Programación de Redes y Servicios
**Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación**
**Universidad de Zaragoza, curso 2019/2020**

# Práctica 4: **Programación de *socket*: sistema de mensajes**

## Objetivos y requisitos

### Objetivos

Tras la realización de esta práctica, el alumno deberá ser capaz de:

- Manejar clases básicas relacionadas con la interfaz *socket* de Java basada en TCP.
- Desarrollar un sistema de mensajes punto a punto basado en *socket* que facilite realizar un envío desacoplado entre procesos.
- Utilizar dicho sistema de mensajes tanto para enviar datos entre procesos distribuidos como para sincronizar procesos distribuidos.

### Requisitos

- Compilador y máquina virtual de Java (JVM) Java Platform Standard Edition 7[1]
- `Eclipse`, como entorno de desarrollo[2].

## 1. Contenidos

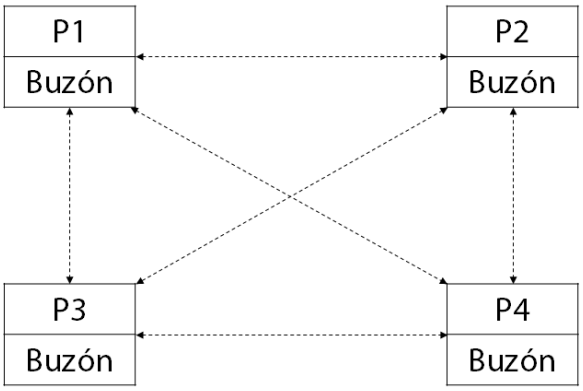
Los objetivos propuestos en esta práctica pretenden afianzar y complementar los contenidos teóricos vistos en clase. Por lo tanto, será necesario un estudio previo de los mismos, así como la utilización de los apuntes de clase (y cualquier material adicional que el alumno considere oportuno) como apoyo a la realización práctica. Se recomienda la consulta de la bibliografía recomendada [1], así como la consulta a la ayuda de Java proporcionada por Oracle.

[1] Disponible para descarga en `http://www.oracle.com/technetwork/java/javase/downloads/index.html`.
[2] Disponible para descarga en `http://www.eclipse.org/downloads/`

- Comunicación entre procesos.
- Interfaz *socket*.
- *Sockets* en Java:
  - Direcciones
  - *Sockets* TCP
  - Envío y recepción de datos: Interfaz *Serializable*
  - Paradigma de Paso de Mensajes: patrones de sincronización

## 2. Descripción del Problema

En esta práctica en primer lugar se va a implementar un sistema de mensajes punto a punto basado en *socket* TCP que permita realizar un envío desacoplado entre procesos. Es decir, el receptor no tiene por qué estar esperando la llegada de un mensaje cuando esta se produce. En su lugar, los mensajes se almacenan en un buzón hasta que el receptor decida consultarlo.




Una vez implementado, este sistema de mensajes constituye un middleware que nos va a permitir abstraer las particularidades de implementación de la programación de *socket*, para utilizarse como punto de partida para el desarrollo de sistemas distribuidos más complejos. En esta práctica se realizará también un ejemplo de aplicación en el que varios procesos distribuidos acceden a datos comunes en un **servidor de variables compartidas** que requieren un acceso sincronizado.

**IMPORTANTE**: desacoplar implica no mantener conexiones permanentes entre usuarios, sino establecerlas de manera transparente y exclusivamente para el envío de los mensajes.

# 3. Implementación del Sistema de Mensajes

## 3.1. Envío de objetos por red

Dado que controlamos la programación de los dos extremos de la comunicación, podemos hacer uso de los mecanismos específicos que proporciona Java para el envío de mensajes. Concretamente vamos a utilizar la interfaz ya conocida `Serializable`. Dado que el envío de mensajes se basará en TCP, será necesario hacer uso de las clases `ObjectInputStream` y `ObjectOutputStream`, tal como se detalla en los contenidos teóricos de la asignatura para el caso de dicho protocolo de transporte. En el sistema de mensajes de la práctica, el objeto a enviar y recibir será la clase `Envelope`. Una instancia de la clase `Envelope` encapsula un objeto serializable cualquiera (*payload*) así como información acerca del remitente y destinatario del mensaje. Este "sobre" es lo que los procesos se enviarán entre sí, para poder identificar fácilmente el origen de cada mensaje. El *payload* podría ser, por lo tanto, cualquier otra clase que contenga los datos que se quieren intercambiar (desde un simple entero –`int`, una cadena de caracteres –`String`, o una clase serializable más compleja —`Message`, como el caso de la práctica previa).

**Nota:** al crear una clase que implementa el interfaz Serializable, Eclipse sugiere incluir un campo llamado `serialVersionUID`. Indica que genere el valor por defecto, en esta práctica es suficiente. Para más información, consulta la documentación del interfaz Serializable.

```
public class Envelope implements Serializable {
    private static final long serialVersionUID = 1L;
    private int source;
    private int destination;
    private Serializable payload;

    public Envelope(int s, int d, Serializable p) {
        source = s;
        destination = d;
        payload = p;
    }

    public int getSource() { return source; }
    public int getDestination() { return destination; }
    public Serializable getPayload() { return payload; }
}
```

## 3.2. Buzón de mensajes

El buzón de un sistema de mensajes punto a punto es lo que permite desacoplar el envío de la recepción de mensajes. No es más que una cola de mensajes recibidos pero no leídos. Es decir, el sistema de mensajes debe recibir los mensajes según vayan llegando, y guardarlos en el buzón para cuando el receptor lo consulte. Esto se puede hacer con un **hilo que se encarga de escuchar conexiones entrantes**. Cada vez que llega un

mensaje nuevo, lo añade al final de la cola. Cuando el hilo principal examine el buzón, le devolverá el primer mensaje de la cola. La gestión del buzón de mensajes es equivalente al modelo Productor-Consumidor conocido, en el que la recepción de mensajes "produce" mensajes en cola, mientras que su lectura los "consume".

En un sistema distribuido habitual, los recursos son limitados, por lo que se debe asignar un tamaño máximo al buzón. Si no se quieren perder mensajes, deberá gestionarse correctamente el acceso al buzón.

## 3.3. Estructura de clases

Cada proceso que quiera utilizar el sistema de mensajes creará una instancia de la clase `MessageSystem`, en el paquete `ms`. A continuación, se muestra un resumen de dicha clase, que se adjunta con el enunciado de la práctica:

Los miembros de la clase son:

- `_user`: identificador del Usuario en el Sistema de mensajes (nº de línea del fichero *users.txt* descrito a continuación)
- `_mailbox`: instancia de la clase (`MailBox`) que implementa el buzón de mensajes
- `_sending`: instancia de la clase (`Sending`) que implementa las funciones de envío de mensajes

```
1  public class MessageSystem {
2
3  private int _user;
4      private ArrayList<InetSocketAddress> _peers;
5      private MailBox _mailbox;
6      private Sending _sending;
7
8  public MessageSystem(int source, String usersFile){...}
9
10 public void send(int dst, Serializable message){...}
11 public Envelope receive(){...}
12
13 public void stopMailbox(){...}
14 private InetSocketAddress loadPeerAddresses(String usersFile) {...}
15 }
```

El constructor (8) recibe dos parámetros. El primer parámetro es el identificador del proceso actual dentro del sistema de mensajes. El segundo es un fichero de red que contiene la dirección de los participantes del sistema, que debe seguir la siguiente estructura:

| **ID proceso** | **(nº línea)** | Contenido **users.txt** |
|---|---|---|
| 0 | 0 | $IP_1$:puerto-1 |
| 1 | 1 | $IP_2$:puerto-2 |
| 2 | 2 | ... |
| n | n | $IP_n$:puerto-n |

Es decir, la línea n contiene una pareja de valores, con el nombre (o la IP) de la maquina donde se está ejecutando el proceso n y el puerto en el que escucha mensajes entrantes (separados por :) . Así, cada proceso puede saber a dónde dirigirse al enviar un mensaje. Además, de la línea que corresponde al proceso en ejecución, el objeto `MessageSystem` puede extraer en qué puerto se tiene que poner a escuchar. Este fichero tiene que ser el mismo para todos los procesos. Si ese fichero no existe, se produce una excepción de clase `FileNotFoundException`.

En el momento de crear una instancia de la clase `MessageSystem`, en el constructor se ejecuta el método `loadPeerAddresses` (14), en el que se realiza la lectura de dicho fichero para identificar a todos los usuarios del sistema junto con sus direcciones IP y puertos, almacenando la información relacionada en la lista `_peers`, con elementos de la clase `PeerAddress`. Adicionalmente, el método devuelve la `InetSocketAddress` (dirección de *socket*, IP y puerto) relativa al propio usuario, necesaria para la creación del `MailBox` correpondiente. A través de la lista de `_peers` se puede identificar en cualquier momento la dirección de *socket* adecuada para cada usuario. Hay que tener en cuenta que, a la hora de enviar mensajes mediante `Envelope`, identificamos a los usuarios mediante su "ID", mientras que el envío real de la información mediante *socket* requiere la utilización de la dirección `InetSocketAddress`.

```
public class PeerAddress {
    public int user;
    public String host;
    private InetAddress address;
    public int port;

    public PeerAddress(int u,String host, int p) {...}
    public String getHost(){...}
    public int getUser(){...}
    public InetSocketAddress getSockAddress(){...}
    public InetAddress getAddress(){..}
    public int getPort(){return port;}
}
```

Los siguientes dos métodos se encargan del envío y la recepción: El método `send` (10) envía un objeto serializable (un `Envelope`) a cualquier destinatario del sistema. Dicho método hace uso de la clase `Sending`. El método `receive` (11) se encarga de extraer el siguiente mensaje del buzón. Finalmente, el método `stopMailbox` (13) detiene la ejecución del hilo que gestiona el buzón. El buzón se implementa mediante la clase `MailBox`. Como se ha comentado en el apartado 3.2, la clase `MailBox` que implementa este buzón es un hilo independiente al hilo principal. Dentro de dicho buzón, hay una cola de mensajes. La interacción con dicha cola involucra a diversos hilos de ejecución: aquel que

pone mensajes en cola (buzón) y aquel que lee mensajes de la cola (principal). Por lo tanto, es imprescindible una gestión adecuada de la misma (recordar mecanismos de sincronización).

Cada usuario dentro del sistema distribuido crea una instancia de la clase `MessageSystem` para poder gestionar el envío y recepción de mensajes, como se indica en el siguiente ejemplo:

```
int src = 2;
String usersFile = users.txt
MessageSystem ms = new MessageSystem(src, usersFile);
```

donde `src` se corresponde con el ID del usuario local (fila #2 del fichero *users.txt*), y `usersFile` es el fichero de texto (*users.txt*) que contiene la identificación ($IP$:puerto) de cada usuario del sistema distribuido, incluido el local (usuario #2).

Al crear `ms`, se crean el `MailBox` y el `Sending` correspondiente, y pueden utilizarse los métodos necesarios para la recepción (`ms.receive`) o el envío (`ms.send`) de "sobres" (`Envelope`) que contengan el mensaje concreto del servicio que utiliza dicho `MessageSystem`. Dicho mensaje podrá ser cualquier objeto que implemente `Serializable` (recordemos la clase `Message` de la práctica anterior), o cualquier tipo básico (que ya es `Serializable`), como `String`, `int`, etc. En esta práctica, la validación básica del `MessageSystem` se realizará mediante la ejecución de unos procesos que hacen uso del mismo mediante el envío/recepción de texto (ver Ejercicio 1). Sin embargo, como se indica en el siguiente apartado, se plantea también la creación de un sistema distribuido que haga uso del `MessageSystem` tanto para la comunicación de datos como la sincronización, por lo que, en ese caso, puede ser necesario utilizar mensajes más específicos (análogo a la clase `Mesasge` de la práctica 3).

## 4. Aplicación del Sistema de Mensajes

Un sistema distribuido es inherentemente concurrente, pero el paradigma o modelo de programación en los sistemas distribuidos no puede hacer uso de memoria compartida. Por este motivo, existe el paradigma de paso de mensajes, mediante el cual los procesos distribuidos se sincronizan y se comunican mediante el intercambio de mensajes. Así, podemos utilizar el sistema de mensajes de esta práctica de dos formas distintas: (i) para enviar datos entre procesos distribuidos o (ii) para sincronizar procesos distribuidos. Por analogía, con la memoria compartida sucede algo parecido. Utilizamos las variables compartidas para comunicar datos entre procesos, pero también utilizamos la memoria (v.gr. los semáforos) para sincronizar procesos.

Como ejemplo de aplicación de lo descrito, en esta práctica se plantea modelar la comunicación entre varios procesos distribuidos que acceden a datos comunes en un **servidor de variables compartidas** (i.e. sección crítica) y que, además, utilizan un **servidor de semáforo** a través del cual sincronizan el acceso. La figura 1 muestra este esquema.

Así, se plantea la programación de cada una de las clases descritas en la figura:

- ***Procesos cliente distribuidos***, clase `Client`
  Cada proceso distribuido o cliente desea acceder a la memoria compartida en el servidor para leer un dato (`getValue`), modificarlo y devolverlo (`setValue`). Para acceder de manera sincronizada con el resto de clientes, debe interactuar con el servidor de semáforo (`SempahoreServer`) con la señalización *wait/signal* mediante paso de mensajes, de modo que solo cuando "obtenga permiso" podrá acceder al valor.

- ***Servidor de variables compartidas***, clase `Server`
  El servidor que contiene los datos puede tener en memoria diversas variables. En nuestro caso práctico nos limitamos a **una única variable**. De manera continuada está pendiente de la solicitud de la misma por parte de los diversos usuarios del sistema distribuido (`Client`). Cuando un usuario le solicita el valor actual, se lo envía y queda a la espera de que el usuario le devuelva el valor modificado para actualizar la variable.

- ***Servidor de semáforo***, clase `SemaphoreServer`
  El servidor de semáforo se encarga de recibir la señalización correspondiente (*wait, signal*) y responder de acuerdo a la misma a los usuarios, dando así permiso o manteniendo a la espera a cada uno de los procesos.

- ***Protocolo de comunicación***, clase `Message`.
  Para comunicar los procesos entre sí mediante paso de mensajes (haciendo uso del `MessageSystem`), es necesario definir los mensajes (*payload* del `Envelope`). Se deberán distinguir los mensajes de señalización (*wait/signal*) de la comunicación de datos con el servidor (*get, value, set/newvalue*). Como simplificación se proprociona una clase `Message` ya completa, que incluye datos tipo `String` e `int`, para enviar tanto texto como datos enteros:

```
public class Message implements Serializable{
    private static final long serialVersionUID = 2L;

    private int _v; // value (valor numérico)
    private String _t; // text (texto significativo)
    Message(int c, String t){
        _c = c;
        _t = t;
    }
    public int getInt() {return _c;}
    public String getString() {return _t;}
    public void setValues(int c, String t){_c=c;_t=t;}
}
```

## 4.1. Servidor de semáforo: Patrón Mutex

El patrón mútex garantiza que un conjunto de procesos acceden a un recurso compartido en exclusión mutua. En sistemas distribuidos existen varias formas de implementarlo.

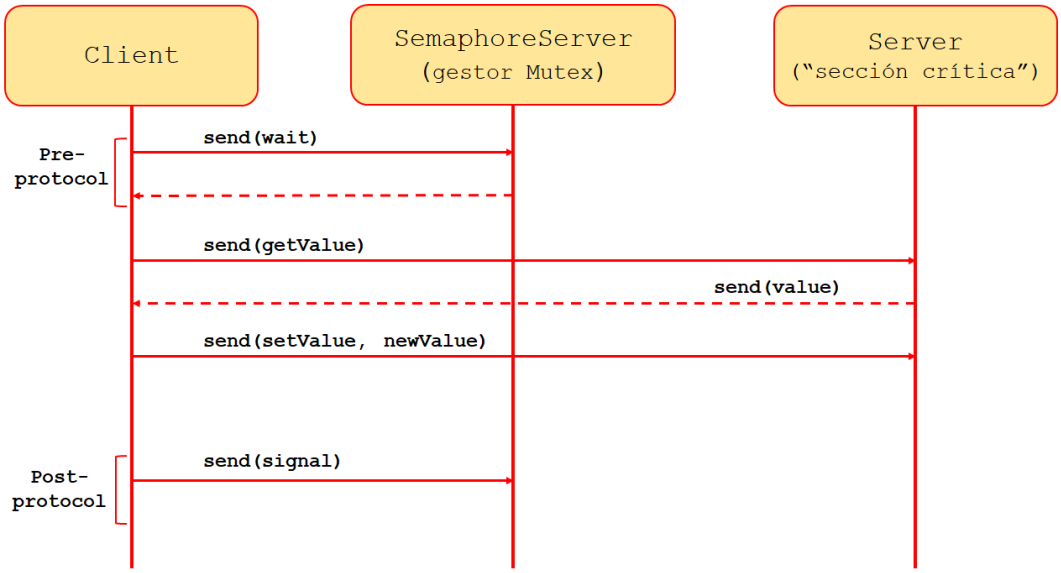



Figura 1: Acceso distribuido a sección crítica y sincronización. Paso de mensajes.

Una de ellas podría ser utilizando un proceso $P$ que actúe como un "servidor de semáforo" y que almacene las variables del semáforo: el valor del semáforo y la cola FIFO. El resto de procesos $Q_i$ cada vez que quieren acceder a la sección crítica le envían un mensaje "wait" y esperan a que se les conteste, esto es, esperan a tener acceso a la sección crítica. Una vez que un proceso $Q_i$ ha terminado de acceder a la sección crítica, le envía un mensaje a $P$ con un "signal". El proceso $P$, por su parte, tiene un comportamiento que se corresponde con la semántica de las operaciones `wait` y `signal` de un semáforo. Si recibe una petición `wait` y el semáforo vale 1, el proceso $P$ responde inmediatamente a la petición con un ACK y decrementa el valor del semáforo en una unidad. Si se recibe un `wait` y el valor del semáforo es 0, entonces $P$ añade al proceso en una cola FIFO y no le contesta. Entonces, el proceso invocante $Q_i$ se queda bloqueado en su instrucción `q2`. Finalmente, si la operación es `signal` y si no había procesos bloqueados, $P$ incrementa el valor del semáforo en una unidad; pero si había procesos bloqueados, se toma el primer proceso $Q_i$ de la cola y se le envía un ACK: dicho proceso entonces se desbloqueará de `q2` y podrá acceder a la sección crítica.

La implementación de la clase `SemaphoreServer` se corresponde con la programación en `Java` del pseudo-código mostrado en la figura 2.

## 4.2. Acceso distribuido en exclusión mutua. Ejecución del código

Como sistema distribuido, cada proceso (incluidos los 2 servidores) puede ejecutarse en una máquina distinta, y por tanto, dispone de una dirección IP y puerto diferenciado. Mediante la utilización del `MessageSystem` descrito previamente esto supone que cada uno de ellos es uno de los usuarios del sistema, identificado en el fichero *users.txt* (todos

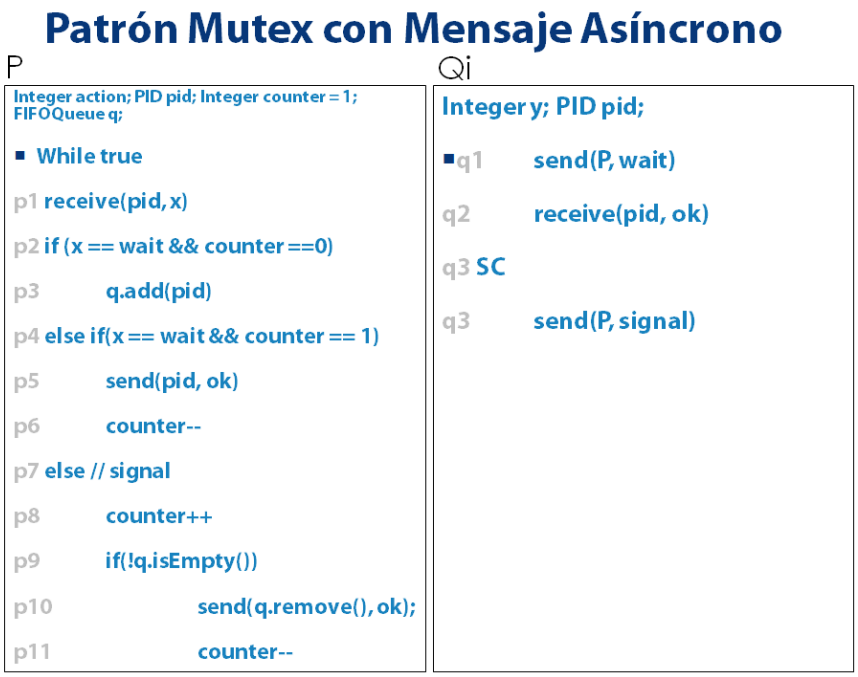



Figura 2: Patrón Mutex con Mensaje Asíncrono (apuntes de la asignatura).

los usuarios deben tener su copia del fichero) con su dirección IP y puerto (la dirección IP, si se ejecuta en modo local, puede ser "localhost", común a todos los procesos).

Para validar el funcionamiento del sistema distribuido se propone la ejecución local[3] de todos los procesos ("usuarios") en ventanas de consola independientes (`cmd`), distinguiendo a cada uno por su identificador y, en consecuencia, dirección de *socket* (IP y puerto) en el fichero *users.txt*. Como simplificación, podemos asumir el criterio ordenado de identificar el **servidor de semáforo/mútex** (`SemaphoreServer`) como el **usuario 0**, el **servidor** (`Server`) como el **usuario 1** y, a partir de ahí, correlativamente cada posible cliente (`Client`). El identificador de usuario, en el caso de cada cliente, puede introducirse como parámetro de entrada.

1. El **servidor de semáforo** se ejecuta y queda a la espera de recibir la señalización y responder en consecuencia. No se requiere ninguna inicialización ni parámetro adicional (su identificador se asume igual a 0, y el **contador inicializado a 1, como mútex** que implementa):

   ```
   $java SemaphoreServer
   ```

   Para verificar el funcionamiento, el servidor de semáforo puede mostrar por pan-

[3] El código debe funcionar igualmente si se ejecuta, ya compilado, en máquinas independientes

talla los diversos eventos (envío y recpción de señalización, indicando el cliente involucrado).

2. El **servidor** mantiene un dato compartido que puede ser modificado por cada cliente. Se propone la inicialización de dicho valor en el servidor, como parámetro de entrada (lo que permitirá verificar cómo dicho valor se devuelve a los clientes y cómo se modifica):

   ```
   $java Server <value>
   ```

   - `value`: valor inicial del dato que mantiene el servidor

   Para verificar el funcionamiento correcto del sistema distribuido, el servidor debe ir mostrando por pantalla las modificaciones del valor guardado.

3. Cada **cliente** trata de acceder al valor compartido para modificarlo (por ejemplo, incrementando una cantidad), por lo que deberá comunicarse tanto con el servidor, como con el servidor de semáforo (para regular el acceso). Para poder modelar posibles situaciones de espera de los clientes se plantea que cada cliente puede "retener" el dato compartido un tiempo variable, suspendiendo su ejecución con `sleep()`. El tiempo de suspensión puede introducirse como parámetro de entrada, así como el valor del dato a modificar:

   ```
   $java Client <id> <time> <value>
   ```

   - `id`: identificador de usuario ($\geqslant 2$)
   - `time`: tiempo que el cliente mantiene el dato (milisegundos)
   - `value`: valor a incrementar sobre el valor en servidor

   Para verificar el funcionamiento, el cliente puede mostrar por pantalla el valor devuelto por el servidor y el modificado a enviar (el envío, tras hacer `sleep`).

## 5. Ejercicios

Como resultado de esta práctica deberán generarse dos paquetes, distinguiendo el código relativo al `MessageSystem` (apartado 3.3) y el código correspondiente al sistema distribuido (apartado 4):

- `package p4.ms`: paquete con todo el código java relativo al `MessageSystem`. De este modo, el sistema de paso de mensajes es portable para su utilización en cualquier código que haga uso del mismo (mediante `import`). En esta práctica, por ejemplo, será necesaria su utilización dentro de los programas del siguiente paquete, relativos al acceso distribuido a la sección crítica.

- `package p4.sc`: paquete con todo el código java relativo al sistema distribuido de acceso a la sección crítica compuesto por el cliente, servidor y servidor de semáforo (`Client`, `Server`, `SemaphoreServer` y `Message`). Para el correcto funcionamiento del sistema de mensajes es necesario importar el paquete adecuado en el código java (por ejemplo, dentro de *SemaphoreServer.java* debe incluirse `import ms`).

En cualquier caso, la utilización de los paquetes deberá adaptarse a la utilización final del `package alumnos.p4.apellido` (igual que en la práctica 3).

Para la **ejecución del código en *modo consola*** deberemos situarnos en la carpeta `bin`, donde se encontrarán las carpetas relativas a los `packages` y los correspondientes `.class`:

- `bin\p4\sc\`: contiene el código relativo a los procesos que deben interaccionar mediante paso de mensajes:

  `Client.class`, `Server.class`, `SemaphoreServer.class`, `Message.class`.

- `bin\p4\ms\`: continene el código del sistema de mensajes:

  `MessageSystem.class`, `MailBox.class`, `Envelope.class`,`PCQueue.class`, `Sending.class`, `PeerAddress.class`.

De acuerdo a esta distribución de paquetes, la ejecución del `SemaphoreServer` sería:

```
$java p4.sc.SemaphoreServer
```

## Ejercicio 1.

El objetivo de este ejercicio es implementar el sistema de mensajes, para lo que será necesario completar las plantillas y utilizar las clases ya completas:

`package p4.ms`

- *MessageSystem.java*: **código completo** que no se debe modificar.
- *Mailbox.java*: **plantilla a completar** de la clase `MailBox`. Incluye como miembro la cola del buzón (*PCQueue.java*, descrito a continuación).
- *Sending.java*: **plantilla a completar** para el código de la clase `Sending`.
- *PCQueue.java*: clase **ya completa** que modela el buzón según el patrón Productor-Consumidor, sincronización incluida.
- *Envelope.java*: **código completo** de la clase `Envelope`, objeto Serializable enviado o recibido.

`package p4.test` (Código adicional **para verificación**) * NO ENTREGAR *

- *C.java*: Proceso `C` que utiliza el `MessageSystem` para recibir datos de `A` y B.
- *A.java*: Proceso `A` que utiliza el `MessageSystem` para enviar datos al proceso `C`.
- *B.java*: Proceso `B` que utiliza el `MessageSystem` para enviar datos al proceso `C`.

Para verificar de manera simplificada el correcto funcionamiento de dicho sistema de mensajes se pueden utilizar los procesos proporcionados `A`, `B` y `C`, que hacen uso del `MessageSystem`:

```
$java p4.test.C <idC>
```
(Consola 1)

```
$java p4.test.B <idB>
```
(Consola 2)

```
$java p4.test.A <idA>
```
(Consola 3)

**Puntuación: 6 puntos**

*Completa los códigos fuente de las clases `MailBox` y `Sending` y entrega el paquete completo `p4.ms`. Todos los .java deberán pertenecer a un mismo paquete (`package alumnos.p4.apellido.ms`) y deben guardarse en la ruta de carpetas* `/alumnos/p4/apellido/ms/`. *Recuerda: en este directorio guarda los códigos fuente (ficheros con extensión* `.java`*), no los ficheros objeto (extensión* `.class`*).*

***Nota:** El "apellido" se corresponderá con 1 alumno de la pareja y se escribirá en minúscula, sin acentos, ni ñ. Si hay apellidos comunes, se indicarán los dos apellidos de la pareja. Como ejemplo, los profesores utilizaríamos, en el caso del ms:*
*package alumnos.p4.canales.ms – archivos en carpetas: alumnos/p4/canales/ms/*.java*
*package alumnos.p4.gallego.ms – archivos en carpetas: alumnos/p4/gallego/ms/*.java*

## Ejercicio 2.

El objetivo de este ejercicio es implementar el acceso distribuido de varios clientes a una memoria compartida (servidor). La sincronización entre procesos hará uso de un servidor de semáforo. El sistema de mensajes servirá para el envío de información: **datos** (dato leído, modificado y devuelto) y **señalización** (signal/wait). Para realizar la programación se hará uso de las plantillas proporcionadas o se crearán las clases solicitadas:

`package p4.ms` Ejercicio 1 completo

- *MessageSystem.java*: **código completo**.
- *Mailbox.java*: **plantilla a completar** de la clase `MailBox`.
- *Sending.java*: **plantilla a completar** para el código de la clase `Sending`.
- *PCQueue.java*: clase **ya completa**.
- *Envelope.java*: **código completo** de la clase `Envelope`.

`package p4.sc` **Implementar completamente** (incluir `import p4.ms`):

- *Client.java*: Clase `Client` cuyas instancias se corresponderán a los distintos procesos (distribuidos) accediendo a la sección crítica.
- *SemaphoreServer.java*: Clase `SemaphoreServer` que implementa el "servidor de semáforos" encargado de arbitrar el acceso a la sección crítica.
- *Server.java*: Clase `Server` que implementa la memoria compartida por los procesos, a los que cada usuario accede para leer un dato, modificarlo y devolverlo.
- *Message.java*: Clase `Message` **ya completa** que implementa el mensaje a utilizar en el protocolo de comunicación (señalización y datos).

Los `package/import` deberán modificarse de acuerdo a la inclusión del paquete `alumnos.p4.apellido` como se indica a continuación.

**Puntuación: 10 puntos**

*Completa los códigos fuente de las clases `Client`, `Server`, `SempahoreServer` y `Message`. Todos los .java deberán pertenecer a un mismo paquete (`package alumnos.p4.apellido` y deben guardarse en la ruta de carpetas `/alumnos/p4/apellido/`. Recuerda: en este directorio guarda los códigos fuente (ficheros con extensión `.java`), no los ficheros objeto (extensión `.class`).*

***Nota:** El "apellido" se corresponderá con 1 alumno de la pareja y se escribirá en minúscula, sin acentos, ni ñ. Si hay apellidos comunes, se indicarán los dos apellidos de la pareja. Como ejemplo, los profesores utilizaríamos, en el caso del sc:*
*package alumnos.p4.canales – archivos en carpetas: alumnos/p4/canales/sc/*.java*
*package alumnos.p4.gallego – archivos en carpetas: alumnos/p4/gallego/sc/*.java*

# 6. Evaluación

La realización de las prácticas y su entrega es por parejas. Las normas de la evaluación y entrega son las siguientes:

- Todos los programas entregados deben compilar correctamente, se valorará de forma muy negativa que no compile algún programa.
- Todos los programas entregados deben funcionar correctamente como se especifica en el problema.
- Todos los programas tienen que seguir el manual de estilo de Java propuesto por Oracle, disponible en el anillo digital docente (un 20 % de la nota estará en función de este requisito). Además de lo especificado en el manual de estilo, cada fichero fuente deberá comenzar con la siguiente cabecera:

```
/*
 * AUTOR: nombre y apellidos de los 2 autores de la práctica
 * NIA: número de identificación de los 2 alumnos
 * FICHERO: nombre del fichero
 * TIEMPO: tiempo (en horas) empleado en la implementación
 * DESCRIPCIÓN: breve descripción del contenido del fichero
 */
```

**Resumen de la puntuación de los ejercicios solicitados en esta práctica**

| Ejercicio | Puntuación |
|---|---|
| *Ejercicio 1* | 6 puntos. |
| *Ejercicio 2* | 10 puntos. |

# 7. Instrucciones de entrega

- Se entregará un único fichero comprimido `.zip`, que contendrá los códigos fuentes de todos los ejercicios solicitados (ficheros con extensión `.java`, **NO los ficheros con extensión .class**). Solamente debe entregarse un ejercicio: **Ejercicio 1 o Ejercicio 2**, teniendo en cuenta que para resolver el Ejercicio 2 se requiere la implementación completa del Ejercicio 1.
- Los códigos fuentes estarán en sus respectivas carpetas, indicadas en el enunciado de cada ejercicio.
- La entrega se realizará a través Moodle (`http://moodle2.unizar.es`), se utilizará para ello una tarea correspondiente a la entrega de la práctica 4 habilitada a tal efecto.
- **Fecha de entrega: no más tarde del día anterior al comienzo de la sesión de prácticas 10**:

- Grupo `prs2` (921): 10 de mayo.
- Grupos `prs1`, `prs2` y `prs3` (931): 11 de mayo.
- Grupo `prs1 y prs6` (921): 12 de mayo.
- Grupos `prs4` y `prs5` (921): 14 de mayo.

**No se permite la entrega tardía.**

## Referencias

[1] Donahoo Michael J. Calvert Kenneth L. *TCP/IP Sockets in Java Bundle: TCP/IP Sockets in Java, Second Edition: Practical Guide for Programmers.* Morgan Kaufmann, USA, 2nd edition, 2008.