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

Práctica 3: **Programación de *socket*: Cliente-Servidor TCP/UDP**

Objetivos y requisitos

Objetivos

- Entender las particularidades de la programación de servicios cliente-servidor basados en TCP/UDP.
- Diseñar e implementar un protocolo de comunicación basado en *Serializable*.
- Manejar clases básicas relacionadas con la interfaz *socket* de Java basada en TCP/UDP.

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
  - *Sockets* UDP
  - Envío y recepción de datos: Interfaz *Serializable*
  - Servidores concurrentes

2. Descripción del Problema

Es muy habitual en la práctica que los sistemas distribuidos utilicen servidores de terceros para obtener cierta funcionalidad como, por ejemplo, capacidad de cálculo para procesos que los usuarios finales no pueden realizar en sus máquinas. En esta práctica se propone la realización de un **servicio de cálculo remoto de números primos** basado en una interacción **cliente-servidor**. El protocolo de transporte de la aplicación puede ser **TCP o UDP** (se plantea la resolución de ambos problemas).

Así, se propone la programación tanto del cliente como del servidor correspondientes al servicio de cálculo de primos, atendiendo a la funcionalidad descrita a continuación:

2.1. Características del servidor

El servidor será una aplicación de consola que se dejará en ejecución continuamente (el programa se parará directamente mediante Ctrl+C). Como parámetro de entrada se especificará únicamente el puerto de escucha del servidor:

```
$java PrimeServer <puerto>
```
[3]

- `puerto`: puerto de escucha del servidor

El servidor recibe peticiones de clientes que solicitan el cálculo de números primos en un intervalo dado. **El servidor funciona, inicialmente, de manera iterativa**, de modo que solo puede cursar peticiones de un único cliente, cualquier petición simultánea queda a la espera para ser servida.

El algoritmo de cálculo de primos se proporciona como un método estático dentro de la clase abstracta Primes (*Primes.java*), pudiendo utilizarse directamente en cualquier parte del código:

```
public abstract class Primes {
    public static int[] GetPrimes(int ini, int fin){..}
}
int[] primes = Primes.GetPrimes(ini, fin);
```

[3] En el apartado 3 se detalla cómo ejecutar en modo consola

2.2. Características del cliente

El cliente será una aplicación de consola que se conecta a un servidor remoto de cálculo de primos para solicitar el cálculo de la lista de primos en un intervalo dado. Como parámetros de entrada se especificarán la dirección IP o nombre de *host* así como el puerto del servidor remoto y el intervalo en el que calcular:

```
$java PrimeClient <ip/host> <puerto> <ini> <fin>
```

- `ip/host`: dirección IP o nombre de host del servidor remoto.
- `puerto`: puerto de escucha del servidor remoto.
- `ini, fin`: intervalo [ini,fin] en el que calcular la lista de números primos.

El cliente, una vez conectado, queda a la espera de recibir el resultado que, una vez devuelto, se visualizará por pantalla (lista completa de primos).

2.3. Protocolo de comunicación empleado a nivel de aplicación

El protocolo de comunicación empleado a nivel de aplicación consiste en el siguiente intercambio de petición-respuesta:

– Mensaje ***REQUEST***: solicitud del cliente en el que se especifica el intervalo para el que calcular el conjunto de números primos.

– Mensaje ***REPLY***: respuesta del servidor indicando el número total de números primos encontrados y la correspondiente lista.

A continuación, se muestra el formato genérico y los campos relevantes de los mensajes:

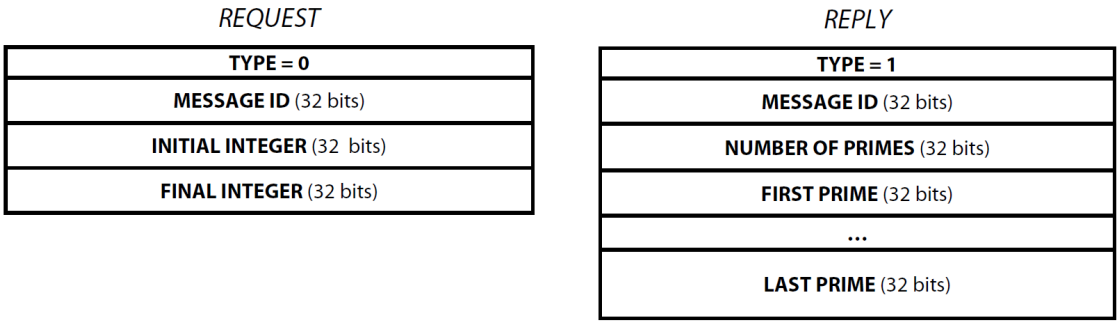
– **TYPE**: Campo indicativo del tipo de mensaje (*REQUEST* y *REPLY*)

– **MESSAGE ID**: Identificador. Cuando un cliente genera una petición, elige un identificador aleatorio. El servidor incluye en la respuesta el ID de la petición que ha servido.

– **INITIAL INTEGER, FINAL INTEGER**: extremos inicial y final del intervalo en el que calcular el número de primos.

– **NUMBER OF PRIMES**: número de primos encontrado en el intervalo. **Puede valer 0** si no se ha encontrado ninguno. En ese caso, se omiten los siguientes campos.

– **FIRST PRIME, . . . , LAST PRIME**: lista de números primos encontrados (tantos elementos en el mensaje como indica el campo NUMBER OF PRIMES)



2.4. Protocolo de nivel de transporte TCP/UDP: serialización

El protocolo de aplicación descrito puede funcionar sobre TCP o UDP como nivel de transporte. El comportamiento tanto del cliente como del servidor en cada caso difiere debido a la distinta naturaleza de ambos protocolos: i) TCP, basado en flujos de bytes y orientado a conexión y ii) UDP, basado en mensajes y no orientado a conexión. En esta práctica se va a plantear la implementación de ambos casos (ver detalle en apartado Ejercicios). En cualquier caso, el transporte de los datos de aplicación requiere de la serialización/de-serialización de los mismos (codificación/entramado y desentramado/-decodificación).

En esta práctica, dado que como programadores tenemos acceso a la implementación completa tanto del cliente como del servidor en Java, se va a utilizar la **serialización de objetos de java**. A la hora de implementar el servicio, se propone la programación de la clase `Message` (*Message.java*, de la que se proporciona la plantilla), en la que se incluirán los miembros y métodos necesarios para representar los tipos de mensajes indicados (REQUEST, REPLY) y acceder correctamente a todos sus campos. La clase `Message` implementará la interfaz `Serializable`: `http://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html`

```
public class Messsage implements Serializable {
private static final long serialVersionUID = 2L; // No modificar
}
```

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`.

Para que un objeto sea serializable, tiene que cumplir dos condiciones: implementar dicha interfaz `Serializable` y que todos sus campos sean a su vez objetos serializables. Todos los tipos básicos y en general los tipos de datos estándar de Java son serializables, por lo que construir clases serializables es sencillo. Una vez definida una clase de objeto serializable, para enviar una instancia por red hay que hacer uso de las clases `ObjectInputStream` y `ObjectOutputStream`, tal como se detalla en los contenidos teóricos

de la asignatura mediante los métodos `writeObject` y `readObject`.

Para reutilizar el código de envío y recepción de datos mediante `Serializable` en las clases involucradas (cliente y servidor) se plantea programar la clase abstracta `TransferMessage` (*TransferMessages.java*, de la que se proporciona la plantilla) que implementará los métodos estáticos `send` y `receive`, que podrán ser utilizados en el código sin crear ningún objeto de dicha clase.

```
public abstract class TransferMessages {
    public static Message receive(Socket sock)
        throws IOException {..}
    public static void send(Message msg, Socket sock)
        throws IOException {..}
}
```

Para utilizar los métodos `send` y `receive` no es necesario crear ningún objeto tipo `TransferMessasges`, sino que basta su ejecución tal y como se indica en el ejemplo:

```
Message m = TransferMessages.receive(sock);
// Recibe en el socket "sock" un objeto de tipo Message,
// que devuelve en "m"

TransferMessages.send(m,sock);
// Envía en el socket "sock" el objeto "m" de tipo Mesasge
```

2.4.1. Particularidades de UDP:

Direccionamiento explícito en los *socket*

Cuando se utilizan *sockets* TCP, el envío y recepción de datos se realiza a través de *socket* orientados a conexión que, por lo tanto, ya "conocen" la dirección (IP y puerto) a la que enviar o de la que se reciben los datos. En el caso de UDP, la dirección es un campo del `DatagramPacket` en el que indicar dónde enviar (`send`) o donde guardar de dónde se recibe (`receive`). Para reutilizar al máximo el código de la clase `TransferMessage`, se incluye un campo cuya utilización solo será necesaria en UDP: **el campo adicional `_address` en la clase `Message`**:

```
transient InetSocketAddress _address; // LOCAL: no forma parte del mensaje
```

Este campo **no forma parte del mensaje** enviado sobre UDP (protocolo). Al ser una variable `transient` no se serializa. Su utilización es local y permite identificar la dirección *socket* (IP y puerto):

- En recepción de un mensaje, permite identificar la **fuente que ha generado el mensaje**.
- En envío de un mensaje, permite identificar el **destino al que enviar el mensaje**.

De este modo, la utilización de la clase `TransferMessages` será análoga al caso de utilizar TCP:

```
public abstract class TransferMessages {
    public static Message receive(DatagramSocket sock)
        throws IOException {..}
    public static void send(Message msg, DatagramSocket sock)
        throws IOException {..}
}

Message m = TransferMessages.receive(sock); // recepción

TransferMessages.send(m,sock); // envío
```

Falta de paquetización

Cuando el transporte de datos se realiza mediante UDP, no hay paquetización del mensaje de aplicación como en TCP. Mientras TCP está basado en flujos de bytes (*streams*), UDP se basa en el envío/recepción de mensajes completos. Es decir, UDP no "parte" el mensaje de aplicación en segmentos (como TCP, que envía segmentos de tamaño MSS), sino que le envía al nivel IP el mensaje completo. Así, mientras la longitud de los datos sobre TCP no es un problema, dado que TCP va enviando todos los bytes poco a poco en segmentos, en UDP el mensaje tiene un tamaño máximo (limitado por los 65535 bytes de tamaño máximo de datagrama IP).

Por lo tanto, **como simplificación, se limita el número máximo de primos resultantes del cálculo** para evitar sobrepasar el límite, tal y como se indica en las plantillas (`public static final int MAXPRIMES = 8192;`). Para permitir resultados mayores debería diseñarse un mecanismo de paquetización o partición en bloques que, en esta práctica, no será necesario. **Si el cliente solicita un cálculo mayor, limitaremos la respuesta al número de primos que caben en el mensaje** (se puede sacar por pantalla un mensaje en el servidor para identificar que se ha dado este caso).

Falta de fiabilidad

El transporte mediante UDP no proporciona fiabilidad, al carecer el protocolo de mecanismos de control de errores (no hay números de secuencia ni retransmisiones). De este modo, cuando el cliente envía una petición sobre UDP, no hay garantía de que esta se entregue al destino, e igualmente la respuesta del servidor puede no llegar. Por otra parte, la naturaleza bloqueante del método `receive()` puede provocar que el proceso se quede infinitamente bloqueado ante la falta de datos recibidos.

No es necesario realizar la implementación de la gestión de errores en UDP en esta práctica. Sin embargo, en una implementación real, como solución a la falta de fiabilidad, limitando a su vez los bloqueos, se debería incluir al menos un mecanismo simplificado de desbloqueo y gestión de errores mediante retransmisión por *timeout*. Aprovechando la gestión de excepciones de java y el tratamiento controlado de la excepción específica `SocketTimeoutException`, se puede diseñar dicho mecanismo:

- Un método bloqueante de un *socket* puede serlo por un tiempo limitado especificando dicho funcionamiento (método `setSoTimeout`):

```
void setSoTimeout(int timeout)
// Enable/disable SO_TIMEOUT with the specified timeout,
// in milliseconds.
```

- Cuando expira un *timeout*, el hilo de ejecución recibe una interrupción en forma de excepción `SocketTimoutException`
- Si se "captura" la excepción se puede decir cómo actuar en consecuencia (retransmitir, finalizar el programa...)

2.5. Servidor iterativo vs. servidor concurrente

Como primera aproximación, se contempla la programación de un servidor secuencial que trata iterativamente cada nueva petición de cálculo de primos. No obstante, el tiempo que dicho cálculo puede ser elevado, dejando en espera a las sucesivas peticiones un tiempo no despreciable. Para aprovechar la capacidad de los computadores de ejecutar procesos en paralelo, **se plantea implementar como alternativa un servidor concurrente**. En este caso, cada nueva petición de cálculo se traslada a un hilo (*Thread*) de ejecución paralelo que permite seguir procesando nuevas peticiones mientras se realiza el cálculo y se envía la respuesta oportuna al cliente particular.

Así, en este caso se incorpora la clase `HandleRequest` (*HandleRequest.java*, de la que ser proporciona la plantilla), que implementa `Runnable` y debe responsabilizarse de servir la petición específica de cálculo de un cliente. Nuevamente, dada la naturaleza diferenciada del transporte TCP y UDP (orientado a conexión vs. no orientado a conexión), la implementación del correspondiente servidor concurrente y la clase `HandleRequest` será distinta en cada uno de los casos.

2.6. Manejo de excepciones

Sin considerar en la implementación ningún mecanismo de tolerancia a fallos, la compilación del código (y su correcta ejecución) nos puede exigir el tratamiento adecuado de excepciones. Únicamente será necesario implementar los `try/catch` involucrados en la gestión adecuada de los *sockets*, tal y como se refleja en los apuntes teóricos correspondientes.

3. Ejercicios

Se proporciona la clase **ya implementada *Primes.java***, donde se encuentra el método `GetPrimes`, que podrá utilizarse en la implementación de los servidores, tanto TCP como UDP (secuencial y/o concurrente).

Se diferencian **2 entregas** distinguiendo la implementación del servicio sobre transporte TCP o transporte UDP. Como entregas separadas se puntuarán independientemente, debiendo obtener en **cada una de ellas una nota igual o superior a 4 sobre 10**. De acuerdo a dichas entregas se plantean dos posibles ejercicios para cada entrega, como se detalla a continuación:

IMPORTANTE: Para poder verificar el correcto funcionamiento del servicio de cálculo remoto de números primos basado en una interacción cliente-servidor, es necesario la ejecución en *modo consola*, de modo que cada uno de los códigos (cliente o clientes y servidor) debe ejecutarse en ventanas de comando diferentes (por ejemplo, `cmd` en Windows).

Para ejecutar código en modo consola hay que ubicarse en la carpeta `bin` del proyecto con el que estamos trabajando. Si el código no está incluido en ningún `package` (**no** es el caso en nuestra práctica), los archivos .class se encontrarán directamente en dicha carpeta y la ejecución se realiza directamente así:

```
$java PrimeClient <ip/host> <puerto> <ini> <fin>
```

Supongamos ahora que el código se encuentra agrupado en el paquete (`package`): `p3.tcp`. Esto hará que los archivos .class no "cuelguen" directamente de la carpeta `bin` (y análogamente los .java de la `src`) sino que estén guardados en la ruta de carpetas `/p3/tcp/`. En este caso, el modo de ejecución desde la misma carpeta `bin` es el siguiente:

```
$java p3.tcp.PrimeClient <ip/host> <puerto> <ini> <fin>
```

3.1. Entrega 1: Transporte TCP

Existen dos posibles ejercicios a realizar en esta entrega: servidor iterativo o secuencial (Ejercicio 1, hasta 6 puntos) y servidor concurrente (Ejercicio 2, hasta 10 puntos). En el caso de realizar el servidor concurrente, **no deben entregarse los dos ejercicios**,

sino solo el ejercicio 2.

Ejercicio 1.

El objetivo de este ejercicio es realizar un servicio de cálculo remoto de números primos basado en una interacción cliente-servidor mediante el protocolo de transporte TCP y utilizando un **servidor secuencial**, que sirve a los clientes de uno en uno. Para ello, se proporcionan las siguientes plantillas a completar:

- *PrimeClient.java*: cliente que interactúa con el usuario (teclado/pantalla) y con un servidor remoto a través de una conexión TCP.
- *PrimeServer.java*: servidor que interactúa *secuencialmente* con clientes mediante una conexión TCP para realizar el cálculo de los primos. El esqueleto proporcionado incluye el método de cálculo de números primos.
- *Message.java*: clase que implementa cualquier tipo de mensaje (*REQUEST* o *REPLY*) de acuerdo al protocolo de comunicación especificado.
- *TransferMessage.java*: clase que encapsula los métodos de Java que facilitan el envío de los mensajes mediante `Serializable` (`writeObject`, `readObject`).

Dichas clases se encuentran dentro del paquete `p3.tcp`, que deberá modificarse al paquete `alumnos.p3.tcp.apellido` como se indica a continuación.

Puntuación: 6 puntos

Completa los códigos fuente de las clases `PrimeClient`, `PrimeServer`, `Message` *y* `TransferMessage`. *Todos los .java deberán pertenecer a un mismo paquete (*`package`*):* `alumnos.p3.tcp.apellido` *y deben guardarse en la ruta de carpetas* `/alumnos/p3/tcp/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 de tcp:
package alumnos.p3.tcp.canales – archivos en carpetas: alumnos/p3/tcp/canales/.java*
package alumnos.p3.tcp.gallego – archivos en carpetas: alumnos/p3/tcp/gallego/.java*

Ejercicio 2.

El objetivo de este ejercicio es realizar un servicio de cálculo remoto de números primos basado en una interacción cliente-servidor mediante el protocolo de transporte TCP y utilizando un **servidor concurrente**, que puede servir a varios clientes simultáneamente. Para ello, se proporcionan las siguientes plantillas a completar:

- *PrimeClient.java*: cliente que interactúa con el usuario (teclado/pantalla) y con un servidor remoto a través de una conexión TCP.
- *PrimeServer.java*: servidor que interactúa *concurrentemente* con clientes mediante una conexión TCP para realizar el cálculo de los primos. El esqueleto proporcionado incluye el método de cálculo de números primos.
- *Message.java*: clase que implementa cualquier tipo de mensaje (*REQUEST* o *REPLY*) de acuerdo al protocolo de comunicación especificado.
- *TransferMessage.java*: clase que encapsula los métodos de Java que facilitan el envío de los mensajes mediante `Serializable` (`writeObject`, `readObject`).
- *HandleRequest.java*: clase para gestionar la tarea concurrente.

Dichas clases se encuentran dentro del paquete `p3.tcp`, que deberá modificarse al paquete `alumnos.p3.tcp.apellido` como se indica a continuación.

Puntuación: 10 puntos

Completa los códigos fuente de las clases `PrimeClient`, `PrimeServer`, `Message`, `TransferMessage` y `HandleRequest`. Todos los .java deberán pertenecer a un mismo paquete (`package`): `alumnos.p3.tcp.apellido` y deben guardarse en la ruta de carpetas `/alumnos/p3/tcp/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 de tcp:*
package alumnos.p3.tcp.canales – archivos en carpetas: alumnos/p3/tcp/canales/.java*
package alumnos.p3.tcp.gallego – archivos en carpetas: alumnos/p3/tcp/gallego/.java*

3.2. Entrega 2: Transporte UDP

Existen dos posibles ejercicios a realizar en esta entrega: servidor iterativo o secuencial (Ejercicio 3, hasta 6 puntos) y servidor concurrente (Ejercicio 4, hasta 10 puntos). En el caso de realizar el servidor concurrente, **no deben entregarse los dos ejercicios**,

sino solo el ejercicio 4.

Ejercicio 3.

El objetivo de este ejercicio es realizar un servicio de cálculo remoto de números primos basado en una interacción cliente-servidor mediante el protocolo de transporte UDP y utilizando un **servidor secuencial**, que sirve a los clientes de uno en uno. Para ello, se proporcionan las siguientes plantillas a completar:

- *PrimeClient.java*: cliente que interactúa con el usuario (teclado/pantalla) y con un servidor remoto a través de una conexión UDP.
- *PrimeServer.java*: servidor que interactúa *secuencialmente* con clientes mediante una conexión UDP para realizar el cálculo de los primos. El esqueleto proporcionado incluye el método de cálculo de números primos.
- *Message.java*: clase que implementa cualquier tipo de mensaje (*REQUEST* o *REPLY*) de acuerdo al protocolo de comunicación especificado.
- *TransferMessage.java*: clase que encapsula los métodos de Java que facilitan el envío de los mensajes mediante `Serializable` (`writeObject`, `readObject`).

Dichas clases se encuentran dentro del paquete `p3.udp`, que deberá modificarse al paquete `alumnos.p3.udp.apellido` como se indica a continuación.

Puntuación: 6 puntos

Completa los códigos fuente de las clases `PrimeClient`, `PrimeServer`, `Message` y `TransferMessage`. Todos los .java deberán pertenecer a un mismo paquete (`package`): `alumnos.p3.udp.apellido` y deben guardarse en la ruta de carpetas `/alumnos/p3/udp/apellido/`. *Recuerda: en este directorio guarda los códigos fuente (ficheros con extensión* `.java`*), no los ficheros objeto (extensión* `.class`*).*

***Nota:** El "apellido" se escribirá en minúscula, sin acentos, ni ñ. Si varios alumnos tienen el mismo apellido, se indicarán los dos apellidos.*

Como ejemplo, los profesores utilizaríamos, en el caso de tcp:

package alumnos.p3.udp.canales – archivos en carpetas: alumnos/p3/udp/canales/.java*

package alumnos.p3.udp.gallego – archivos en carpetas: alumnos/p3/udp/gallego/.java*

Ejercicio 4.

El objetivo de este ejercicio es realizar un servicio de cálculo remoto de números primos basado en una interacción cliente-servidor mediante el protocolo de transporte UDP y utilizando un **servidor concurrente**, que puede servir a varios clientes simultáneamente. Para ello, se proporcionan las siguientes plantillas a completar:

- *PrimeClient.java*: cliente que interactúa con el usuario (teclado/pantalla) y con un servidor remoto a través de una conexión UDP.
- *PrimeServer.java*: servidor que interactúa *concurrentemente* con clientes mediante una conexión UDP para realizar el cálculo de los primos. El esqueleto proporcionado incluye el método de cálculo de números primos.
- *Message.java*: clase que implementa cualquier tipo de mensaje (*REQUEST* o *REPLY*) de acuerdo al protocolo de comunicación especificado.
- *TransferMessage.java*: clase que encapsula los métodos de Java que facilitan el envío de los mensajes mediante `Serializable` (`writeObject`, `readObject`).
- *HandleRequest.java*: clase para gestionar la tarea concurrente.

Dichas clases se encuentran dentro del paquete `p3.udp`, que deberá modificarse al paquete `alumnos.p3.udp.apellido` como se indica a continuación.

Puntuación: 10 puntos

Completa los códigos fuente de las clases `PrimeClient`, `PrimeServer`, `Message`, `TransferMessage` y `HandleRequest`. Todos los .java deberán pertenecer a un mismo paquete (`package`): `alumnos.p3.udp.apellido` y deben guardarse en la ruta de carpetas `/alumnos/p3/udp/apellido/`*. Recuerda: en este directorio guarda los códigos fuente (ficheros con extensión* `.java`*), no los ficheros objeto (extensión* `.class`*).*
***Nota:** El "apellido" se escribirá en minúscula, sin acentos, ni ñ. Si varios alumnos tienen el mismo apellido, se indicarán los dos apellidos.*
Como ejemplo, los profesores utilizaríamos, en el caso de tcp:
package alumnos.p3.udp.canales – archivos en carpetas: alumnos/p3/tcp/canales/.java*
package alumnos.p3.udp.gallego – archivos en carpetas: alumnos/p3/tcp/gallego/.java*

4. 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.
Ejercicio 3	6 puntos.
Ejercicio 4	10 puntos.

5. 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 deben entregarse dos ejercicios de los cuatro propuestos: **Ejercicio 1 o Ejercicio 2 y Ejercicio 3 o Ejercicio 4.**
- 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 3 habilitada a tal efecto.
- **Fecha de entrega: no más tarde del día anterior al comienzo de la sesión de prácticas 8**:
  - Grupo `prs2` (921): 26 de abril.
  - Grupos `prs1`, `prs2` y `prs3` (931): 27 de abril.
  - Grupo `prs1` (921): 21 de abril.
  - Grupos `prs4` y `prs5` (921): 28 de abril.

  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.