MANUAL DE JAVA

Autor:

Lic. Jorge Herrero

Introducción a Threads

¿Qué es un Thread? (Hilo de Ejecución).

Un concepto fundamental en programación es manejar más de una tarea a la vez. Muchos problemas de programación requieren que el programa pueda detener lo que esté haciendo, tratar con algún otro problema y regresar al proceso principal.

Dentro de un programa, las partes que corren separadamente se llaman hilos (Thread) y el concepto general se llama Multithreading.

Ordinariamente, los hilos son una manera de asignar el tiempo de un solo procesador. Pero si el sistema operativo apoya procesadores múltiples, cada hilo puede asignarse a un procesador diferente y ellos pueden correr realmente en paralelo. Una de las características convenientes de Multithreading es que el programador no necesita preocupar sobre si existe uno o más procesadores. El programa es lógicamente dividido en hilos y si la maquina tiene más de un procesador entonces el programa corre más rápidamente, sin necesidad de ajuste especial.

¿Qué es un proceso?

Un proceso es un programa ejecutándose en forma independiente y con un espacio propio de memoria. Un Sistema Operativo multitarea es capaz de ejecutar más de un proceso simultáneamente. Un Thread o hilo es un flujo secuencial simple dentro de un proceso. Un único proceso puede tener varios hilos ejecutándose. Por ejemplo, los programas Firefox, Internet Explorer, Google Chrome serían un proceso, mientras que cada una de las ventanas y/o pestañas que se pueden tener abiertas simultáneamente trayendo páginas HTML estaría formada por al menos un (1) hilo.

Un sistema multitarea da realmente la impresión de estar haciendo varias cosas a la vez y eso es una gran ventaja para el usuario. Sin el uso de hilos hay tareas que son prácticamente imposibles de ejecutar, particularmente las que tienen tiempos de espera importantes entre etapas.

Los Threads o hilos de ejecución permiten organizar los recursos de la PC de forma que pueda haber varios programas actuando en paralelo. Un hilo de ejecución puede realizar cualquier tarea que pueda realizar un programa normal y corriente. Bastara con indicar lo que tiene que hacer en el método run(), que es el que define la actividad principal de las tareas.

¿Qué significa Multi-Threading?

La Máquina Virtual Java (JVM) es un sistema multi-thread. Es decir, es capaz de ejecutar varias secuencias de ejecución (programas) simultáneamente.

La programación multithreading (multi-hilo) permite la ocurrencia simultánea de varios flujos de control. Cada uno de ellos puede programarse independientemente y realizar un trabajo, distinto, idéntico o complementario, a otros flujos paralelos.

Un procesador ejecuta las instrucciones secuencialmente una después de la otra. A esta ejecución secuencial de instrucciones se le llama un Execution Thread (Hilo de Ejecución).

Multithreading es cuando se están ejecutando varios hilos simultáneamente.

Si un procesador ejecuta las instrucciones secuencialmente, una después de la otra ¿Cómo hace para ejecutar varios hilos simultáneamente? No puede hacer, pero puede simular que lo hace.

CREACIÓN DE THREADS

Alternativas de creación

Existen dos modos de crear Hilos en Java:

- Extender la clase Thread
- Implementar la interface Runnable

Creación a través de la clase Thread

- Crear una nueva clase que hereda la clase Thread y redefinir el método run(). (Este método contendrá el código que va a ser ejecutado por el hilo)
- Luego se instancian objetos de la clase derivada y se llama a su método start(). (Así se arrancan hilos que ejecutan el código contenido en el método run())

El paquete **java.lang** define una clase Thread que sirve para crear hilos.

Veamos un ejemplo:

```
/*
Esta clase define el prototipo para presentar mensajes en pantalla cada x segundos
*/
import java.awt.*;
import java.lang.*;
//definimos la clase Hilo derivada de la clase Thread
//así podremos instanciar todos los hilos que queramos
class Hilo extends Thread{
   private String s; //para almacenar los mensajes
```

```
    private int pausa; //pausa antes de presentar el mensaje
    //Constructor, se indica el mensaje y la pausa
    public Hilo(String m, int p){
        s = m;
        pausa = p;
    }
    //Definimos el método run() similar al main()pero para Hilos
    //recordar el contenido de Mensaje.java
    public void run(){
        do{
            try{
                sleep (pausa*1000);
            }catch (InterruptedException e){
                ;
            }
            System.out.println(s);
        }while (true);
    }
}
```

```
public class Mensaje{

   public static void main(String args[]){

      Hilo m1,m2; //instanciamos dos objetos de la clase Hilo

      String s1,s2;

      s1 = new String("Prueba del 1er Hilo");

      s2 = new String("Prueba del 2do Hilo");

      //creamos los hilos m1 y m2

      m1 = new Hilo(s1,3);

      m2 = new Hilo(s2,5);

      //arrancamos los hilos

      m1.start();

      m2.start();

   }

}
```

Creación a través de la interfaz Runnable

El otro modo de crear hilos es mediante la implementación de la interfaz Runnable. La razón para utilizar otro método para la creación de hilos es que nos puede interesar heredar de la clase Applet y a la vez emplear los servicios de threads.

En Java no está permitida la herencia múltiple, al menos de clases, la forma de solucionar este problema es utilizando una interface (lo que sería una clase abstracta pura en

C++) que permita que las clases que implementa la interface puedan ser utilizada por la clase Thread como base para crear un hilo, la interfaz utilizada se denomina Runnable.

Ya hemos visto la solución concurrente que dimos al problema inicial de presentar en pantalla dos mensajes sin utilizar applets, utilizamos la clase Hilo derivada de Thread y mensaje.java como programa principal que hacía uso de dicha clase.

Para poder hacer uso de la concurrencia en un applet es necesario incluir la interfaz Runnable, tal como hicimos en la solución concurrente con appletes, donde utilizamos la clase HiloApplet y el applet AppletConcurrent.

Podemos utilizar dos formas de crear Hilos utilizando Applets:

Una consiste en crear clases extendidas de Thread, que contendrá el código de ejecución del hilo, donde pasamos al constructor una referencia al applet que crea los hilos mediante this, esto es lo que hemos hecho en AppletConcurrent.java

```
hilo1 = new HiloApplet(this,pausa1);

hilo2 = new HiloApplet(this,pausa2);
```

Otra forma consiste en colocar el código de ejecución del hilo en el método run() del applet y crear los hilos como una instancia de la clase Thread. Esto es lo que hemos hecho en el applet appletConcurrent2.java

Ejemplo:

```
public class CreateThreadRunnableExample implements Runnable{

    public void run(){

        for(int i=0; i < 5; i++){

            System.out.println("Hijo Hilo : " + i);
```

```
                try{
                        Thread.sleep(50);
                }
                catch(InterruptedException ie){
                        System.out.println("Hijo del hilo interrumpido! " + ie);
                }
        }
        System.out.println("Hijo del hilo finalizada!");
}
public static void main(String[] args) {
        t.start();
        for(int i=0; i < 5; i++){
                System.out.println("Main thread : " + i);
                try{
                        Thread.sleep(100);
                }
                catch(InterruptedException ie){
                        System.out.println("Child thread interrupted! " + ie);
                }
        }
```

```
            System.out.println("Main thread finished!");
        }
}
```

MANEJO DE THREADS

El método start()

Es usado para iniciar el cuerpo del hilo definido por el método run().

Ejemplo:

```
MiHilo elHilo = new MiHilo();
elHilo.start();
```

El método run()

Es usado para definir la tarea concurrente a realizar por el (los) hilo(s).

Ejemplo:

```
public void run() {
    // mientras continuar ...
    while (continuar) {
        String texto = leeDelTeclado();
        enviaAlServidor(texto);
```

```
    }
  }
```

El método join()

Es usado para esperar por el término del hilo sobre la cual el método es invocado, por ejemplo por término de método run(), “se forma en una cola de espera de otro hilo”.

Ejemplo:

```
Thread t = new Thread();
t.start();
t.join();
```

El método yield()

Se encargar de mover un hilo desde el estado de corriendo al final de la cola de procesos en espera por la CPU.

Ejemplo:

```
public class CD_PrimerThread extends Thread {
  public void run() {
    while(true) {
      System.out.print("Primer Hilo ");
      yield();
    }
```

```
  }
}
public class CD_SegundoThread extends Thread {
  public void run() {
    while(true) {
      System.out.print("Segundo Hilo ");
      yield();
    }
  }
}
```

El método sleep()

Se encargar de poner a dormir un hilo por un tiempo mínimo especificado.

Ejemplo:

```
public class Counter
{
   public static void main(String[]args)
   {
      int i;
      for (i = 0; i <= 10; i++)
```

```
      {
         Thread.sleep(1000);
         System.out.println(i);
      }
      System.out.println("Espere un momento por favor…");
   }
}
```

El método suspend()

Se encarga de suspender la ejecución de un hilo temporalmente.

Ejemplo:

```
Thread t = new Thread();
t.suspend();
```

El método resume()

Se encarga de reactivar la ejecución de un hilo, que fue suspendido anteriormente.

Ejemplo:

```
Thread t = new Thread();
t.suspend();
t.resume();
```

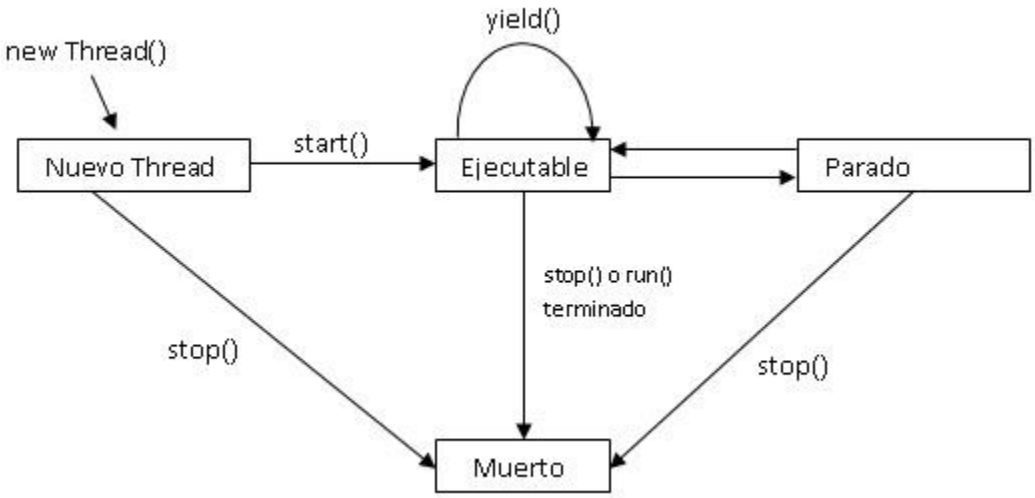
CONTROLANDO LOS THREADS

CICLO DE VIDA

¿Qué es el ciclo de vida?

El ciclo de vida de un Hilo representa los estados por los cuales puede pasar un Hilo desde que nace hasta que muere. Durante el ciclo de vida de un Hilo, este se puede encontrar en diferentes estados. La figura siguiente muestra estos estados y los métodos que provocan el paso de un estado a otro.

Diagrama de ciclo de vida



ESTADOS DE UN THREAD

Estado Nuevo

El Hilo ha sido creado pero no inicializado, es decir, no se ha ejecutado todavía el método start(). Se producirá un mensaje de error (IllegalThreadStateException) si se intenta ejecutar cualquier método de la clase Thread (hilo) distinto de start().

Estado Ejecutable

El Hilo puede estar ejecutándose, siempre y cuando se le haya asignado un determinado tiempo de CPU. En la práctica puede no estar siendo ejecutado en un instante determinado en beneficio de otro thread (hilo).

Estado Bloqueado

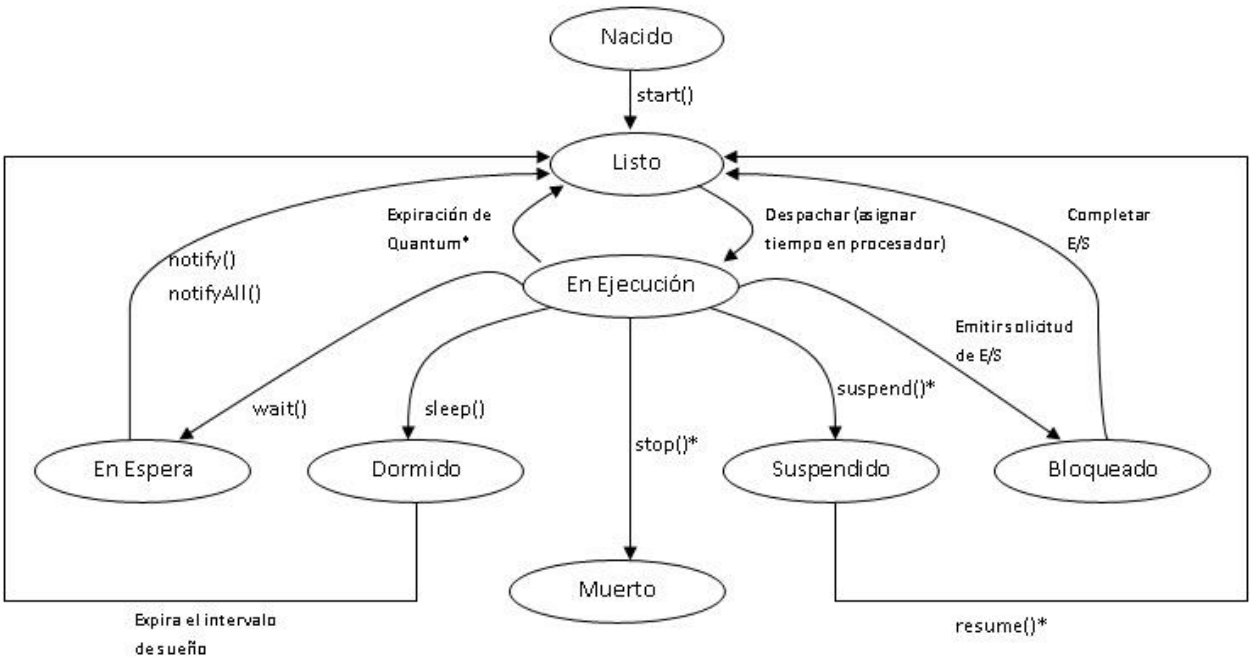
El Hilo podría estar ejecutándose, pero hay alguna actividad interna suya que lo impide, como por ejemplo una espera producida por una operación de escritura o lectura de datos por teclado (Entrada/Salida). Si un Thread (hilo) está en este estado, no se le asigna tiempo de CPU.

Estado Muerto

La forma habitual de que un Hilo muera es finalizando el método run(). También puede llamarse al método stop() de la clase Thread, aunque dicho método es considerado “peligroso” y no debe utilizarse.

Ampliación del diagrama de ciclo de vida

Ampliando los conceptos del diagrama de ciclo de vida, un Hilo desde su ejecución hasta su muerte puede pasar por diversos estados de vida. Estos estados y sus transiciones se muestran en forma más detallada en el diagrama siguiente:



PLANIFICACIÓN DE THREADS

¿Qué significa planificación?

Un tema fundamental dentro de la programación multihilo es la planificación de los hilos. Este concepto se refiere a la política a seguir que el hilo tome el control del procesador y cuando. Obviamente en el caso de que un hilo este bloqueado esperando una operación de Entrada/Salida (E/S) este hilo debería dejar el control del procesador y que este control lo tomara otro hilo que si pudiera hacer uso del tiempo de CPU. ¿Pero qué pasa si hay más de un hilo esperando? ¿A cuál de ellos le otorgamos el control del procesador?

Prioridades

Para determinar que hilo debe ejecutarse primero, cada hilo posee su propia prioridad: un hilo de prioridad más alta que se encuentre en el estado LISTO entrara antes en el estado EN EJECUCION que otro de menor prioridad.

El método setPriority()

Para establecer la prioridad de un hilo se utiliza el método setPriority() de la siguiente manera:

- h1.setPriority(10);
- h1.setPriority(1);

También existen constantes definidas para la asignación de prioridades estas son:

- MIN_PRIORITY = 1
- NORM_PRIORITY = 5
- MAX_PRIORITY = 10

Las cuales se pueden utilizar de la siguiente manera:

- h1.setPriority(Thread. MAX_PRIORITY); //Le concede la mayor prioridad
- h1.setPriority(Thread. MIN_PRIORITY); //Le concede la menor prioridad

El método wait() permite paralizar temporalmente la ejecución de un hilo bajo determinada circunstancia, mientras los métodos notify() y notifyAll() permiten enviar notificaciones individuales y a todos los hilos respectivamente, sobre la terminación de un hilo especifico.

El siguiente ejemplo nos muestra cómo utilizar los métodos wait(), notify() y notifyAll() en un pentágono de tiro:

Archivo: pentágono.java

```
public class Pentagono {

   public static void main(String[] args) {

      Pistola arma = new Pistola();
```

```
        Cargar c = new Cargar(arma, 1);
        Descargar d = new Descargar(arma, 1);
        c.start();
        d.start();
    }
}
```

Archivo: pistola.java

```
public class Pistola {
    private int cartucho;
    private boolean enposicion = true;
    public synchronized void disparar(int cartucho) {
        while (enposicion == false) {
            try {
                // Esperar a apuntar
                wait();
            } catch (InterruptedException e) { }
        }
        enposicion = false;
        notifyAll();
```

```
    }
    public synchronized void apuntar() {
        while (enposicion == true) {
            try {
                // Esperar a disparar
                wait();
            } catch (InterruptedException e) { }
        }
        enposicion = true;
        notifyAll();
    }
}
```

Archivo: Cargar.java

```
public class Cargar extends Thread {
    private Pistola arma;
    private int cartucho;
    public Cargar(Pistola arma, int cartucho) {
        this.arma = arma;
        this.cartucho = cartucho;
```

```
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            arma.apuntar();
            System.out.println("Apuntar #" + this.cartucho
                    + " bala: " + i);
        }
    }
}
```

Archivo: Descargar.java

```
public class Descargar extends Thread {
    private Pistola arma;
    private int cartucho;
    public Descargar(Pistola arma, int cartucho) {
        this.arma = arma;
        this.cartucho = cartucho;
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
```

```
            arma.disparar(i);
            System.out.println("Descargar #" + this.cartucho
                        + " bala: " + i);
        }
    }
}
```

Nota: se deben compilar los cuatros (4) archivos, ejecutándose el archivo pentágono.java que se encargara de instanciar los otros tres (3) archivos.

La importancia de la sincronización

La programación concurrente puede dar lugar a muchos errores debido a la utilización de recursos compartidos que pueden ser alterados. Las secciones de código potencialmente peligrosas de provocar estos errores se conocen como secciones críticas.

En general los programas concurrentes deben ser correctos totalmente. Este concepto es la suma de dos (2) conceptos distintos de la corrección parcial esto es que terminara en algún momento de tiempo finito. En esto programas por lo tanto hay que evitar que varios hilos entren en una sección critica (exclusión mutua) y que los programas se bloqueen (deadlock – abrazo mortal).

Además los programas que no terminan nunca (programas reactivos) deben cumplir la ausencia de inanición (starvation) esto es que tarde o temprano todos los hilos alcancen el control del procesador y ninguno quede indefinidamente en la lista de hilos listos y también deben cumplir la propiedad de equitativita (fairwess) esto es repartir el tiempo de la forma más justa entre todos los hilos.

Un monitor impide que varios hilos accedan al mismo recurso compartido a la vez. Cada monitor incluye un protocolo de entrada y un protocolo de salida. En Java los monitores se consiguen mediante el uso de la palabra reservada syncronized bien como instrucción de bloque o bien como modificador de acceso de métodos.

La keyword synchronized (Exclusion Mutua)

Cuando se utiliza la palabra clave synchronized se está indicando una zona restringida para el uso de Hilos, esta zona restringida para efectos prácticos puede ser considerada un candado ("lock") sobre la instancia del objeto en cuestión.

Lo anterior implica que si es invocado un método synchronized únicamente el hilo que lo invoca tiene acceso a la instancia del objeto, y cualquier otro Thread que intente acceder a esta misma instancia tendrá que esperar hasta que sea terminada la ejecución del método synchronized. En resumen podemos decir:

- Es posible garantizar la ejecución en exclusión mutua de un método definiéndolo como synchronized.
- Los métodos synchronized bloquean el cerrojo del objeto actual, o del objeto Class si el método es estático.
- Si el cerrojo está ocupado, el hilo se suspende hasta que éste es liberado.
- No se ejecutarán simultáneamente dos métodos synchronized de un mismo objeto, pero sí uno que lo sea y cualquier número de otros que no.
- Pueden sobrescribirse métodos synchronized para que no lo sean en las clases nuevas. Sin embargo sí lo seguirá siendo el método super.metodo(...).
- La exclusión mutua es interesante para garantizar la consistencia del estado de los objetos.
- Se suelen utilizar en los métodos que hacen que el objeto pase por estados transitorios que no son correctos. Si también se hacen synchronized los métodos para consultar el estado, se evita que puedan verse estados inestables del objeto.

- Puede bloquearse el cerrojo de un objeto dentro de una porción de código mediante synchronized(objeto) { }.
- Si el cerrojo está bloqueado por un hilo diferente (ya sea porque está ejecutando un método synchronized o porque está dentro de un bloque como el anterior), el hilo actual se suspenderá.
- Pueden usarse estos bloques para clases sin sincronizar que no podamos modificar. Es inseguro, mejor usar herencia.

Candados (locks): todos los objetos (incluidos los arrays) tienen un "candado" (lock). Solo un hilo puede tener bloqueado el candado de un objeto en un momento dado. Podrá bloquearlo más de una vez antes de liberarlo y solo quedará completamente libre cuando el hilo lo libere tantas veces como lo ha obtenido. Si un hilo intenta obtener un candado ocupado, quedará suspendida hasta que éste se libere y pueda obtenerlo. No se puede acceder directamente a los candados.

Los métodos wait() y notify() - notifyAll()

Los mecanismos anteriores sirven para evitar la interferencia entre hilos. Es necesario algún método de comunicación entre ellos. Todos los objetos implementan los métodos wait() y notify(). Mediante wait() se suspende el hilo actual hasta que algún otro llame al método notify() del mismo objeto. Todos los objetos tienen una lista de hilos que están esperando que alguien llame a notify().

Estos métodos están pensados para avisar de cambios en el estado del objeto a hilos que están esperando dichos cambios:

```
synchronized void hacerMientrasCondicion() {

while(!condicion) wait();
```

```
/* ... */

}

synchronized void cambiarCondicion() {

/*...*/

notify();

}
```

Tanto el método wait() como el notify() deben ejecutarse dentro de métodos synchronized.

Cuando se llama a wait() se libera el cerrojo del objeto que se tiene bloqueado y se suspende el hilo, de forma atómica. Cuando se llama a notify() se despierta uno de los hilos que estaban esperando. Ésta competirá con cualquier otra por volver a obtener el candado del objeto, sin tener ningún tipo de prioridad. De ahí es que sea mejor usar while(!condicion) wait(); en resumen:

- Todos los métodos anteriores son finales.
- Todas las variantes de wait(...) pueden lanzar la excepción InterruptedException.
- notify() despierta un hilo cualquiera. No se garantiza que sea el que más tiempo lleva esperando. Es solo interesante cuando se está seguro de que solo habrá un hilo esperando. Es peligroso.
- notifyAll() despierta a todos los hilos. Es más seguro. Requiere más que nunca el uso de while() en lugar de if(!condition) wait();

Interbloqueos

La existencia de varios hilos, y el uso de la exclusión mutua pueden ocasionar la aparición de interbloqueos (abrazo mortal), en el que ninguno de dos hilos puede ejecutarse porque están esperando algo de la otra.

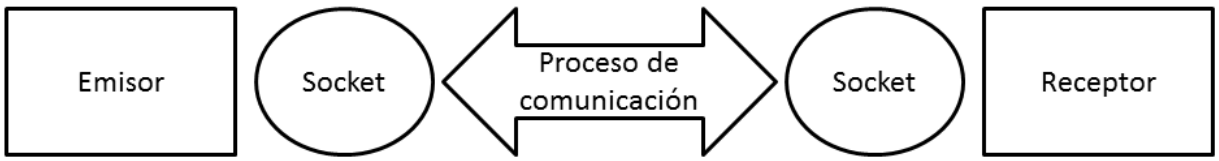
INTRODUCCIÓN A NETWORKING

¿Qué es Networking?

Networking se refiere a la posibilidad de trabajar con diversas aplicaciones ubicadas físicamente en distintas estaciones de trabajo y permitir que se conecten vía una red, para trabajar de manera cooperativa o simplemente enviar y recibir información.

¿Qué es un socket?

Un Socket es una representación abstracta del extremo (endpoint) en un proceso de comunicación. Para que se dé la comunicación en una Red, el proceso de comunicación requiere un Socket a cada extremo Emisor/Receptor y viceversa.



La comunicación con sockets sigue el modelo Cliente/Servidor/Cliente. En la mayoría de los casos un programa Servidor fundamentalmente envía datos, mientras que un programa Cliente recibe esos datos, aunque es raro que un programa exclusivamente reciba o envíe datos. Una distinción confiable se logra si consideramos Cliente al programa que inicia la comunicación y Servidor al programa que espera a que algún otro inicie la comunicación con él.

La comunicación con sockets se hace mediante un protocolo de la familia TCP/IP en la mayoría de los casos, y es el programador quien decide qué protocolo utilizar dependiendo de la necesidades de la aplicación a desarrollar.

Socket designa un concepto abstracto por el cual dos programas (posiblemente situados en computadoras distintas) pueden intercambiarse cualquier flujo de datos, generalmente de manera fiable y ordenada.

Un socket queda definido por una dirección IP, un protocolo y un número de puerto.

Para que dos programas puedan comunicarse entre sí es necesario que se cumplan ciertos requisitos:

- Que un programa sea capaz de localizar al otro.
- Que ambos programas sean capaces de intercambiarse cualquier secuencia de octetos, es decir, datos relevantes a su finalidad.

Para ello son necesarios los tres recursos que originan el concepto de socket:

- Un protocolo de comunicaciones, que permite el intercambio de octetos.
- Una dirección del Protocolo de Red (Dirección IP, si se utiliza el Protocolo TCP/IP), que identifica una computadora.
- Un número de puerto, que identifica a un programa dentro de una computadora.

Los sockets permiten implementar una arquitectura cliente-servidor. La comunicación ha de ser iniciada por uno de los programas que se denomina programa cliente. El segundo programa espera a que otro inicie la comunicación, por este motivo se denomina programa servidor.

Características de un socket

Las propiedades de un socket dependen de las características del protocolo en el que se implementan. El protocolo más utilizado es TCP, aunque también es posible utilizar UDP o IPX. Gracias al protocolo TCP, los sockets tienen las siguientes propiedades:

- Orientado a conexión.
- Se garantiza la transmisión de todos los octetos sin errores ni omisiones.
- Se garantiza que todo octeto llegará a su destino en el mismo orden en que se ha transmitido.

Estas propiedades son muy importantes para garantizar la corrección de los programas que tratan la información.

El protocolo UDP es un protocolo no orientado a la conexión. Sólo se garantiza que si un mensaje llega, llegue bien. En ningún caso se garantiza que llegue o que lleguen todos los mensajes en el mismo orden que se mandaron. Esto lo hace adecuado para el envío de mensajes frecuentes pero no demasiado importantes, como por ejemplo, mensajes para los refrescos (actualizaciones) de un gráfico.

IDENTIFICACIÓN DE PROCESOS

¿Qué es una dirección IP?

Es un número que identifica de manera lógica y jerárquica a una interfaz de un dispositivo (habitualmente una computadora) dentro de una red que utilice el protocolo IP (Internet Protocol), que corresponde al nivel de red del protocolo TCP/IP. Dicho número no se debe confundir con la dirección MAC que es un número hexadecimal fijo que es asignado a la tarjeta o dispositivo de red por el fabricante, mientras que la dirección IP se puede cambiar.

Es habitual que un usuario que se conecta desde su hogar a Internet utilice una dirección IP. Esta dirección puede cambiar cada vez que se conecta; y a esta forma de

asignación de dirección IP se denomina una dirección IP dinámica (normalmente se abrevia como IP dinámica).

Los sitios de Internet que por su naturaleza necesitan estar permanentemente conectados, generalmente tienen una dirección IP fija (se aplica la misma reducción por IP fija o IP estática), es decir, no cambia con el tiempo. Los servidores de correo, DNS, FTP públicos, y servidores de páginas web necesariamente deben contar con una dirección IP fija o estática, ya que de esta forma se permite su localización en la red.

¿Qué es un puerto?

Es una dirección numérica a través de la cual se procesa un servicio, es decir, no son puertos físicos semejantes al puerto paralelo para conectar la impresora, sino que son direcciones lógicas proporcionadas por el sistema operativo para poder responder.

Las comunicaciones de información relacionada con Web tienen lugar a través del puerto 80 mediante protocolo TCP. Para emular esto en Java, se utiliza la clase Socket.

Teóricamente hay 65535 puertos disponibles, aunque los puertos del 1 al 1023 están reservados al uso de servicios estándar proporcionados por el sistema, quedando el resto libre para utilización por las aplicaciones de usuario. De no existir los puertos, solamente se podría ofrecer un servicio por máquina. Nótese que el protocolo IP no sabe nada al respecto de los números de puerto.

Se puede decir que IP pone en contacto las máquinas, TCP y UDP (protocolos de transmisión) establecen un canal de comunicación entre determinados procesos que se ejecutan en tales equipos y, los números de puerto se pueden entender como números de oficinas dentro de un gran edificio. El edificio (equipo), tendrá una única dirección IP, pero dentro de él, cada tipo de negocio, en este caso HTTP, FTP, etc., dispone de una oficina individual.

¿Qué es la URL (Uniform Resource Locator)?

Una URL, o dirección, es en realidad un puntero a un determinado recurso de un determinado sitio de Internet. Al especificar una URL, se está indicando:

- El protocolo utilizado para acceder al servidor (http, por ejemplo)
- El nombre del servidor
- El puerto de conexión (opcional)
- El camino
- El nombre de un archivo determinado en el servidor (opcional a veces)
- Un punto de referencia dentro del archivo (opcional)

La sintaxis general, para una dirección URL, sería:

protocolo://nombre_servidor[:puerto]/directorio/archivo#referencia

El puerto es opcional y normalmente no es necesario especificarlo si se está accediendo a un servidor que proporcione sus servicios a través de los puertos estándar; tanto el navegador como cualquier otra herramienta que se utilice en la conexión conocen perfectamente los puertos por los cuales se proporciona cada uno de los servicios e intentan conectarse directamente a ellos por defecto.

Networking en JAVA

Utilización de Sockets

Introducción a la programación con Sockets

Java proporciona dos formas diferentes de manejar la programación de comunicaciones a través de red, al menos en lo que a la comunicación web concierne. Por un lado están las

clases Socket, DatagramSocket y ServerSocket, y por otro lado están las clases URL, URLEncoder y URLConnection.

Los sockets son puntos finales de enlaces de comunicaciones entre procesos. Los procesos los tratan como descriptores, de forma que se pueden intercambiar datos con otros procesos transmitiendo y recibiendo a través de sockets. El tipo de sockets describe la forma en la que se transfiere información a través de ese socket.

¿Qué son los Stream Sockets (TCP)?

Son un servicio orientado a conexión, donde los datos se transfieren sin encuadrarlos en registros o bloques. Si se rompe la conexión entre los procesos, éstos serán informados de tal suceso para que tomen las medidas oportunas.

El protocolo de comunicaciones con streams es un protocolo orientado a conexión, ya que para establecer una comunicación utilizando el protocolo TCP, hay que establecer en primer lugar una conexión entre un par de sockets. Mientras uno de los sockets atiende peticiones de conexión (servidor), el otro solicita una conexión (cliente). Una vez que los dos sockets estén conectados, se pueden utilizar para transmitir datos en ambas direcciones.

¿Que son los Datagrama Sockets (UDP)?

Son un servicio de transporte sin conexión. Son más eficientes que TCP, pero en su utilización no está garantizada la fiabilidad. Los datos se envían y reciben en paquetes, cuya entrega no está garantizada. Los paquetes pueden ser duplicados, perdidos o llegar en un orden diferente al que se envió.

El protocolo de comunicaciones con datagramas es un protocolo sin conexión, es decir, cada vez que se envíen datagramas es necesario enviar el descriptor del socket local y la

dirección del socket que debe recibir el datagrama. Como se puede ver, hay que enviar datos adicionales cada vez que se realice una comunicación, aunque tiene la ventaja de que se pueden indicar direcciones globales y el mismo mensaje llegará a una o muchas máquinas a la vez.

Stream Sockets vs. Datagram Sockets

La decisión depende de la aplicación cliente/servidor que se esté escribiendo; aunque hay algunas diferencias entre los protocolos que sirven para ayudar en la decisión y decantar la utilización de sockets de un tipo.

En UDP, cada vez que se envía un datagrama, hay que enviar también el descriptor del socket local y la dirección del socket que va a recibir el datagrama, luego los mensajes son más grandes que los TCP. Como el protocolo TCP está orientado a conexión, hay que establecer esta conexión entre los dos sockets antes de nada, lo que implica un cierto tiempo empleado en el establecimiento de la conexión, que no es necesario emplear en UDP.

En UDP hay un límite de tamaño de los datagramas, establecido en 64 kilobytes, que se pueden enviar a una localización determinada, mientras que TCP no tiene límite; una vez que se ha establecido la conexión, el par de sockets funciona como los streams: todos los datos se leen inmediatamente, en el mismo orden en que se van recibiendo.

UDP es un protocolo desordenado, no garantiza que los datagramas que se hayan enviado sean recibidos en el mismo orden por el socket de recepción. Al contrario, TCP es un protocolo ordenado, garantiza que todos los paquetes que se envíen serán recibidos en el socket destino en el mismo orden en que se han enviado.

Los datagramas son bloques de información del tipo enviar y olvidar. Para la mayoría de los programas que utilicen la red, el usar un flujo TCP en vez de un datagrama UDP es más sencillo y hay menos posibilidades de tener problemas. Sin embargo, cuando se requiere un

rendimiento óptimo, y está justificado el tiempo adicional que supone realizar la verificación de los datos, la comunicación a través de sockets TCP es un mecanismo realmente útil.

En resumen, TCP parece más indicado para la implementación de servicios de red como un control remoto (rlogin, telnet) y transmisión de archivos (ftp); que necesitan transmitir datos de longitud indefinida. UDP es menos complejo y tiene una menor sobrecarga sobre la conexión; esto hace que sea el indicado en la implementación de aplicaciones cliente/servidor en sistemas distribuidos montados sobre redes de área local.

IMPLEMENTACION EN JAVA

La clase URL

Para comenzar con la programación de sockets, resulta necesario comprender las clases que ofrece Java. En primer lugar, la clase URL contiene constructores y métodos para la manipulación de URL (Universal Resource Locator): un objeto o servicio en Internet. El protocolo TCP necesita dos tipos de información: la dirección IP y el número de puerto. Vamos a ver cómo podemos recibir entonces la página Web siguiente:

http://www.google.com

En primer lugar, Google tiene registrado su nombre, permitiendo que se use google.com como su dirección IP, o lo que es lo mismo, cuando indicamos google.com es como si hubiésemos indicado 74.125.21.147, su dirección IP real.

Si queremos obtener la dirección IP real de la red en que estamos corriendo, podemos realizar llamadas a los métodos getLocalHost() y getAddress(). Primero, getLocalHost() nos devuelve un objeto iNetAddress, que si usamos con getAddress() generará un array con los cuatro bytes de la dirección IP, por ejemplo:

InetAddress direccion = InetAddress.getLocalHost();

byte direccionIp[] = direccion.getAddress();

Si la dirección de la máquina en que estamos corriendo es 150.150.112.145, entonces:

direccionIp[0] = 150 direccionIp[1] = 150 direccionIp[2] = 112 direccionIp[3] = 145

Por otro lado, podemos especificar las partes que componen una url y así podríamos construir un objeto de tipo url utilizando los siguientes constructores:

URL (“http","www.google.com","80","index.html”);

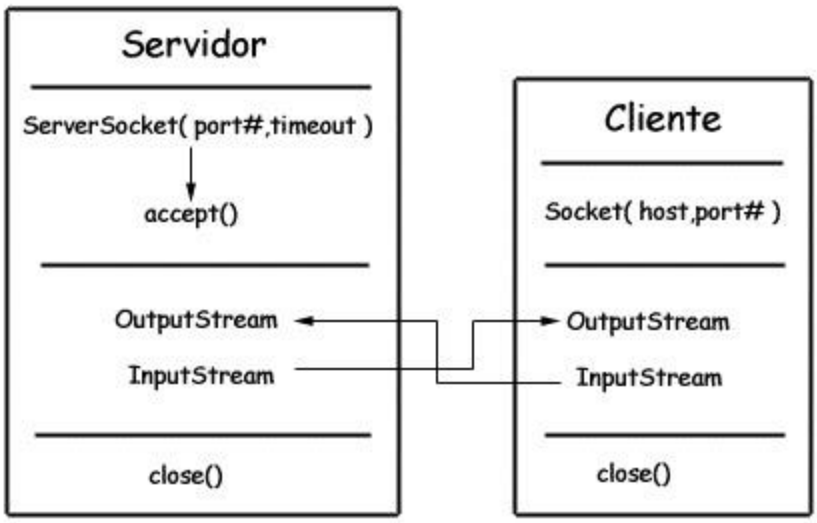
o dejar que los sistemas utilicen todos los valores por defecto que tienen definidos, como en:

URL (“http://www.google.com”);

Y en los dos casos obtendríamos la visualización de la página principal de Google en nuestro navegador.

Arquitectura de comunicaciones

En Java, crear una conexión socket TCP/IP se realiza directamente con el paquete java.net. A continuación mostramos un diagrama de lo que ocurre en el lado del cliente y del servidor:



El modelo de sockets más simple es:

- El servidor establece un puerto y espera durante un cierto tiempo (timeout segundos), a que el cliente establezca la conexión. Cuando el cliente solicite una conexión, el servidor abrirá la conexión socket con el método accept().
- El cliente establece una conexión con la máquina host a través del puerto que se designe en puerto#
- El cliente y el servidor se comunican con manejadores InputStream y OutputStream

La clase Socket

Si estamos programando un cliente, el socket se abre de la siguiente forma:

Socket miCliente; miCliente = new Socket (“maquina", numeroPuerto);

Donde “maquina” es el nombre de la computadora (host) en donde estamos intentando abrir la conexión y “numeroPuerto” es el puerto (un número) del servidor que está corriendo sobre el cual nos queremos conectar. Para las aplicaciones que se desarrollen, asegurarse de seleccionar un puerto por encima del 1023. (Los puertos del 1 al 1023 están reservados por el sistema).

En el ejemplo anterior no se usan excepciones; sin embargo, es una gran idea la captura de excepciones cuando se está trabajando con sockets. El mismo ejemplo quedaría como:

```
Socket miCliente;

try {

        miCliente = new Socket( "maquina",numeroPuerto );

} catch( IOException e ) {

        System.out.println(e);

}
```

La clase ServerSocket

Si estamos programando un servidor, la forma de apertura del socket es la que muestra el siguiente ejemplo:

```
Socket miServicio;

try {

        miServicio = new ServerSocket(numeroPuerto);

} catch( IOException e ) {

        System.out.println(e);

}
```

A la hora de la implementación de un servidor también necesitamos crear un objeto socket desde el ServerSocket para que esté atento a las conexiones que le puedan realizar clientes potenciales y poder aceptar esas conexiones:

```
Socket socketServicio = null;

try {

        socketServicio = miServicio.accept();

} catch( IOException e ) {

         System.out.println( e );

}
```

Creación de Streams de Entrada

En la parte cliente de la aplicación, se puede utilizar la clase DataInputStream para crear un stream de entrada que esté listo a recibir todas las respuestas que el servidor le envíe.

```
DataInputStream entrada;

try {

        entrada = new DataInputStream( miCliente.getInputStream() );

} catch (IOException e) {

        System.out.println( e );

}
```

La clase DataInputStream permite la lectura de líneas de texto y tipos de datos primitivos de Java de un modo altamente portable; dispone de métodos para leer todos esos tipos como: read(), readChar(), readInt(), readDouble() y readLine(). Deberemos utilizar la función que creamos necesaria dependiendo del tipo de dato que esperemos recibir del servidor.

En el lado del servidor, también usaremos DataInputStream, pero en este caso para recibir las entradas que se produzcan de los clientes que se hayan conectado:

```
DataInputStream entrada;
try {
    entrada = new DataInputStream( socketServicio.getInputStream());
} catch( IOException e ) {
    System.out.println( e );
}
```

Creación de Streams de Salida

En el lado del cliente, podemos crear un stream de salida para enviar información al socket del servidor utilizando las clases PrintStream o DataOutputStream:

```
PrintStream salida;
try {
    salida = new PrintStream( miCliente.getOutputStream() );
} catch( IOException e ) {
    System.out.println( e );
}
```

La clase PrintStream tiene métodos para la representación textual de todos los datos primitivos de Java. Sus métodos write y println() tienen una especial importancia en este

aspecto. No obstante, para el envío de información al servidor también podemos utilizar DataOutputStream:

```
DataOutputStream salida;

try {

        salida = new DataOutputStream( miCliente.getOutputStream() );

} catch( IOException e ) {

        System.out.println( e );

}
```

La clase DataOutputStream permite escribir cualquiera de los tipos primitivos de Java, muchos de sus métodos escriben un tipo de dato primitivo en el stream de salida. De todos esos métodos, el más útil quizás sea writeBytes().

En el lado del servidor, podemos utilizar la clase PrintStream para enviar información al cliente:

```
PrintStream salida;

try {

        salida = new PrintStream (socketServicio.getOutputStream());

} catch( IOException e ) {

        System.out.println( e );

}
```

Pero también podemos utilizar la clase DataOutputStream como en el caso de envío de información desde el cliente.

Cierre de Sockets

Siempre deberemos cerrar los canales de entrada y salida que se hayan abierto durante la ejecución de la aplicación. En la parte del cliente:

```
try {
        salida.close();
        entrada.close();
        miCliente.close();
} catch( IOException e ) {
        System.out.println( e );
}
```

Y en la parte del servidor:

```
try {
        salida.close(); entrada.close();
        socketServicio.close();
        miServicio.close();
} catch( IOException e ) {
        System.out.println( e );
}
```

CONSTRUCCIÓN DE UN SERVIDOR TCP/IP

La clase Servidor TCP/IP

Veamos el código que presentamos en el siguiente ejemplo, donde desarrollamos un servidor TCP/IP, para el cual desarrollaremos luego su cliente TCP/IP. La aplicación servidor TCP/IP depende de una clase de comunicaciones proporcionada por Java: ServerSocket. Esta clase realiza la mayor parte del trabajo de crear un servidor.

Ejemplo:

```
import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.io.PrintStream;

import java.net.ServerSocket;

import java.net.Socket;

public class Servidor {

   public static void main(String [] args) {

      ServerSocket  s; //Socket servidor

      Socket  sc; //Socket cliente

      PrintStream  p; //Canal de escritura

      BufferedReader  b; //Canal de Lectura

      String  mensaje;

      try {
```

```
//Creo el socket server
s = new ServerSocket (9999);
//Invoco el metodo accept del socket servidor, me devuelve una referencia al socket cliente
sc = s.accept();
//Obtengo una referencia a los canales de escritura y lectura del socket cliente
b = new BufferedReader ( new InputStreamReader  ( sc.getInputStream() ) );
p = new PrintStream  ( sc.getOutputStream() );
while ( true ) {
   //Leo lo que escribio el socket cliente en el canal de lectura
   mensaje = b.readLine();
   System .out.println(mensaje);
   //Escribo en canal de escritura el mismo mensaje recibido
   p.println(mensaje);
   if ( mensaje.equals("by")) {
      break;
   }
}
p.close();
b.close();
sc.close();
```

```
            s.close();
        } catch (IOException  e) {
            System .out.println("No puedo crear el socket");
        }
    }
}
```

La clase Cliente TCP/IP

El lado cliente de una aplicación TCP/IP descansa en la clase Socket. De nuevo, mucho del trabajo necesario para establecer la conexión lo ha realizado la clase Socket. Vamos a presentar ahora el código de nuestro cliente más simple, que encaja con el servidor presentado antes. El trabajo que realiza este cliente es que todo lo que recibe del servidor lo imprime por la salida estándar del sistema.

Ejemplo:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintStream;
import java.net.Socket;
import java.net.UnknownHostException;
public class Cliente {
    public static void main(String [] args) {
```

```
Socket  s;
PrintStream  p;
BufferedReader  b;
String  host = "localhost";
int port = 9999;
String  respuesta;
//Referencia a la entrada por consola (System.in)
BufferedReader  in = new BufferedReader (new InputStreamReader (System .in));
try {
   //Creo una conexion al socket servidor
   s = new Socket (host,port);
   //Creo las referencias al canal de escritura y lectura del socket
   p = new PrintStream (s.getOutputStream());
   b = new BufferedReader  ( new InputStreamReader  ( s.getInputStream() ) );
   while ( true ) {
      //Ingreso un String por consola
      System .out.print("Mensaje a enviar: ");
      //Escribo en el canal de escritura del socket
      p.println( in.readLine() );
      //Espero la respuesta por el canal de lectura
```

```
            respuesta = b.readLine();
            System .out.println(respuesta);
            if ( respuesta.equals("by")) {
               break;
            }
         }
         p.close();
         b.close();
         s.close();
      } catch (UnknownHostException  e) {
         System .out.println("No puedo conectarme a " + host + ":" + port);
      } catch (IOException  e) {
         System .out.println("Error de E/S en " + host + ":" + port);
      }
   }
}
```

Nota: se deben compilar los dos (2) archivos, ejecutándose ambos por separados (1ro el servidor), en el archivo cliente se encargara de enviar información al servidor, el servidor solo recibirá la información y la mostrara por pantalla.