

Asignación de Prácticas Número 6

Programación Concurrente y de Tiempo Real

Antonio J. Tomeu¹

¹Departamento de Ingeniería Informática
Universidad de Cádiz

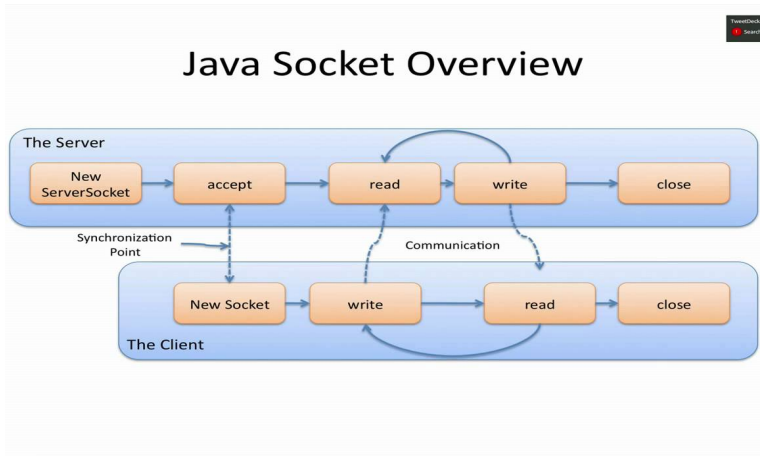
PCTR, 2021

Objetivos de la Práctica

- ▶ Aprender a comunicar códigos distribuidos mediante una aproximación básica a los *sockets*.
- ▶ Introducir la tecnología básica que subyace en cualquier servidor.
- ▶ Utilizar las herramientas que el API estándar de Java proporciona para controlar la concurrencia, mediante bloques de código y métodos *synchronized*.
- ▶ Desarrollar clases seguras frente a concurrencia.

- ▶ Es posible distribuir aplicaciones comunicando a códigos que se ejecutan en máquinas distintas, a través de una red de comunicaciones.
- ▶ La tecnología de comunicación basada en *sockets* proporciona probablemente la aproximación de más bajo nivel para resolver la comunicación
- ▶ Un *socket* es una abstracción que permite a dos procesos (eventualmente en ejecución en máquinas diferentes) intercambiar un flujo de datos a través del mismo.
- ▶ La práctica totalidad de los lenguajes de programación ofrecen soporte a comunicaciones no orientadas a conexión mediante *sockets* de datagrama, y orientados a conexión mediante *sockets* de *stream*.

Modelando un Servidor Elemental Multihebrado de Sockets: Estructura I



La Clase ServerSocket I

- ▶ Sirve para poner a un servidor a «escuchar» a un puerto de red.
- ▶ El constructor suele estar parametrizado con el puerto que queremos escuchar.
- ▶ El servidor queda bloqueado escuchando el puerto a la espera de peticiones de servicio mediante el método `accept`.
- ▶ Para multihebrar el servidor, inmediatamente que se recibe un petición de servicio activamos una hebra (que se comunicará con el cliente) y el ciclo principal del servidor se pone de nuevo a la escucha.
- ▶ Esto permite tiempo de respuesta razonable cuando el número de peticiones es alto.
- ▶ Navegamos al API de la clase `ServerSocket`.

Modelando un Servidor Elemental Multihebrado de Sockets: El Código I

```
1  import java.net.*;
2  import java.io.*;
3
4  public class Servidor_Hilos
5      extends Thread
6  {
7      Socket enchufe;
8      public Servidor_Hilos(Socket s)
9      {enchufe = s; this.start();}
10
11     public void run()
12     {
13         try{
14             BufferedReader entrada = new BufferedReader(
15                                     new InputStreamReader(
16                                         enchufe.getInputStream()));
17             String datos = entrada.readLine();
18             int j;
19             int i = Integer.valueOf(datos).intValue();
20             for(j=1; j<=20; j++){
```

Modelando un Servidor Elemental Multihebrado de Sockets: El Código II

```
21         System.out.println("El hilo "+this.getName()+"  
           escribiendo el dato "+i);  
22         sleep(1000);}  
23         enchufe.close();  
24         System.out.println("El hilo "+this.getName()+"cierra su  
           conexion...");  
25     } catch(Exception e) {System.out.println("Error...");}  
26 }  
27  
28 public static void main (String[] args)  
29 {  
30     int i;  
31     int puerto = 2001;  
32     try{  
33         ServerSocket chuff = new ServerSocket (puerto,  
           3000);  
34  
35         while (true){  
36             System.out.println("Esperando solicitud de  
               conexion...");
```

Modelando un Servidor Elemental Multihebrado de Sockets: El Código III

```
37         Socket cable = chuff.accept();
38         System.out.println("Recibida solicitud de
           conexion...");
39         new Servidor_Hilos(cable);
40     } //while
41 } catch (Exception e)
42 { System.out.println("Error en sockets...");}
43 } //main
44
45 } //Servidor_Hilos
```


La Clase Socket I

- ▶ Soporta *sockets* de stream (TCP) en lenguaje Java
- ▶ El cliente pide al servidor (que está escuchando) abrir un nuevo *socket* para comunicarse con el.
- ▶ Para ello, el cliente necesita conocer:
 - ▶ dónde se ubica el servidor; esto habitualmente es una dirección IP.
 - ▶ qué puerto está escuchando el servidor
- ▶ Una vez abierto el canal (*socket*) servidor y cliente se comunican con los métodos de la clase para enviar y recibir flujos de datos. Estos métodos normalmente se «envuelven» en clase decoradoras para facilitar al programador el uso del *Socket*.
- ▶ El cliente recibe el servicio mediante una hebra dedicada desde el lado del servidor... aunque esto es transparente para él.
- ▶ Este clase no proporciona canales seguros, pero la clase *SSLSocket* sí.

El Cliente: Código I

```
1  import java.net.*;
2  import java.io.*;
3
4  public class Cliente_Hilos
5  {
6      public static void main (String[] args)
7      {
8          int i = (int)(Math.random()*10);
9          int puerto = 2001;
10         try{
11             System.out.println("Realizando conexion...");
12             Socket cable = new Socket("localhost", 2001);
13             System.out.println("Realizada conexion a "+cable);
14             PrintWriter salida= new PrintWriter(
15                                     new BufferedWriter(
16                                         new OutputStreamWriter(
17                                             cable.getOutputStream())));
18             salida.println(i);
19             salida.flush();
20             System.out.println("Cerrando conexion...");
21             cable.close();
22         }
```

```
23         }//try
24         catch (Exception e)
25         {System.out.println("Error en sockets...");}
26     }//main
27 }//Cliente_Hilos
```

- ▶ A partir de la solución que acabamos de analizar, realice las siguientes modificaciones:
 - ▶ soporte las tareas del servidor mediante objetos que implementen a Runnable.
 - ▶ procese la tareas mediante un ejecutor.
 - ▶ juegue con el código a fondo; no se limite a los cuatro cambios básicos que pedimos.

Control de Sección Crítica con synchronized I

```
1 public class secureCriticalSection
2     extends Thread{
3     public static long iterations = 1000000;
4     public static long n          = 0;
5     public static Object lock     = new Object();
6
7     public secureCriticalSection(){
8     public void run(){
9         for(long i=0; i<iterations; i++) synchronized(lock){n++;}
10    }
11
12    public static void main(String[] args) throws Exception{
13        secureCriticalSection A = new secureCriticalSection();
14        secureCriticalSection B = new secureCriticalSection();
15        A.start(); B.start();
16        A.join(); B.join();
17        System.out.println(n);
18    }
19 }
```

Control de Objetos con synchronized I

Sincronizamos el objeto...

```
1 public class secureObject{
2     public long n = 0;
3
4     public secureObject(){
5     public synchronized void inc(){n++;}
6     public synchronized long get(){return this.n;}
7 }
```

...y lo usamos de forma segura:

```
1 public class usingSecureObject
2     extends Thread{
3     public static long iterations = 1000000;
4     public secureObject obj;
5
6     public usingSecureObject(secureObject obj){this.obj = obj;}
7     public void run(){
8         for(long i=0; i<iterations; i++)obj.inc();
9     }
10
11     public static void main(String[] args) throws Exception{
```

Control de Objetos con synchronized II

```
12     secureObject o      = new secureObject();
13     usingSecureObject A = new usingSecureObject(o);
14     usingSecureObject B = new usingSecureObject(o);
15     A.start(); B.start();
16     A.join(); B.join();
17     System.out.println(o.get());
18 }
19 }
```

Resolviendo el Ejercicio 2: Exclusión Mutua sobre un Array I

- ▶ En Java, los arrays son objetos.
- ▶ No son seguros frente a concurrencia.
- ▶ Esto es, no debemos tener a múltiples hebras escribiendo concurrentemente en un array... salvo que lo hagan en regions diferentes del mismo..
- ▶ ... o tomemos precauciones con control de exclusión mutua...
- ▶ ... o utilicemos un contenedor sincronizado.
- ▶ Ahora construya múltiples hebras que compartan el acceso a un array, y controle el acceso a dicho objeto de forma que sea seguro frente a concurrencia.

Clases Seguras en Java: Objetos Inmutables I

- ▶ Un objeto es inmutable cuando no cambia de estado una vez ha sido creado.
- ▶ Técnicamente, carece de métodos modificadores.
- ▶ Son objetos seguros frente a concurrencia por diseño...
- ▶ ... pero a costa de restringir la fidelidad a la realidad modelada con el objeto, que suele ser dinámica y cambiante.
- ▶ Java establece una estrategia para diseñar objetos inmutables, que tiene las siguientes fases:
 - ▶ son objetos que carecen de modificadores («setters»).
 - ▶ todos los atributos son `private` y `final`.
 - ▶ la clase suele ser declarada como `final`.
 - ▶ si hay atributos que referencian a objetos mutables, no permitir que tales objetos sean modificados.
- ▶ La anterior es una estrategia muy radical...

- ▶ ... y puede tener una alternativa razonable en objetos donde todos los métodos están sincronizados, excepto los constructores, conservando la mutabilidad de los objetos, pero de forma controlada.

Resolviendo el Ejercicio 3: Objeto Heterogéno I

- ▶ En Java, es posible tener objetos con regiones de código sincronizadas y no sincronizadas.
- ▶ Esto dota al programador de una enorme flexibilidad para decidir qué regiones de sus objetos pueden ser ejecutadas por una única hebra, o por muchas...
- ▶ ... pero introduce riesgos si el control es incorrecto.
- ▶ En este ejercicio, se le pide que diseñe un objeto con regiones de código sincronizadas y no sincronizadas, a fin de que aprenda que este efecto puede ser ventajoso... o letal.

Resolviendo el Ejercicio 4: Provocando un *deadlock* I

- ▶ En el Tema 1 se le proporcionó un ejemplo de código con la estructura adecuada para (eventualmente) entrar en interbloqueo. Rescátelo
- ▶ Extienda ese código para que trabaje con tres hebras.
- ▶ Verifique que los interbloqueos se pueden dar.

Resolviendo el Ejercicio 5: Integración Numérica con Tareas Callable I

- ▶ Se trata de obtener la integral definida aproximada por Monte-Carlo de $f(x) = \cos(x)$ en $[0, 1]$
- ▶ Cada tarea Callable recibe un número de puntos y devuelve los que están bajo la curva. Los resultados parciales se recolectan utilizando Future.
- ▶ El programa principal efectúa los cálculos finales y muestra el valor final resultante.

Resolviendo el Ejercicio 6: A Vuelta con las Condiciones de Concurso I

- ▶ En la asignación número 2 escribió varias condiciones de concurso sin control y comprobó la presencia de entrelazados indeseables que estas producen.
- ▶ En este ejercicio se le pide que rescate esos códigos, y utilizando cerrojos y `synchronized`, proceda a controlar las condiciones de concurso que escribió, obteniendo códigos que accedan a secciones críticas de forma controlada.