Programació Concurrent: resum

Bartomeu Kane Binimelis

Tardor del 1712

Índex

1 Introducció

1.1 Definicions inicials

Definició 1. Un *procés* és un fil d'execució.

Definició 2. El *programa* és el codi o algoritme. Un programa pot donar lloc a més d'un procés.

Definició 3. L'*estat* d'un procés queda definit pel codi del procés, per les variables explícites definides al codi i per les variables implícites necessàries per a l'execució: valores de registres, comptadors d'instruccions, etc.

Definició 4. Diem que una *acció* és la responsable d'un canvi d'estat.

Definició 5. Diem que una *acció atòmica* és aquella que pasa d'un estat a un altre de forma indivisible.

1.2 Zona crítica

Definició 6. La *zona crítica* són les parts del codi que s'han d'executar complint les següents propietats:

- Exclusió mútua: un únic procés pot executar simultàneament la zona crítica.
- Absència de bloqueig: sempre ha d'haver un procés que pugui entrar a la zona crítica.
- (Addicional) Tot procés ha d'accedir eventualment a la zona crítica.
- (Addicional) S'han d'evitar esperes innecesàries.

2 Alguns comentaris sobre Java

2.1 Variables

Per tal de definir variables utilitzarem:

```
[modificadors] <tipus> <nom_atribut>
```

On tots els modificadors són opcionals. Poden ser:

- El primer modificador pot ser *public* o *private*. *public* vol dir que la variable és accesible des d'una altra classe. Si és *private* només és accesible des de la mateixa classe.
- El segon és *static*: provoca que la variable sigui compartida per totes les instàncies d'aquell objecte.
- El tercer és *final*, i vol dir que la variable només s'inicialitzarà una vegada i no canviarà de valor.

Per crear un objecte utilitem:

```
<nom classe> <nom_objecte> = new <nom classe>
```

2.2 Mètodes

Per crear un mètode utilitzem:

```
[modificadors] <tipus_retorn> nom_metode (<tipus_parametre> nom)
```

Observació. Notem que els paràmetres es pasen per valor, però en el cas dels objectes, aquest valor és la seva posició de memòria. Per tant, és equivalent a pasar per referència.

Observació. En la implementació d'un mètode podem utilitzar la variable *this* que apunta al propi objecte.

2.3 Herència i polimorfismes

La herència consisteix en que a partir d'una classe ja existent, Java permet crear-ne una d'altre *extenent* aquesta. La sintaxi és:

```
public class B extends A {
/* ... */
}
```

Observació. Al extendre una classe, no podrem cridar als mètodes privats de la original (ni sobreescriure'ls!).

El polimorfisme consisteix en que per tal de crear un objecte de la classe B, podem fer-ho amb una variable del tipus *pare*. Per exemple, es correcte:

```
A a = new B();
```

Observació. En aquest cas només es podran utilitzar els mètodes que estiguin definits en la classe A, però executant el codi escrit a la classe B (si aquest codi s'ha sobreescrit).

Observació. Tota classe de Java hereta de la classe *Object*. Per tant, podem crear un Array de *Object* i guardar els objectes que vulguem.

2.4 La classe Thread

La *classe Thread* és aquella que extrendrem per tal de crear procesos. Hi trobem els següents mètodes:

```
1 public void run()        //codi que executara el fil
2 public void start()      //crea el nou fil, executant el seu metode run
3 public static void sleep(int ms)    //adorm el proces ms milisegons
4 public void join()       //atura el proces que invoca el join fins que
5                          //  acaba el fil al qual pertany el join
```

2.5 Creació d'arrays

Per tal de crear arrays d'un objecte concret, escriurem:

```
1 My_class[] nom = new My_class[10]    //array amb 10 elements
```

Observació. Posteriorment caldrà crear cada objecte a cada posició.

També podem pensar en utilitzar una altra estructura que ens dóna Java, la classe *ArrayList*. Permeten posar-hi objectes i tenen un conjunt de funcions ja implementades prou útils. La declaració i les funcions son:

```
1 Arraylist a = new Arraylist();
2 a.size();
3 a.get(int posicio);
4 a.add(Objecte O);
5 a.remove(int posicio);  //esborra i mou tots cap a l'esquerra
```

3 Espera activa

Tots aquells mecanismes per aconseguir exclusió mútua sense bloqueig que ens poden ser útils són:

3.1 Solucions hardware

- **Test&Set**: és una instrucció de llenguatge màquina que permet consultar i modificar de forma atòmica el valor d'una variable. Implementa, atòmicament (és a dir, durant un sol cicle de rellotge), el codi:

```
1 public boolean testAndSet() {
2     boolean tmp = lock;
3     lock = true;
4     return tmp;
5 }
```

Aquest codi ens permet utilitzar estructures com:

```
1 public void entrar_zc() {
2     while(testAndSet());
3 }
4 public void sortir_zc() {
5     lock = false;
6 }
```

En Java, existeix una simulació software d'aquesta implementació: la classe *AtomicBoolean*.

3.2 Solucions software

- **Algoritme de tie Break de Peterson**: vàlid per a 2 processos (generalitzable per a un nombre prefixat de processos N). Els processos comparteixen variables de manera intel·ligent per a no bloquejar-se, tenir exclusió mútua i evitar esperes innecessàries.

```
1  boolean dinsP1 = false;
2  boolean dinsP2 = false;
3  int ultim = 1;
4
5  entrar_zcP1() {
6      dinsP1 = true;
7      ultim = 1;
8      while(dinsP2 && ultim == 1);
9  }
10 sortir_zcP1() {
11     dinsP1 = false;
12 }
13
14 entrar_zcP2() {
15     dinsP2 = true;
16     ultim = 2;
17     while(dinsP1 && ultim == 2);
18 }
19 sortir_zcP2() {
20     dinsP2 = false;
21 }
```

- **Algoritme de la fleca de Lamport**: vàlid per a un nombre prefixat de processos N. Els processos estan ordenats a priori. Quan un procés vol entrar a la zona crítica agafa el primer nombre que no ha agafat ningú. Si dos processos agafen el mateix nombre, te prioritat el primer procés (segons l'ordre entre els processos).

```
1  int nombre[] = new int[N];           //Iniciats a 0
2  boolean elegint[] = new boolean[N];  //Iniciats a false
3
4  entrar_zc(int i) {       //per al proces i-essim
5      elegint[i] = true;
6      nombre[i] = 1 + max(nombre[0], nombre[1], ..., nombre[N-1]);
7      elegint[i] = false;
8
9      for(int j=0; j<N; j++) {
10         //Evita que adelantis a un proces que pot agafar el mateix
11         //nombre que tu:
12         while(elegint[j]);
13         //Evita que comencis la zona critica abans del teu torn
14         while( (nombre[j] != 0) && (nombre[j]<nombre[i] || (nombre[j]==nombre[i]
               && j<i)) );
15     }
16 }
17 sortir_zc(int i) {       //Per al proces i-essim
18     nombre[j] = 0;
19 }
```

3.3 Await

Per facilitar la lectura, quan estem tractant amb problemes d'espera activa, a vegades utilitzarem la notació:

```
1  await(CONDICIO) {
2      // CODI
3  }
```

Que considerarem equivalent al següent codi Java:

```
1  entrar_zc();
2  while(!CONDICIO) {
```

```
3       sortir_zc();          //Sortim per evitar bloqueig
4       while(!CONDICIO);
5       entrar_zc();
6   }
7   // CODI
8   sortir_zc();
```

4 Semàfors

4.1 Definició

Definició 7. Un *semàfor* és un mecanisme per evitar la concurrència en memòria compartida. És un comptador protegit no negatiu que te dues operacions considerades atòmiques i fins a 4 paràmetres (que habitualment no són accessibles):

```
1   //Operacions:
2   semafor.V();      //Incrementa el comptador
3   semafor.P();      //Decrementa el comptador
4
5   //Parametres
6   semafor.s;        //Valor actual del semafor
7   semafor.i;        //Valor inicial del semafor
8   semafor.v;        //Nombre d'operacions v() que s'han fet
9   semafor.p;        //Nombre d'operacions p() que s'han fet
```

Definició 8. Diem que un semàfor és *feble* si quan el comptador està nul i s'incrementa no tenen prioritat els procesos que estan esperant prèviament.

Definició 9. Diem que un semàfor és *fort* si quan el comptador està nul i s'incrementa tenen prioritat els procesos que estan esperant prèviament.

4.2 Implementació

A partir d'una classe per aconseguir l'exclusió mútua, *Mutex*, podriem implementar un semàfor com:

```
1   public class Semafor {
2       private int s;
3       private Mutex m;
4
5       public Semafor (int s) {
6           this.s = s;
7           m = new Mutex();
8       }
9       public void V() {
10          m.entrar_zc();
11          s = s+1;
12          m.sortir_zc();
13      }
14      public void P() {
15          m.entrar_zc();
16          while(s == 0) {
17              m.sortir_zc();
18              while(s == 0);
19              m.entrar-sc();
20          }
21          s = s-1;
22          m.sortir_zc();
23      }
24  }
```

4.3 Exemples

1. **Barrera de N processos:** Un cas particular i prou rellevant de l'ús de semàfors és la barrera de N processos. Volem doncs un objecte que fagi esperar els procesos fins que no n'hagin arribat N, deixant-los pasar posteriorment. Una implementació de la barrera de N passos amb semàfors és:

```
1  public class Barrera {
2      int N;
3      int n = 0;       //n es el comptador de processos a la barrera
4      Semafor EM = new Semafor(1);
5      Semafor P1 = new Semafor(0);
6      Semafor P2 = new Semafor(1);
7      public Barrera (int N) {
8          this.N = N;
9      }
10     public void barrera_wait() {
11         EM.P();
12         n = n+1;
13         if(n == N) {
14           P1.V();
15           P2.P();
16         }
17         EM.V();
18         P1.P();
19         P1.V();
20         EM.P();
21         n = n-1;
22         if(n == 0) {
23           P2.V();
24           P1.P();
25         }
26         EM.V();
27         P2.P();
28         P2.V();
29     }
30 }
```

   Observació. Aquesta barrera només funciona si tenim exactament N processos (ja que falla el codi de les linies 18 i 19).

2. **Pas de testimoni:** Una altra estrategia típica a l'hora de treballar amb concurrència és el *pas de testimoni*. L'estratègia consisteix bàsicament en dirigir els diferents procesos en un ordre específic. Ho podem entendre com que els procesos es van pasant un únic testimoni que els permet continuar amb la seva tasca (quan un acaba, comença el següent). Implementat amb semàfors funcionaria de la següent manera: quan un procés vol entrar a la zona crítica agafa boleta d'un semàfor inicialitzat a 1. Quan el procés surt de la zona crítica, retorna la boleta. Això assegura que només un procés es trobarà a dins de la zona crítica.

5 Monitors

5.1 Definició

Definició 10. Anomenem *monitor* a una classe (i als seus mètodes associats) que ens permeten delimitar una zona crítica per tal d'executar-la amb exlusió mutua. Com a màxim un procés pot estar executant codi dins el monitor. Per tal d'assolir tan atrevit objectiu, el monitor ens proporciona mecanismes per tal d'aturar (adormir) i despertar processos. El monitor ens proporciona una implementació d'algún mecanisme per aconseguir exclusió mútua i cues de processos adormits.

Definició 11. A cada una de les cues de processos adormits li anomenem *variable de condició*, o simplement *condició*, i proporciona els mètodes següents:

- **cond_wait()**: Adorm el procés que l'executa i el posa a la cua de procesos. Deixa lliure el monitor.
- **cond_signal()**: Desperta només un procés (si hi ha algun adormit) i el treu de la cua.

- **cond_signalAll()**: Desperta tots els procesos de la cua. Només està implementat en els monitors *Signal & Continue*: en els altres no esta ben definit.

Observació. Aquestes funcions només estan pensades per a ser executades per processos que haguin entrat a dins del monitor (és a dir, que haguin accedit a la part protegida per l'exclusió mútua). En monitors natius en Java, aquesta condició es tradueix en que només podem utilitzar aquests mètodes en funcions definides com *synchronized*.

Ens trobem varies polítiques de senyalització (i.e., de funcionament intern del mètode *cond_signal()* de les condicions del monitor).

- **Signal & Continue**: Quan el procés A fa un *cond_signal()* i desperta a un altre procés B, el procés A continua executant el codi. Encara que els dos processos A i B estiguin desperts no executaràn la zona crítica simultàneament ja que la exclusió mútua ho impedirà.
- **Signal & Wait**: Quan el procés A fa un *cond_signal()* i desperta a un altre procés B, el desgraciat de A es posa a la cua de processos esperant. Poc habitual.
- **Signal & Urgent Wait**: Quan el procés A fa un *cond_signal()* i desperta a un altre procés B, el procés B pasa a executar el codi al monitor mentre el semidesgraciat de A pasa a esperar a una cua urgent. Per tant, A tendrà preferència sobre els processos que no han fet cap *cond_signal()*. Un monitor *Signal & Urgent Wait* també es coneix com un *Monitor Hoare*.

5.2 Implementació

1. **Monitor S&C a partir de semàfors**: on tant l'exclusió mútua com les condicions utilitzen semàfors.

```
1  // Inici fitxer Monitor_SC.java:
2  public class Monitor_SC {
3      private Semafor exclusioMutua;
4
5      public Monitor_SC() {
6          exclusioMutua = new Semafor(1);
7      }
8
9      public void entrarMonitor() {
10         exclusioMutua.P();
11     }
12
13     public void sortirMonitor() {
14         exclusioMutua.V();
15     }
16
17     public Condicio_SC crearCondicio() {
18         return new Condicio_SC(this);
19     }
20 }
21
22
23 // Inici fitxer Condicio_SC.java:
24 public class Condicio_SC {
25     private Semafor condicio;
26     private Monitor_SC monitor;
27     private int dormint;
28
29     public Condicio_SC(Monitor_SC monitor) {
30         condicio = new Semafor(0);
31         this.monitor = monitor;
32         dormint = 0;
33     }
34
35     public void cond_wait() { //wait
36         dormint++;
37         monitor.sortirMonitor();
38         condicio.P();
39         monitor.entrarMonitor();
40     }
41
42     public void cond_signal() { // notify
```

```
43          if (dormint > 0) {
44              dormint--;          //Politica S&C: desperta un thread pero
45              condicio.V();       // no adorm al que executa aquest codi
46          }
47      }
48
49      public void cond_signalAll() { // notifyAll
50          for (; dormint > 0; dormint--)
51              condicio.V();
52      }
53  }
```

2. **Monitor S&UW a partir d'un monitor S&C**:

```
1   // Inici fitxer Monitor_SUW.java:
2   public class Monitor_SUW {
3       Monitor_SC monitor_sc;
4       Condicio_SC condNormal, condUrgent;
5       int processosUrgents;
6
7
8       public Monitor_SUW() {
9           monitor_sc = new Monitor_SC();
10          condNormal = monitor_sc.crearCondicio();
11          condUrgent = monitor_sc.crearCondicio();
12          processosUrgents = 0;
13      }
14
15      public void entrarMonitor() {
16          monitor_sc.entrarMonitor();
17          while (processosUrgents != 0)
18              condNormal.cond_wait();
19      }
20
21      public void sortirMonitor() {
22          if(processosUrgents != 0)
23              condUrgent.cond_signal();
24          else
25              condNormal.cond_signal();
26
27          monitor_sc.sortirMonitor();
28      }
29
30      public Condicio_SUW crearCondicio() {
31          return new Condicio_SUW(this);
32      }
33  }
34
35
36  // Inici fitxer Condicio_SUW.java:
37  public class Condicio_SUW {
38      private Condicio_SC condicio;
39      private Monitor_SUW monitor_suw;
40      private int processosAdormits;
41
42      public Condicio_SUW(Monitor_SUW monitor_suw) {
43          this.monitor_suw = monitor_suw
44          condicio = monitor_suw.monitor_sc.crearCondicio();
45          processosAdormits = 0;
46      }
47
48      public void cond_wait() {       //wait
49          processosAdormits++;
50          condicio.cond_wait();
51          while (monitor_suw.processosUrgents != 0)
52              monitor_suw.condNormal.cond_wait();
53          processosAdormits--;
54      }
55
56      public void cond_signal() {     //notify
```

```
57          if (processosAdormits != 0) {
58              condicio.cond_signal();
59
60              monitor_suw.processosUrgents++;
61              monitor_suw.condUrgent.cond_wait();
62              monitor_suw.processosUrgents--;
63          }
64      }
65  }
```

5.3 Monitor natiu de java

Java té implementat a la classea *Object* un monitor amb política *Signal&Continue* i una sola variable de condició. Donat que qualsevol objecte hereta de *Object*, tots els objectes ho tenen implementat implícitament. Per a utilitzar-lo, necessitem dues coses:

- **Assegurar l'exclusió mútua**: aquest codi Java:

```
1  public synchronized void metode() {
2      // CODI
3  }
```

és equivalent a:

```
1  public void metode() {
2      entrar_monitor();
3      // CODI
4      sortir_monitor();
5  }
```

- **Utilitzar la condició**: cada objecte te definits els següents mètodes (que només es poden utilitzar en mètodes *synchronized*)

```
1  public final void wait();          //equivalent a cond_wait()
2  public final void notify();        //equivalent a cond_signal()
3  public final void notifyAll();     //equivalent a cond_signalAll()
```

5.4 Exemples

1. **Intercanviador de nombres**: es tracta de implementar una classe amb un unic mètode, *intercanvi(int valor)*, que quan hi accedeixen dos threads diferents intercanvia el seu valor. Està implementat amb un monitor S&C.

```
1  public class Intercanviador extends Monitor_SC {
2      private Condicio cond;
3      private boolean alguEsperant;
4      private boolean processantIntercanvi;
5      private int valorProvisional;
6
7      public void Intercanviador() {
8          cond = this.crearCondicio();
9          alguEsperant = false;
10         processantIntercanvi = false;
11         valorProvisional = 0;
12     }
13
14     public int intercanvi(int valor) {
15         int aux;
16         this.entrarMonitor();
17         while(processantIntercanvi) //Evita que entrin mes de 2 processos
18             cond.cond_wait();
19         if(alguEsperant) {          //Ets el primer
```

```
20              alguEsperant = true;
21              cond.cond_wait();
22              aux = valorProvisional;
23              processantIntercanvi = false;
24              alguEsperant = false;
25          } else {                    //Ets el segon
26              processantIntercanvi = true;
27              aux = valorProvisional;
28              valorProvisional = valor;
29          }
30          cond.cond_signalAll();
31          this.sortirMonitor();
32          return aux;
33      }
34  }
```

6 Pas de Missatges

6.1 Definicio

Un cop assolida l'exclusió mutua, volem treballar en l'accés a un servidor des de múltiples clients. Això ens comportara un problema de concurrència equivalent al que hem resolt anteriorment, però treballant des de màquines diferents.

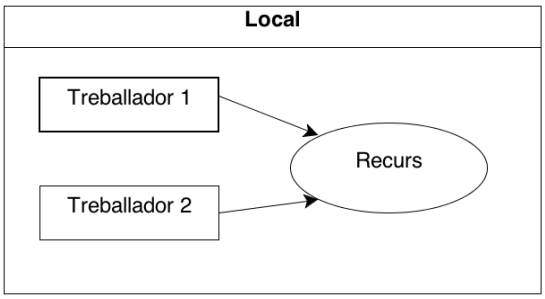

Figura 1: Programació concurrent en un entorn local

La idea és aprofitar el codi que utilitzem en el problema local, però afegint classes de connexió (*Socket*) per a fer transparent la comunicació entre el servidor i el client, i que les classes client i servidor segueixin creient que treballen en un entorn local (figura 1). Tendrem que programar el codi del Servidor i el codi del Client. Treballarem amb dues arquitectures de servidor diferents:

- **Arquitectura multifil**: Utilitzem la solució en local trobada en els apartats anteriors i creem un *Socket* tant al client com al servidor. D'aquesta manera, el servidor treballarà igual que en local i el client no haurà de fer cap canvi a l'hora de treballar amb recursos externs. El treball de comunicació el faran les dues classes *Socket* (figura 2).
- **Monitor Actiu**: Totes les connexions es guarden a un buffer. Tindrem un únic fil de treball on s'aniran agafant les peticions i procesant de manera sequencial. Per tant, el servidor no troba concurrència per a accedir al recurs, sinó que te una cua de *Sockets* que ha de gestionar (figura 3).

Treballarem, en els dos casos, amb connexions que s'obren per a fer una acció, fan l'acció, i es tanquen. És a dir, no reutilitzem la mateixa connexió per a diferentes accions d'un mateix client.

6.2 Sockets en Java

IMPORTANT. En aquesta secció es descriu el comportament dels Sockets en el llenguatge Java per a poder entendre com es comporten. No obstant, podem encapsular tota aquesta construcció en les classes *MySocket* i *MyServerSocket*, així que no és necessari conèixer els detalls ni la sintaxi concreta de tots els objectes. Al següent apartat s'expliquen MySocket i MyServerSocket.

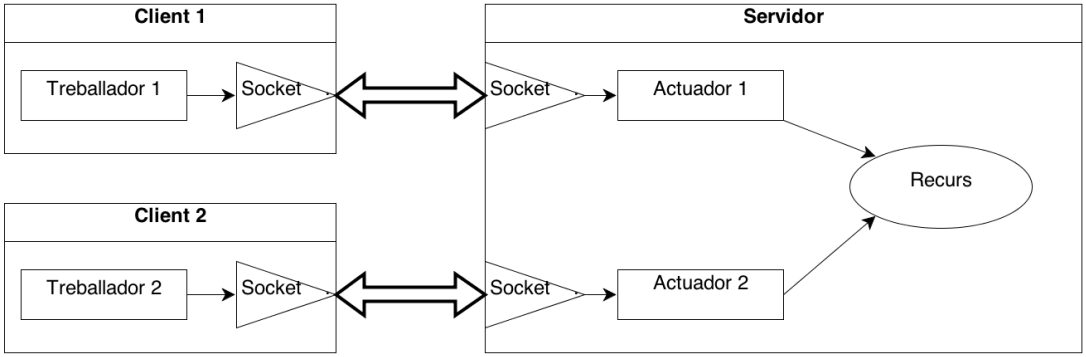

Figura 2: Diagrama funcional d'un sistema d'arquitectura multifil

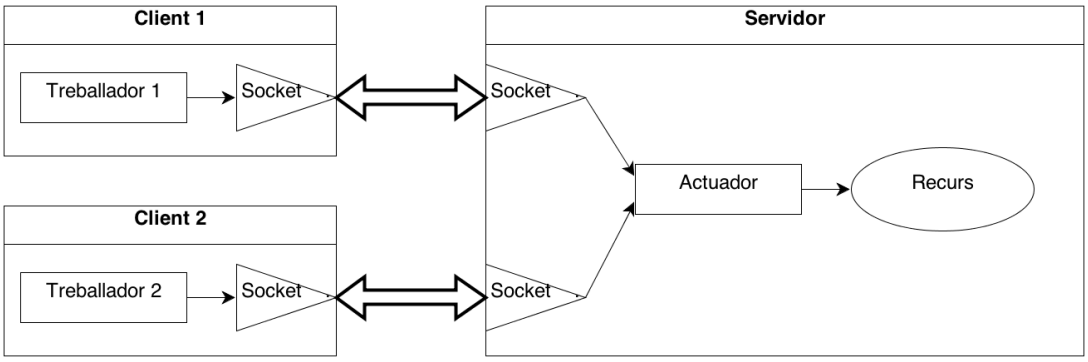

Figura 3: Diagrama funcional d'un sistema de monitor actiu

La classe *Socket* serà una classe que tendràn tant el client com el servidor per enviar i rebre informació. S'hauran de connectar mitjançant el mateix port.

- El servidor tendrà una classe *ServerSocket* que acceptarà connexionts entrants, creant una classe *Socket* per a cada connexió.
- El client crearà una classe *Socket* directament, que es connectarà al servidor quan sigui creada.

Per una altra part, una vegada creada la connexió, per a communicar-nos utilitzarem Streams (classes que treballen amb fluxos de dades). Existeixen molts tipus d'Streams, però usualment utilitzarem les classes *ObjectInputStream* i *ObjectOutputStream*, que permeten enviar i rebre dades dels tipus primitius i objectes en general.

Els mètodes que utilitzarem seran:

- Definir el port i el servidor:

```
1 public class Comms {
2     public static final int port = 8888;            //El podem escollir
3     public static final String s = "localhost";
4 }
```

- Crear la connexió:

```
1 // Servidor:
2 ServerSocket ss = new ServerSocket(Comms.port);
3 Socket s;
4 while(true) {
5     s = ss.accept();
6     //processar la connexio s en un altre Thread i seguir rebent connexions
7 }
```

- Crear els Streams:

```
// Al servidor:
ObjectInputStream entrada = new ObjectInputStream(socket.getInputStream());
ObjectOutputStream sortida = new ObjectOutputStream(socket.getOutputStream());
// Al client:
ObjectOutputStream sortida = new ObjectOutputStream(socket.getOutputStream());
ObjectInputStream entrada = new ObjectInputStream(socket.getInputStream());
```

Observació. És important observar que els streams s'inicialitzen creuats! Això és perque a l'inicialitzar l'entrada per al servidor estarem inicialitzant la sortida per al client i viceversa.

- Llegir i escriure dades:

```
sortida.writeChar('a');                          //Escriure char
sortida.writeInt(1712);                          //Escriure int
sortida.writeDouble(1712.860656);                //Escriure double
sortida.writeObject("Novembre de 1712");         //Escriure un String
sortida.writeObject(Objecte obj);                 //Escriure un objecte qualsevol
sortida.flush();                 //passar al socket tot allo que hem escrit

c = entrada.readChar();              //Llegir char
i = entrada.readInt();               //Llegir int
d = entrada.readDobule();            //Llegir double
s = (String)entrada.readObject();    //Llegir String
o = entrada.readObject();            //Llegir un objecte qualsevol
```

Observació. Observem que tots els mètodes de lectura són bloquejants! (és a dir, fins que no arribi la dada que volem llegir, el Thread que l'executa no continuarà).

Observació. Després de qualsevol escriptura hem de fer *sortida.flush()* per a buidar l'Stream.

Observació. Les classes *DataInputStream* i *DataOutputStream* són similars a les que utilitzem, però no permeten escriure o llegir objectes de la classe *Object* (i, per tant, tampoc Strings... encara que no entrem en detalls, per a enviar Strings la Corona Britànica ens permet fer algun arreglillo).

6.3 MySocket i MyServerSocket

MySocket i MyServerSocket són dues classes que no són natives del llenguatge Java, pero que podem utilitzar per a fer més llegible el codi dels exemples/exercicis.

```
// Exemple d'us de la classe MySocket:
    MySocket s = new MySocket("localhost", 8888);
    s.write_int(1712);
    char c = s.read_char();
    s.write_string("Novembre de 1712");
    s.close();

// Exemple d'us de la classe MyServerSocket:
    MyServerSocket ss = new MyServerSocket(8888);
    while(true) {
        MySocket s = ss.accept();
        int i = s.read_int();
        if(i == 0)
            s.write_char('K');
        else
            s.write_char('a');
        String s = s.read_string();
        s.close();
    }
```

6.4 Exemples

1. **Arquitectura multifil**: suposem que tenim resolt per monitors un problema on molts threads iguals han d'obtenir permis per a utilitzar un recurs extern. És a dir, ens donen el següent codi:

```
1  //Classe abstracta per accedir a un recurs
2  public abstract class AccesRecurs {
3      public abstract void iniciAcces();
4      public abstract void fiAcces();
5
6      public abstract boolean estaLliure();
7  }
8
9  //Implementacio amb monitors de AccesRecurs
10 public class AccesRecurs_Monitors extends AccesRecurs { /*...*/ }
11
12 //Implementacio d'un thread treballador
13 public class Treballador extends Thread {
14     public void Treballador(AccesRecurs a);
15     public void run() { /*...*/ }
16     /* ... */
17 }
```

Si volem utilitzar l'arquitectura multifil, haurem de programar els sockets i connectar-los convenientment:

```
1  /********* CLIENT *********/
2  public class TreballadorRemot_M {
3
4      public static void main(String[] args) throws Exception {
5          AccesRecurs a = new AccesRecursSimulat_M();
6          Treballador t = new Treballador(a);
7          t.start();
8      }
9  }
10
11 class AccesRecursSimulat_M extends AccesRecurs {
12     Socket s;
13     ObjectOutputStream out;
14     ObjectInputStream in;
15
16     public void iniciAcces() {
17         s = new Socket(Comms.host, Comms.port);
18         out = new ObjectOutputStream(s.getOutputStream());
19         in = new ObjectInputStream(s.getInputStream());
20
21         out.writeInt(Comms.peticioIniciAcces);
22         out.flush();
23         in.readInt();   //No cal guardar el valor, no ens interesa
24
25         out.close();
26         in.close();
27         s.close();
28     }
29
30     public void fiAcces() {
31         s = new Socket(Comms.host, Comms.port);
32         out = new ObjectOutputStream(s.getOutputStream());
33         in = new ObjectInputStream(s.getInputStream());
34
35         out.writeInt(Comms.peticioFiAcces);
36         out.flush();
37
38         out.close();
39         in.close();
40         s.close();
41     }
42
43     public boolean estaLliure() {   //Necessari sobreescriure perque es abstract
44         throw new UnsupportedOperationException("No cal implementacio.");
45     }
```

```
46 }
47
48 /********* SERVIDOR *********/
49 public class AccesRecursServidor_M {
50
51     public static void main (String[] args) {
52         AccesRecurs a = new AccesRecurs_Monitors();
53         ServerSocket ss = new ServerSocket();
54         Socket s;
55         Actuador_M act;
56         while(true) {
57             s = ss.accept();            //acceptar connexions
58             act = new Actuador_M(a, s);
59             act.start();
60         }
61     }
62 }
63
64 class Actuador_M extends Thread {
65     AccesRecurs a;
66     ObjectOutputStream out;
67     ObjectInputStream in;
68
69     public Actuador_M(AccesRecurs a, Socket s) {
70         this.a = a;
71         in = new ObjectInputStream(s.getInputStream());
72         out = new ObjectOutputStream(s.getOutputStream());
73     }
74
75     public void run() {
76         int accio;
77         while(true) {
78             accio = in.readInt();
79             switch(accio) {
80                 case Comms.peticioIniciAcces:
81                     a.iniciAcces();
82                     out.writeInt(Comms.ACK);
83                     out.flush();
84                     break;
85                 case Comms.peticioFiAcces:
86                     a.fiAcces();
87                     break;
88             }
89         }
90     }
91 }
92
93
94 /*********  COMUNS  *********/
95 public class Comms {
96     public static final int port = 8888;
97     public static final String host = "localhost";
98     public static final int peticioIniciAcces = 0;
99     public static final int peticioFiAcces = 1;
100    public static final int ACK = 2;
101 }
```

2. **Monitor actiu**: seguint amb el mateix exemple anterior, i donat el mateix codi, volem utilitzar el sistema de monitor actiu per a implementar el sistema distribuit. Notem que els codis són molt semblants al cas d'arquitectura multifil.

   Observació. En aquest cas, no cal que AccesRecurs estigui implementat per un monitor, ja que mai es trobarà amb una situació de concurrència: només treballem amb un fil. Els problemes de concurrència els tracta la classe ServerSocket.

   Observació. No obstant, encara que no hi hagui concurrència, haurem de resoldre un problema que sorgeix: *no podem tancar una connexió entrant fins que no hem processat la seva petició.* Si en un moment donat no la podem

processar (per exemple, perque el recurs compartit s'està utilitzant), la guardarem a una cua de connexions, i la finalitzarem quan sigui possible.

```
1  /********* CLIENT *********/
2  public class TreballadorRemot_MA {
3
4      public static void main(String[] args) {
5          AccesRecurs a = new AccesRecursSimulat_MA();
6          Treballador t = new Treballador(a);
7      }
8  }
9
10 class AccesRecursSimulat_MA extends AccesRecurs {
11     Socket s;
12     ObjectOutputStream out;
13     ObjectInputStream in;
14
15     public void iniciAcces() {
16         int rebut = Comms.NACK;
17         while(rebut != Comms.ACK) {
18             s = new Socket(Comms.host, Comms.port);
19             out = new ObjectOutputStream(s.getOutputStream());
20             in = new ObjectInputStream(s.getInputStream());
21
22             out.writeInt(Comms.peticioIniciAcces);
23             out.flush();
24             rebut = in.readInt();    //En arquitectura multifil no caldria
25                                      //  ni guardar el valor
26             out.close();
27             in.close();
28             s.close();
29         }
30     }
31
32     public void fiAcces() {
33         s = new Socket(Comms.host, Comms.port);
34         out = new ObjectOutputStream(s.getOutputStream());
35         in = new ObjectInputStream(s.getInputStream());
36
37         out.writeInt(Comms.peticioFiAcces);
38         out.flush();
39
40         out.close();
41         in.close();
42         s.close();
43     }
44 }
45
46
47 /********* SERVIDOR *********/
48 public class AccesRecursServidor_MA {
49
50     public static void main (String[] args) {
51         AccesRecurs a = new AccesRecurs_Monitors();
52         ArrayList<Actuador_MA> cua = new ArrayList<Atuador_MA>();
53         ServerSocket ss = new ServerSocket();
54         Socket s;
55         Actuador_MA act;
56         while(true) {
57             s = ss.accept();            //acceptar connexions
58             act = new Actuador_MA(a, s, cua);
59             act.processa();
60         }
61     }
62 }
63
64 class Actuador_MA extends Thread {
65     AccesRecurs a;
66     Socket s;
67     ArrayList<Actuador_MA> cua;
68     ObjectOutputStream out;
```

```
69      ObjectInputStream in;
70
71      public Actuador_MA(AccesRecurs a, Socket s, ArrayList<Actuador_MA> cua) {
72          this.a = a;
73          this.s = s;
74          this.cua = cua;
75          in = new ObjectInputStream(s.getInputStream());
76          out = new ObjectOutputStream(s.getOutputStream());
77      }
78
79      public void processa() {
80          int accio;
81          accio = in.readInt();
82          switch(accio) {
83              case Comms.peticioIniciAcces:
84                  if(a.estaLliure()) {        //No cal exclusio mutua
85                      a.iniciAcces();
86                      out.writeInt(Comms.ACK);
87                      tancaConnexio();
88                  } else {
89                      cua.add(this);        //Insertar a la cua
90                  }
91                  break;
92              case Comms.peticioFiAcces:
93                  a.fiAcces();
94                  //No cal enviar res a qui tanca la connexio
95                  tancaConnexio();
96                  if(cua.size() != 0) {
97                      Actuador_MA actuador = cua.remove(0);  //Extreu element
98                      a.iniciAcces();
99                      actuador.out.writeInt(Comms.ACK);
100                     actuador.out.flush();
101                     actuador.tancaConnexio();
102                 }
103                 break;
104         }
105     }
106
107     public void tancaConnexio() {
108         out.flush();
109         out.close();
110         in.close();
111         s.close();
112     }
113 }
114
115
116 /********* COMUNS *********/
117 public class Comms {
118     public static final int port = 8888;
119     public static final String host = "localhost";
120     public static final int peticioIniciAcces = 0;
121     public static final int peticioFiAcces = 1;
122     public static final int ACK = 2;
123 }
```

Si volem utilitzar les classe *MySocket* i *MyServerSocket*, la cua la fariem directament d'objectes MySocket, en lloc de fer-la de objectes Actuador$_MA$.