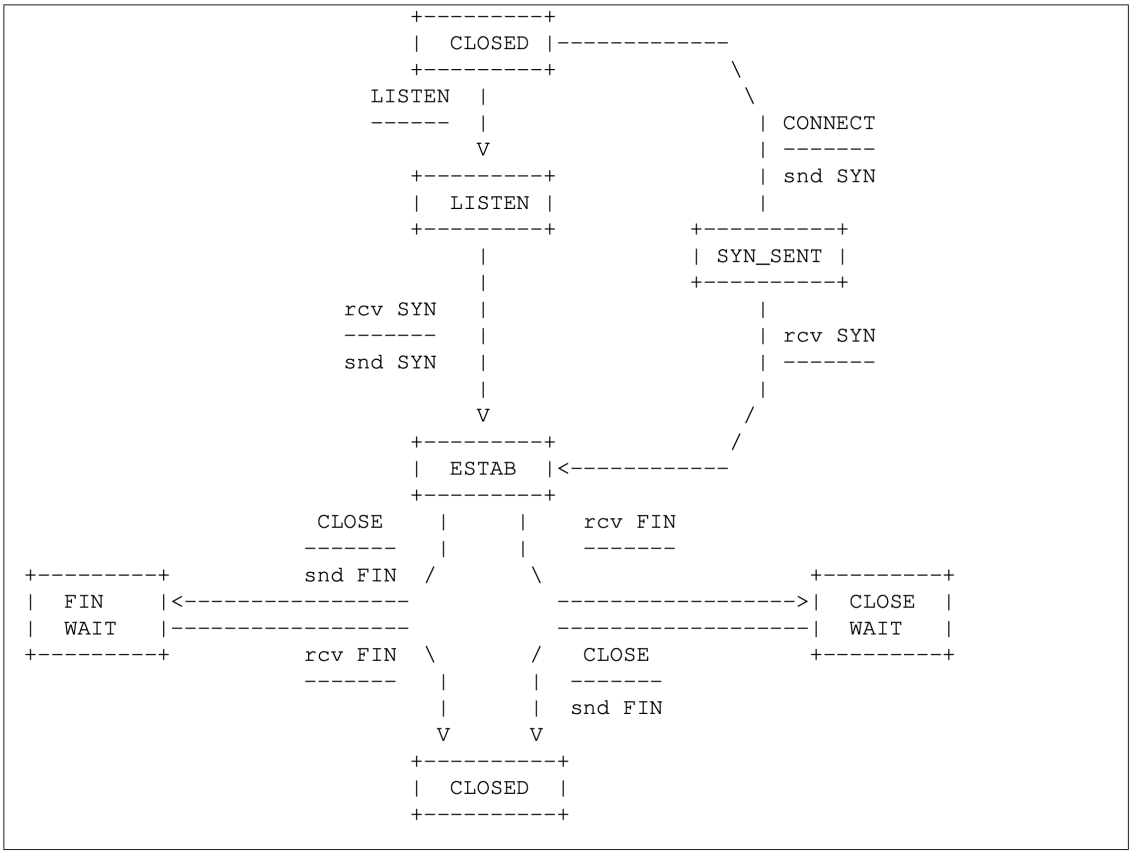

Figura 7.1: Diagrama d'estats del protocol de transport.

7.2.1 L'extrem passiu

L'extrem passiu, mitjançant la classe `TServerSocket`, utilitza els mètodes `listen()` i `accept()`. En el mètode `run()` de la classe `Test.java` del host servidor es pot observar com s'utilitza el mètode `accept()`, que és un mètode bloquejant en espera d'una nova petició d'establiment de connexió, i retorna una instància de la classe `TSocket` resultat de l'establiment de la connexió amb l'extrem remot.

```
@Override
public void run() {
  log.printBLUE("\t\t\t\t\t\t\tServer started");
  TServerSocket serverSocket = new TServerSocket(proto, HostSrv.PORT);
  for (int i = 0; i < 2; i++) {
    TSocket sc = serverSocket.accept();
    new Thread(new Worker(sc)).start();
  }
}
```

7.2.2 L'extrem actiu

Com hem dit, l'extrem que inicia la petició d'establiment de connexió s'acostuma a anomenar extrem actiu. A la classe `Test.java` es pot veure com s'utilitza la classe `TSocket` per a establir la connexió:

```
public void run() {
  log.printBLUE("Client started");

  TSocket sc = new TSocket(proto, localPort, HostSrv.PORT);
  sc.connect();
```

```
      log.printBLUE("Client connected with localport: " + sc.localPort);
      try {
        Thread.sleep(5000);
      } catch (Exception e) {
        e.printStackTrace();
      }
      log.printBLUE("Client about to close from localport: " + sc.localPort);
      sc.close();
      log.printBLUE("Client closed from localport: " + sc.localPort);
    }
  }
```

7.2.3 Protocol d'establiment de connexió

El procés és el següent:

1. En el costat del client es crea un nou objecte `TSocket`.
2. S'invoca el mètode `connect()` sobre aquest nou objecte. Dins d'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) Es crea un segment de tipus `SYN` i s'envia.
   (c) Es posa el `TSocket` en estat `SYN_SENT`.
   (d) S'espera a que l'estat del `TSocket` sigui `ESTABLISHED` (això passarà quan es rebi un segment de `SYN` de l'extrem remot).
3. En el costat del servidor es crea un nou objecte `TServerSocket` i s'invoca el mètode `accept()`.
4. Quan al cantó servidor es rep un segment de `SYN` el `Protocol` decidirà quin `TServerSocket` l'ha de processar. Aquest processament consisteix en:
   (a) Es crea un nou `TSocket` per a materialitzar la nova connexió establerta amb l'extrem remot, i s'inicialitza aquest nou `TSocket` correctament (ports local i remot, i l'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 que aquesta cua no estigui buida?
   (d) Es crea un nou segment de tipus `SYN`, que és enviat pel nou `TSocket`.
5. Quan arriba un segment de tipus `SYN` a l'extrem acitu (client) el processament per part del `TSocket` (que és el que ha iniciat la petició de 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`?

Exercici 23 [Connexió]

1. Canvieu el mètode `getMatchingTSocket(...)` de la classe `Protocol`, que vàreu fer en pràctiques anteriors, de manera que busqui tant en la cua `activeTSockets` com en la cua `listenTSockets`. Per què?
2. Implementeu el mètode `accept()`. Aquest mètode espera fins que la cua `acceptQueue` deixa d'estar buida, i un cop no ho està, en treu i retorna el primer element.
3. Implementeu el mètode `connect()`.
4. Completeu el mètode `processReceivedSegment()` per als dos tipus de sockets, per als estats: `LISTEN` i `SYN_SENT`.

7.3 Alliberament de la connexió

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

Per a 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 7.1 (el `TSocket` passarà d'`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. Més tard o més d'hora també es cridarà al mètode `close()` des d'aquest extrem, de manera que els dos extrems acabaran en estat `CLOSED`.

Exercici 24 [Desconnexió]

1. Implementeu el mètode `close()`.
2. Completeu el mètode `processReceivedSegment()` per als estats: `ESTABLISHED`, `FIN_WAIT` i `CLOSE_WAIT`.
3. Intenteu entendre què fa la classe `Test` i proveu el codi realitzat. Com ja s'ha dit, en aquesta pràctica no hi ha intercanvi de dades d'aplicació.

Una possible traça:

```
run:
                                                    Server started

Client started
Client started
    sent: [SYN, src = 10, dst = 80, seqNum = 0]
    sent: [SYN, src = 20, dst = 80, seqNum = 0]
                                                    rcvd: [SYN, src = 10, dst = 80, seqNum = 0]
                                                    sent: [SYN, src = 80, dst = 10, seqNum = 0]
                                                    rcvd: [SYN, src = 20, dst = 80, seqNum = 0]
    rcvd: [SYN, src = 80, dst = 10, seqNum = 0]
                                                    sent: [SYN, src = 80, dst = 20, seqNum = 0]
Client connected with localport: 10
    rcvd: [SYN, src = 80, dst = 20, seqNum = 0]
                                                    Worker providing service to client with port: 20
                                                    Worker providing service to client with port: 10
Client connected with localport: 20
Client about to close from localport: 10
                                                    Worker about to close to client with port: 20
                                                    sent: [FIN, src = 80, dst = 20, seqNum = 0]
Client about to close from localport: 20
                                                    Worker about to close to client with port: 10
    sent: [FIN, src = 10, dst = 80, seqNum = 0]
                                                    Worker closed from client with port: 20
    rcvd: [FIN, src = 80, dst = 20, seqNum = 0]
                                                    sent: [FIN, src = 80, dst = 10, seqNum = 0]
Client closed from localport: 10
    sent: [FIN, src = 20, dst = 80, seqNum = 0]
Client closed from localport: 20
                                                    Worker closed from client with port: 10
                                                    rcvd: [FIN, src = 10, dst = 80, seqNum = 0]
    rcvd: [FIN, src = 80, dst = 10, seqNum = 0]
                                                    rcvd: [FIN, src = 20, dst = 80, seqNum = 0]
BUILD SUCCESSFUL (total time: 8 seconds)
```