

# Práctica 5 – Conmutando paquetes Ethernet

## 1. Introducción

En prácticas anteriores se han utilizado los conmutadores Ethernet del laboratorio y entre otras cosas se ha visto su comportamiento en cuanto al aprendizaje de direcciones MAC para el reenvío de paquetes, cómo estas direcciones caducan al tiempo si no se recibe tráfico de una de ellas o si se desconecta el cable de la interfaz, así como la posibilidad de que existan pérdidas de paquetes en el conmutador si la tasa de recepción es superior a la capacidad de este para reenviarlas.

En esta práctica se tratará de ver en detalle estos aspectos mediante la implementación en software de un conmutador Ethernet, de forma que no simplemente se vean los efectos desde fuera, sino también los mecanismos internos que permiten a los conmutadores realizar su función.

El objetivo de esta práctica es interiorizar el funcionamiento de estos mecanismos mediante la implementación en Java de un Conmutador Ethernet.

## 2. Topología y entorno

Se va a utilizar el PC B para simular un conmutador que permita comunicar varios PCs. Dado que en el laboratorio solo dispone de dos PCs adicionales, tendrán que usarse varias interfaces del mismo PC. A efectos prácticos, cada interfaz se comportará como un PC independiente, ya que el conmutador desconoce qué hay al otro lado de un enlace. Por simplicidad se recomienda utilizar distintas interfaces de un mismo PC, pero puede utilizar interfaces de distintos PCs si lo prefiere. Interconecte 3 interfaces con PC B. Levante las interfaces y valide mediante ethSend y Wireshark que la comunicación funciona adecuadamente desde los extremos hasta PC B.

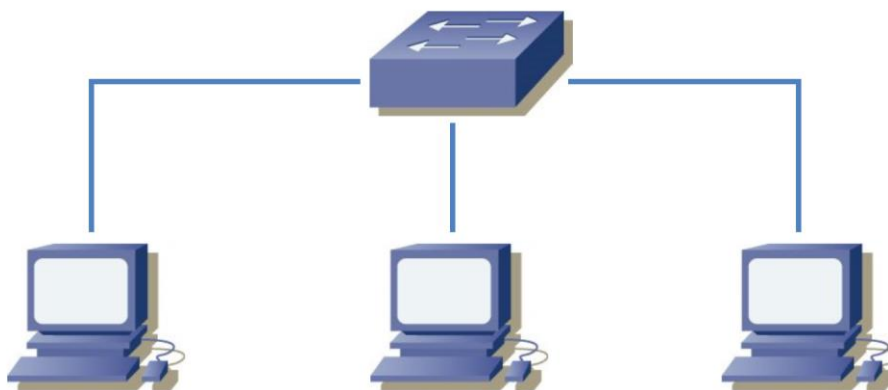


Figura 1 – PCs conectados a un conmutador virtual.

Para implementar un conmutador, este deberá ser capaz de leer las tramas que reciba por todas las interfaces que se quieren conmutar (identificando la interfaz por la que se recibe), realizar la toma de decisiones correspondiente para decidir la interfaz de salida, y realizar el reenvío de la trama.

### 2.1 Captura de tráfico

Para obtener desde el Conmutador acceso a las tramas recibidas se pide utilizar tshark de forma similar a lo que se ha hecho en prácticas anteriores. En este caso es importante tener cuidado y limitar las tramas que se capturan, ya que por defecto tshark imprimirá tanto las tramas enviadas como las recibidas. Si el conmutador recibiese para reenvío también las tramas enviadas, se generaría un bucle infinito donde al reenviar una trama se generaría nuevamente otra a la entrada del conmutador. A su vez, es posible que el conmutador reciba tramas que van dirigidas a él y que por tanto no debe reenviar.

Para evitar estos problemas, se pide capturar exclusivamente las tramas recibidas con dirección MAC destino desconocidas (no pertenecen a PC B), aplicando un filtro de captura que excluya las salientes y aquellas cuyo destino sea una de las interfaces del propio conmutador. Puede quedarse exclusivamente con el tráfico

entrante utilizando la directiva `inbound` en el filtro de captura. Deberá identificar las direcciones MAC de las interfaces del equipo y recordar cómo excluirlas en el filtro de captura. Utilice el comando `tshark-raw-frames`, que es un programa que se ha habilitado específicamente en los PCs del laboratorio y que genera un formato de salida más cómodo para lo que se requiere en esta práctica. Si tiene interés puede ver y analizar su implementación.

```
$ tshark-raw-frames -i eth0 -i eth1 -i eth2 -f "inbound and <mac_filter>"
```

Asumiendo una correcta selección del filtro de captura e interfaces, mediante este comando se conseguirá capturar todo el tráfico de interés e imprimir por pantalla el nombre de la interfaz desde la que se recibe este paquete y el contenido del mismo en hexadecimal.

Inicie un Wireshark y el `tshark` indicado anteriormente. Pruebe a enviar y recibir tramas y compruebe que en `tshark` únicamente visualiza las deseadas. Inspeccione el contenido del paquete hexadecimal que le imprime `tshark`, compárelo con lo que ve en Wireshark e interprételo con la ayuda de la disección que realiza Wireshark.

## 2.2. Preparando el envío

Para realizar el reenvío de las tramas se pide utilizar `ethSend`. El conmutador deberá decidir por qué interfaz (o interfaces) debe reenviar cada trama. Asumiendo que esta decisión se toma correctamente, este deberá extraer las direcciones MAC origen y destino, el *ethertype* y el *payload* de la trama Ethernet, de forma que pueda reconstruir y reenviar el mismo paquete por otra interfaz. Busque en la ayuda de `ethSend` las opciones que necesita para enviar un paquete indicando estos 4 campos.

Seleccione uno de los paquetes capturados en `tshark` y extraiga los valores que necesita. Pruebe a realizar manualmente el reenvío de alguno de ellos y compruebe en Wireshark que el paquete recibido y reenviado coinciden a excepción de la interfaz desde la que se han capturado.

### 3. Hub Ethernet

Un hub<sup>1</sup> es el dispositivo predecesor de los actuales switches de comunicaciones. Estos equipos permitían interconectar varios equipos en una LAN Ethernet y se encargaban de reenviar la señal recibida por todos los puertos a excepción del puerto por el que se recibía (trabajaban a nivel físico así que no entendía las tramas ni por lo tanto podían procesarlas y tomar decisiones según su contenido). A efectos prácticos se comporta de forma muy similar a un switch que siempre realiza inundación, pero no hacen almacenamiento y reenvío.

Para la primera versión del switch se pide implementar uno que realice inundación de todos los paquetes recibidos, de forma que se parezca un poco a lo que hacía un hub. Al iniciar el switch virtual será necesario indicarle por argumentos las interfaces en las que debe operar, de forma que las tenga disponibles cuando necesite hacer inundación. Este deberá recibir el tráfico por *stdin* y leerlo línea a línea utilizando la clase *Scanner*.

Comience por preparar un programa en Java que reciba como argumentos una lista de interfaces y el flujo de tráfico de tshark por *stdin*, separe la interfaz del contenido de la trama e imprima ambos por *stdout*.

```
$ tshark-raw-frames ... | java Switch eth0 eth1 eth2
eth0 ff:ff:ff:ff:ff:ff:01:02:03:04:05:06:08:00:aa:aa:aa:..
```

Procese ahora el contenido de la trama para extraer las direcciones MAC origen y destino, el *ethertype* y el *payload*. Imprimálas por *stdout*.

```
$ tshark-raw-frames ... | java Switch eth0 eth1 eth2
eth0 01:02:03:04:05:06 ff:ff:ff:ff:ff:ff 0x0800 aa:aa:aa:..
```

Desde Java es posible invocar comandos del sistema, como pudieran ser *ls*, *mkdir*, *echo*, o en este caso *ethSend*, para realizar las acciones que estos comandos llevan a cabo y recoger su salida. En muchos lenguajes de programación tendrá esta funcionalidad bajo el nombre de *system* o *exec*<sup>2</sup>. En este caso se utilizará la clase *ProcessBuilder*<sup>3</sup> ya que nos permite recuperar las salidas estándar y de error muy cómodamente. El constructor de *ProcessBuilder* acepta un array de *Strings* y los utiliza para construir la llamada al sistema. Debe entenderse que el resultado de esta ejecución será relativamente equivalente a colocar directamente en la *shell* los distintos elementos separados por espacios, tal y como se indican<sup>4</sup>.

Añada a su programa la siguiente función:

---

<sup>1</sup> <https://www.redeszone.net/tutoriales/redes-cable/switch-vs-hub/>

<sup>2</sup> <https://docs.oracle.com/javase/8/docs/api/java/lang/Runtime.html#exec-java.lang.String->

<sup>3</sup> <https://docs.oracle.com/javase/8/docs/api/java/lang/ProcessBuilder.html>

<sup>4</sup> [https://owasp.org/www-community/attacks/Command\\_Injection](https://owasp.org/www-community/attacks/Command_Injection)

```

public static void sendFrame(String interfaceName, String srcMAC, String
dstMAC, String ethertype, String payload) {
    try {
        String[] command = new String[] {
            "sudo", "ethSend",
            "-i", interfaceName,
            "-s", srcMAC,
            "-d", dstMAC,
            // ethertype
            // payload
        };
        Process p = new ProcessBuilder(command).inheritIO().start();
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}

```

Añada al array `command` los argumentos de `ethSend` necesarios para indicar el *ethertype* y el *payload* de la trama. Asegúrese de que el formato de estos argumentos coincide con el formato que espera `ethSend`.

Utilice esta función para conseguir que el Conmutador reenvíe todos los paquetes recibidos por el resto de las interfaces. Inicie desde los extremos flujos de tramas sin fin con intervalos entre paquetes de al menos 1 segundo. Configure al menos los siguientes flujos:

- Uno dirigido a broadcast.
- Uno dirigido a una dirección unicast de otro extremo.
- Uno dirigido a una de las interfaces de PC B.
- Uno dirigido a una dirección unicast que no se encuentre en la red.
- Uno dirigido a la dirección MAC de la interfaz que realiza el envío.

Valide que cada flujo recibe el tratamiento esperado en el conmutador.

Punto de control 1 (40%): Muestre el conmutador funcionando y cómo ha modelado las tramas ethernet en su programa.

## 4. Aprendizaje de direcciones MAC

Implemente la tabla de direcciones MAC en el Conmutador.

Para ello deberá utilizar los paquetes recibidos para mantener un registro de qué interfaz debe utilizar para alcanzar una dirección MAC conocida. Posteriormente deberá utilizar esta información para evitar la inundación cuando sea posible.

**Nota:** *Tenga en cuenta que una dirección MAC solo debe ser alcanzable a través de una de las interfaces del conmutador.*

Adicionalmente, las entradas de la tabla de direcciones MAC de un conmutador caducan automáticamente cuando este lleva cierto tiempo sin recibir paquetes desde dicho origen. Se pide que su conmutador implemente esta funcionalidad con una caducidad de 10 segundos.

Registre para cada dirección MAC una marca temporal con la fecha de recepción del último paquete. Para simular de forma sencilla el comportamiento descrito sin requerir temporizadores en su programa que le avisen de tener que borrar entradas en la tabla se le sugiere que cuando obtenga un paquete que debe reenviar compruebe la diferencia de tiempo desde el momento actual a la del último paquete recibido para dicha dirección MAC, y en caso de ser superior al umbral, elimine dicha entrada de la tabla de direcciones y realice inundación de esa trama. Puede utilizar `System.currentTimeMillis`<sup>5</sup> para obtener los tiempos que necesita. Imprima por `stderr` un mensaje que avise cuando se eliminan entradas por caducidad.

Asegúrese de que tiene configurado un flujo unicast dirigido a una dirección MAC que está generando tráfico. Valide que el flujo de broadcast se reenvía por todas las interfaces mientras que el unicast solo se reenvía por la que corresponde. Pare el flujo originado desde el destino del flujo unicast y espere hasta que se active el mecanismo de caducidad. Valide que desde ese momento el flujo unicast empieza a reenviarse por inundación.

Punto de control 2 (40%): Muestre el conmutador funcionando y explique las modificaciones realizadas en el programa.

---

<sup>5</sup> <https://docs.oracle.com/javase/8/docs/api/java/lang/System.html#currentTimeMillis-->

## 5. Desconexión de enlaces

En este punto se dispone de un Conmutador virtual que es capaz de aprender direcciones MAC e implementa un mecanismo de caducidad por tiempo para eliminar de la tabla aquellas para las que no se ha recibido tráfico reciente (aunque no implementa un tamaño limitado de tabla de direcciones MAC). Si se desconecta un equipo de una de las interfaces y se conecta en otra, el conmutador podría seguir reenviando el tráfico dirigido a este equipo por la interfaz equivocada hasta que la entrada de la tabla caduque o el equipo genere nuevo tráfico que fuerce una actualización de la tabla. ¿Qué tendría que hacer para forzar este escenario y observar el comportamiento del conmutador?

Para resolver este problema, los conmutadores reaccionan ante la desconexión de una interfaz invalidando automáticamente todas las entradas de la tabla de direcciones que utilizaban dicho interfaz, volviendo a realizar inundación con las tramas dirigidas a los destinos afectados.

En linux el comando `ip` nos permite monitorizar cambios en el estado de las interfaces de red en tiempo real. Ejecute `ip monitor link` en una terminal y pruebe a desconectar y reconectar físicamente alguna de las interfaces de PC B. Observe la salida generada.

```
$ ip monitor link
```

Este comando también le notificaría si modifica el estado de una interfaz con `ifconfig`, pero aquí tanto Wireshark como `tshark` se cerrarían si estaban usando dicha interfaz. Para evitar estos problemas, nos limitaremos a la desconexión física de las interfaces.

Se requiere que el conmutador ejecute este comando internamente con `ProcessBuilder` para recibir estas notificaciones y poder actuar en consecuencia. Anteriormente se ha utilizado para ejecutar `ethSend` y se han redirigido las salidas del programa con `inheritIO` para no tener que gestionarlas manualmente. En este caso hace falta recibir la salida del programa dentro de Java para poder actuar ante la desconexión de interfaces. Puede utilizar la clase `Scanner` para leer la salida de este proceso de forma idéntica a como se utiliza para leer la entrada estándar del programa, simplemente cambiando `System.in` por `p.getInputStream()`.

```
String[] command = new String[] {"ip", "monitor", "link"};
Process p = new ProcessBuilder(command).start();
Scanner sc = new Scanner(p.getInputStream());
```

La limitación de este caso es que realizar un bucle de lectura convencional desde este flujo de datos bloquearía el programa y no permitiría que el conmutador siga reenviando tráfico. Para poder responder de manera inmediata a estas notificaciones y que a su vez el conmutador pueda seguir realizando su labor, es

necesario que esta lectura se realice desde un hilo de ejecución independiente (para más detalles vea el anexo sobre concurrencia y sincronización).

Defina e inicie un nuevo *thread* que ejecute este comando. Deberá leer la salida del mismo para identificar los eventos de desconexión y extraer el nombre de la interfaz afectada. Por el momento imprima únicamente un mensaje por *stderr* que avise de la desconexión de dicha interfaz. Pruebe nuevamente a desconectar y reconectar interfaces y valide que estos eventos se detectan adecuadamente.

Una vez el programa está detectando adecuadamente estos eventos, deberá operar sobre la tabla de direcciones MAC y la lista de interfaces que utiliza para reenviar a broadcast y realizar inundación. Tenga en cuenta los posibles problemas de acceso concurrente a memoria compartida (ver anexo). Puede resolverlos utilizando las clases envolventes de sincronización del paquete `java.util.Collections`<sup>6</sup>.

Asegúrese de que dispone de un flujo unicast que esté siendo reenviado exclusivamente por una de las interfaces. Fuerce un evento de desconexión y valide que el ese tráfico comienza a reenviarse por inundación por el resto de interfaces.

Punto de control 3 (10%): Muestre el conmutador funcionando y explique las modificaciones realizadas en el programa.

---

<sup>6</sup> <https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html>

## 6. Encolado de paquetes

Hasta el momento se ha supuesto que el conmutador es capaz de reenviar todos los paquetes que recibe. No obstante, en la práctica es posible que el volumen de tráfico que debe ser enviado por una interfaz supere el ancho de banda disponible en el mismo o las capacidades de procesamiento del conmutador. Por ejemplo, se están usando 3 interfaces de PC B, por defecto todas ellas a 100Mbps. Si se recibiesen 100Mbps por dos de las interfaces, y todo el tráfico debe salir por la tercera, el equipo será físicamente incapaz de sacar todo este tráfico por la interfaz correspondiente.

Simplificando mucho, diremos que los conmutadores implementan una cola FIFO<sup>7</sup> para cada interfaz. Cuando el equipo recibe un paquete, decide por qué puerto (o puertos) debe ser reenviado y lo almacena en la cola de salida correspondiente. Si en el momento de almacenar un paquete el equipo se encuentra con que la cola de la interfaz está llena, el paquete se descarta y se producen pérdidas en la retransmisión del tráfico. Una vez los paquetes se han almacenado en las colas correspondientes, cada interfaz es capaz de actuar de forma independiente, cogiendo paquetes de su cola y enviándolos por la interfaz al ritmo al que sea capaz.

**Nota:** *En realidad la arquitectura de los conmutadores comerciales como el Cisco Catalyst 2950 que hay en el laboratorio es más compleja, ya que se da soporte a más funcionalidades y hay limitaciones de implementación y memoria debido a la tasa que estos deben dar. Por ejemplo, muchos conmutadores implementan varias colas independientes por interfaz para priorizar un tipo de tráfico u otro según configuración<sup>8</sup>. También pueden implementar colas de paquetes a la entrada de las interfaces, de forma que estos puedan esperar a capacidad de cómputo para asignarlos a la interfaz de salida correspondiente. En ocasiones la memoria disponible para estas colas es compartida.*

Para replicar este comportamiento va a ser necesario realizar varias modificaciones:

- El programa deberá definir una cola FIFO de tamaño limitado para cada interfaz. Para esta implementación se recomienda utilizar la clase `LinkedBlockingQueue`<sup>9</sup> de la librería estándar de Java. Se utilizarán los métodos `offer`<sup>10</sup> para añadir paquetes y `take`<sup>11</sup> para extraerlos.
- El programa principal (la función `main`) deberá mantener el bucle actual de lectura de paquetes desde `stdin`. Ahora, al recibir paquetes estos se deberán asignar y almacenar en la cola correspondiente (o colas), siempre que

---

<sup>7</sup> [https://es.wikipedia.org/wiki/First\\_in,\\_first\\_out](https://es.wikipedia.org/wiki/First_in,_first_out)

<sup>8</sup> [https://es.wikipedia.org/wiki/Calidad\\_de\\_servicio](https://es.wikipedia.org/wiki/Calidad_de_servicio)

<sup>9</sup> <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedBlockingQueue.html>

<sup>10</sup> [https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html#offer-](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html#offer-E-)

<sup>11</sup> <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html#take-->

exista espacio disponible en la misma. Una vez el paquete se almacene, el programa deberá volver a quedarse esperando el siguiente paquete.

- Debido al cambio anterior, hará falta que mientras el programa principal está bloqueado esperando para leer nuevos paquetes por *stdin*, cada interfaz sea capaz de actuar de forma independiente sobre los paquetes ya asignados a las colas y que estos se vayan enviando según sea posible. Para ello deberá iniciar un hilo independiente para cada interfaz.

Realice los cambios solicitados para añadir una cola de salida en cada interfaz, con tamaño máximo de 15 paquetes. Añada un `sleep`<sup>12</sup> en la función `sendFrame` para simular el tiempo que le supone a una interfaz realizar el envío de cada paquete. ¿Cuál es la interfaz del conmutador que más tráfico está cursando a su salida y cuál es esta tasa en paquetes por segundo? ¿Qué valor tendría que establecer en el `sleep` para que se llene la cola de esa interfaz y el conmutador empiece a perder paquetes?

Configure el `sleep` para asegurarse de que el conmutador sufre pérdidas. Imprima un mensaje por `stderr` cada vez que el conmutador descarte un paquete.

Punto de control 4 (10%): Muestre el conmutador funcionando y explique las modificaciones realizadas en el programa.

---

<sup>12</sup> <https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html#sleep-long->

## 7. Mejoras adicionales

Para quien tenga interés, se proponen las siguientes mejoras adicionales que pueden realizarse sobre el programa actual:

- Eliminar la necesidad de indicar las interfaces como argumentos al iniciar el programa. Puede tratar de conseguir que el conmutador descubra autónomamente las interfaces disponibles y estas se mantengan actualizadas gracias al hilo que ejecuta `ip monitor link`.
- Realizar la captura desde el propio conmutador. Si el conmutador se encarga internamente de ejecutar el proceso de `tshark` para realizar la captura, se puede realizar las siguientes mejoras:
  - Es posible responder ante la configuración de nuevas interfaces (`ifconfig up/down`) para adaptar la captura a las interfaces disponibles. Actualmente `tshark` no responde ante la configuración de un nuevo interfaz (`ifconfig up`), y se para si se desconfigura una en uso.
  - Deja de ser necesario que quien ejecute el programa sepa qué programa debe ejecutar y con qué configuración para generar el flujo de tráfico que espera el conmutador por `stdin`.
  - Al liberar `stdin`, puede utilizarlo para aceptar comandos de depuración. Por ejemplo, replicar el funcionamiento del Catalyst2950 e imprimir la tabla de direcciones si escribe `show mac address-table`.
- Incluir contadores en las interfaces. Los equipos comerciales definen contadores acumulados por interfaz que permiten ver la cantidad de paquetes recibidos, enviados y descartados desde el arranque del equipo. Estos contadores se podrán mostrar bajo petición del usuario mediante un comando leído desde `stdin`.
- Anteriormente, por simplicidad, se había pedido comprobar la caducidad de las entradas de la tabla de direcciones cuando se va a reenviar un paquete destinado a la dirección MAC correspondiente. Ahora que conoce cómo definir e iniciar un thread adicional, puede conseguir que el conmutador caduque las tablas de forma proactiva en vez de reactiva únicamente al recibir tráfico hacia un destino concreto.

## A. Concurrency y sincronización

La concurrencia<sup>13</sup> es la capacidad de un programa de paralelizar o alternar la realización de varias tareas. Es gracias a esto que su ordenador es capaz de ejecutar varios programas de forma simultánea, como una shell, un programa de java y una instancia de Wireshark. El procesador ejecuta el código de los programas de forma secuencial, pero es capaz de tener varios programas que requieren de su atención e intercalar sus ejecuciones, de forma que todos ellos se ven atendidos.

**Nota:** *Según la capacidad del procesador, es posible que disponga de varios núcleos o cores, y pueda ejecutar varias tareas de forma simultánea (cada una en un core). Esta ejecución simultánea da concurrencia, pero esta puede obtenerse sin necesidad de simultaneidad.*

A su vez, un mismo programa puede requerir de realizar varias tareas de larga duración y querer ejecutarlas de forma concurrente para no tener que terminar una para empezar con la siguiente. Una forma de conseguir esto es mediante la utilización de varios *Threads* o hilos de ejecución. Entiéndase un hilo de ejecución como una serie de instrucciones (su código) que se ejecutan de forma secuencial y representan una tarea a realizar. De esta forma, un programa es capaz de tener un hilo de ejecución realizando una tarea bloqueante o de larga duración, mientras la intercala con otras.

La librería estándar de Java expone la clase `Thread`<sup>14</sup>. Hay múltiples formas de iniciar *threads* en Java a partir de esta clase, cada una con sus ventajas e inconvenientes. Una de las más habituales es definir una clase que herede de `Thread` y sobrescriba el método `run`. De esta forma, cuando se inicia el thread, ejecutará el contenido de este método hasta su finalización.

En el siguiente ejemplo puede ver cómo definir una clase que hereda de la clase `Thread`, define un constructor para aceptar cierta parametrización y sobrescribe el método `run` con la lógica que se quiere ejecutar. En concreto, desde el hilo principal se instancian e inician 5 hilos que contarán hasta 100 e imprimirán estos valores junto con el nombre del hilo.

---

<sup>13</sup>

<https://www.tutorialesprogramacionya.com/computacionparalelaconcurrente/introduccion/index.html>

<https://javaknowledgebase.com/java-concurrency-and-multithreading-a-complete-guide-with-examples/>

<sup>14</sup> <https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html>

```

class MyThread extends Thread {
    String name;
    int count;

    public MyThread(String name, int count) {
        this.name = name;
        this.count = count;
    }
    public void run() {
        System.out.printf("Started thread %s\n", this.name);
        for (int i = 0; i < this.count; i++) {
            System.out.printf("%s: %d\n", this.name, i);
        }
        System.out.printf("Finished thread %s\n", this.name);
    }
}

public class Main {
    public static void main(String[] args) {
        for (int i = 0; i < 5; i++) {
            String tname = "Thread"+i;
            MyThread t = new MyThread(tname, 100);
            System.out.printf("Starting thread %s\n", tname);
            t.start();
        }
        System.out.println("Finished starting threads");
    }
}

```

Si realiza la ejecución varias veces verá que el resultado no es determinista, ya que la selección de qué *thread* ejecutar y durante cuánto tiempo depende del sistema operativo y no se puede controlar desde el programa en Java. Puede obtener un resultado similar al siguiente:

```

Starting thread Thread0
Starting thread Thread1
Started thread Thread0
Thread0: 0
Thread0: 1
Starting thread Thread2
Started thread Thread1
Thread1: 0
Thread1: 1
...
Thread1: 99
Finished thread Thread1
Thread0: 2

```

```
Thread0: 3
Starting thread Thread3
Started thread Thread2
Thread2: 0
Thread2: 1
...
```

Cuando se define un thread, de igual manera que es posible parametrizarlo con *Strings* o enteros, es posible compartir referencias a objetos mutables como por ejemplo *arrays*. En estos casos, todos los hilos compartirán acceso al mismo objeto y verán las modificaciones que otros realicen sobre el mismo. Por ejemplo:

```
import java.util.Arrays;

class MyThread extends Thread {
    int[] arr;
    boolean exit;

    public MyThread(int[] arr) {
        this.arr = arr;
        this.exit = false;
    }
    public void run() {
        while (!this.exit) {
            System.out.println(Arrays.toString(this.arr));
            try {
                Thread.sleep(1000);
            }
            catch (Exception e) {}
        }
    }
}

public class Main {
    public static void main(String[] args) throws Exception {
        int[] arr = new int[10];
        MyThread printer = new MyThread(arr);
        printer.start();

        for (int i = 0; i < 10; i++) {
            arr[i] = i;
            Thread.sleep(500);
        }

        printer.exit = true;
    }
}
```

En este ejemplo se levanta un thread que se mantendrá en ejecución hasta que se establezca la variable `exit` a `true`. Mientras se ejecute, imprimirá el contenido del array en intervalos de un segundo. El hilo principal asignará los valores de este array en intervalos de 500ms, y una vez termine establecerá la variable `exit` del *thread* a `true`. Puede esperar una salida similar a la siguiente:

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 1, 2, 3, 0, 0, 0, 0, 0, 0]
[0, 1, 2, 3, 4, 5, 0, 0, 0, 0]
[0, 1, 2, 3, 4, 5, 6, 7, 0, 0]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Este tipo de acceso a memoria compartida permite una gran flexibilidad a la hora de diseñar programas, pudiendo resolver problemas de gran complejidad de forma eficiente. Distintos *threads* pueden realizar tareas diferentes, pero dependen de un estado compartido y de comunicarse entre ellos para notificar cambios en el estado del sistema. No obstante, si bien el acceso concurrente a memoria compartida puede permitir resolver muchos problemas, también puede ser origen de complejidad y comportamientos inesperados difíciles de depurar, ya que el orden de ejecución de los distintos *threads* no es determinista. Véase el siguiente ejemplo:

```
class Counter {
    int value = 0;
    public void increment() {
        this.value = this.value + 1;
    }
}

class CounterThread extends Thread {
    Counter c;

    public CounterThread(Counter c) {
        this.c = c;
    }
    public void run() {
        for (int i = 0; i < 1_000_000; i++) {
            this.c.increment();
        }
    }
}
```

```

public class Main {
    public static void main(String[] args) throws Exception {
        Counter c = new Counter();

        Thread[] threads = new Thread[2];
        for (int i = 0; i < threads.length; i++) {
            threads[i] = new CounterThread(c);
            threads[i].start();
        }

        for (Thread t: threads) {
            t.join();
        }

        System.out.printf("Expected: %,d\n", 2_000_000);
        System.out.printf("Actual:    %,d\n", c.value);
    }
}

```

Se está definiendo una clase `Counter` que simplemente contiene un valor que se puede incrementar. Se define una clase `CounterThread`, que se construirá con una referencia a un `Counter` y que al ejecutarse lo incrementará 1 millón de veces. En el hilo principal, se crea este contador y dos *threads* que incrementarán el mismo contador (acceso concurrente a memoria compartida). Los *threads* se inician y el hilo principal espera a que ambos terminen (`Thread.join`<sup>15</sup>). Finalmente, se imprime el valor resultante después de todos los incrementos.

Puesto que se están ejecutando dos *threads* y cada uno realiza un millón de incrementos, se esperaría que el resultado final fuese 2,000,000, pero ejecutando este programa varias veces se puede ver que no es el caso y además el resultado final no es determinista. El problema que tiene este código es conocido como una condición de carrera o *race condition*<sup>16</sup>, y viene generado por el hecho de que el incremento del contador no es una operación atómica<sup>17</sup>. El acto de incrementar esa variable internamente se divide en: 1) leer el valor actual, 2) calcular el valor incrementado y 3) escribir el nuevo valor en la variable. Puesto que se están ejecutando 2 hilos de forma concurrente, puede suceder lo siguiente:

- [thread1] Lee el valor del contador. Obtiene el valor 5.
- [thread2] Lee el valor del contador. Obtiene el valor 5.
- [thread1] Calcula el incremento. Obtiene el valor 6.
- [thread1] Escribe el valor al contador. Se almacena el valor 6.
- [thread2] Calcula el incremento. Obtiene el valor 6.
- [thread2] Escribe el valor al contador. Se almacena el valor 6.

---

<sup>15</sup> <https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html#join-->

<sup>16</sup> [https://en.wikipedia.org/wiki/Race\\_condition#In\\_software](https://en.wikipedia.org/wiki/Race_condition#In_software)

<sup>17</sup> <https://www.geeksforgeeks.org/operating-systems/atomic-operations-in-os/>

De esta forma se están realizando 2 incrementos independientes, pero el resultado conjunto es equivalente a incrementar una única vez el contador debido a la falta de sincronización entre hilos. Incluso podría pasar que uno de los *threads* incremente varias veces el contador en lo que el otro solo realiza un incremento, y al escribir el valor resultante este se vea efectivamente reducido. Debido a esto es muy improbable que el resultado final del programa anterior alcance el valor esperado.

Para resolver este problema es necesario que ambos *threads* se sincronicen para que cuando uno ha leído un valor, el otro espere a que este termine de escribir el valor actualizado antes de proceder con su lectura para el siguiente incremento. Hay multitud de formas de realizar esta sincronización<sup>18</sup>, lo que habitualmente encontrará disponible en cualquier lenguaje de programación son cerrojos o *locks*<sup>19</sup>. En Java, se dispone además del *keyword* `synchronized`, que internamente hace uso de un lock pero le abstrae de esta complejidad al usuario. Con esto lo que se consigue es indicar una condición de exclusión mutua, de forma que mientras un hilo se encuentra ejecutando una sección de código, ningún otro hilo puede ejecutarla.

```
class CounterThread extends Thread {
    Counter c;

    public CounterThread(Counter c) {
        this.c = c;
    }
    public void run() {
        for (int i = 0; i < 1_000_000; i++) {
            synchronized (this.c) {
                this.c.increment();
            }
        }
    }
}
```

Aquí a `synchronized` se le indica el objeto en base al cual debe sincronizarse, y utiliza este como *lock*. Un programa puede tener varios bloques de código sincronizados en base al mismo objeto y Java se asegurará de que en ningún momento se ejecute más de uno de estos bloques de forma concurrente. A su vez, puede tener varios bloques sincronizados en base a diferentes objetos y estos no se interferirán entre sí. Lo habitual es utilizar como referencia el objeto que va a sufrir las modificaciones.

---

<sup>18</sup> <https://www.codebegun.com/learn/java/multithreading/synchronization>

<sup>19</sup> <https://www.baeldung.com/java-concurrent-locks>

Adicionalmente, es habitual que encuentre objetos de la librería estándar del lenguaje que soporten esta sincronización de forma transparente al usuario. Cuando un objeto (o función) está protegido ante este tipo de problemas, suele denominarse como *thread-safe*<sup>20</sup>. Por norma general, los objetos no dan soporte a este tipo de sincronización por defecto ya que supone un impacto en el rendimiento. Lo habitual es contar con una segunda implementación con la misma interfaz de uso, pero este soporte adicional. Por ejemplo, en Java tiene las siguientes clases:

- `java.util.Collections.synchronizedList`
- `java.util.Collections.synchronizedMap`
- `java.util.Collections.synchronizedSet`
- `java.util.concurrent.CopyOnWriteArrayList`
- `java.util.concurrent.ConcurrentHashMap`
- `java.util.concurrent.CopyOnWriteArraySet`
- `java.util.concurrent.PriorityBlockingQueue`

---

<sup>20</sup> [https://en.wikipedia.org/wiki/Thread\\_safety](https://en.wikipedia.org/wiki/Thread_safety)