

Bloque N°1.Programacion con sockets

I. Desarrollo de servicios no estandar

Ejercicio 1

En este ejercicio hay que implementar un servicio echo sobre el protocolo UDP. Ademàs, el servidor tendra que ser concurrente, de manera que podamos aceptar varias peticiones a la vez.

Empezare explicando el servidor concurrente:

```
//Creamos un datagram socket
DatagramSocket ds = null;
try {
    //Asignamos la direccion y el puerto al datagramSocket
    InetAddress localAddr = InetAddress.getByName(host: "localhost");
    int wellKnownPort = 3000; // Puerto conocido del servidor
    ds = new DatagramSocket(wellKnownPort, localAddr);
    //Creamos un byte buffer y un DatagramPacket para recibir datos
    byte[] buffer = new byte[2048];
    DatagramPacket datagram = new DatagramPacket(buffer,
    buffer.length);
    //Creamos un bucle infinito para que no se cierre el servidor
    while (true) {
        //Recibimos la peticion
        ds.receive(datagram);
        System.out.println(x: "Nueva peticion de servicio");
        // Inicio de una hebra para la peticion actual
        (new ServerUDP(datagram)).start();
    }
} catch (IOException e) {
    System.err.println("Error E/S en: " + e.getMessage());
} //Cerramos el server si hay algun problema
} finally {
    if (ds != null)
        ds.close();
}
```

Este codigo es lo perteneciente a el servidor concurrente UDP donde creamos un socket principal, el cual se encarga de recibir las peticiones y redirigirlas a otro servidor que iremos creando mediante hebras. Estos servidores secundarios se encargan solo de la peticion que se les envia y tras terminar se cierran. Es decir, al recibir una peticion, se crea una hebra, se redigire el mensaje recibido a este y el servidor principal se queda esperando mas peticiones mientras el secundario se encarga de el. Esto tiene una limitacion, el numero de hebras que nuestro pc soporte.

Ahora toca explicar los servidores secundarios, los cuales son simples servidores UDP que creamos en una nueva hebra:

```
public class ServerUDP extends Thread{
    private byte[] datos;
    private InetAddress address;
    private int port;

    //Construcor de la hebra, donde le pasamos un DatagramPacket y le asignamos los datos y direccion
    public ServerUDP(DatagramPacket D){
        this.datos = D.getData();
        this.address = D.getAddress();
        this.port = D.getPort();
    }
}
```

Como vemos, creamos una clase que extienda de Thread para poder crear nuevos threads con ella, y simplemente creamos un constructor al que le pasamos un DatagramPacket y quedarnos con los datos y la direccion desde donde hemos recibido el paquete.

```
//Inicio de la hebra
public void run() {
    try {
        //Creamos un datagramSocket y un datagramPacket al que le asignamos los datos
        DatagramSocket ds = new DatagramSocket();
        DatagramPacket dp = new DatagramPacket(datos, datos.length, address, port);
        //Enviamos los datos y cerramos el socket
        ds.send(dp);
        System.out.println(x: "Cerrando peticion");
        ds.close();
        // FIN DE LA HEBRA
    } catch (Exception e) {
    }
}
```

Ademas del constructor, tenemos el metodo run, el cual es el que se encarga de correr el codigo que queremos, el cual simplemente un socket, un datagramPacket y envia los datos.

Cuando termina de correr este codigo, se cierra el socket y la hebra termina.

Por ultimo, queda el cliente:

```
public class clientUDP {
    //Numero maximo de intentos
    private static int maxtries = 5;

    Run | Debug
    public static void main(String[] args) throws IOException {
        //Si el numero de args son distintos de 2 lanzamos una excepcion
        if ((args.length < 1) || (args.length > 2)) {
            throw new IllegalArgumentException(s: "Parameter(s): <Server> [<Port>]");
        }
        //Asignamos la direccion recibida por argumentos que sera la direccion del servidor
        InetAddress serverAddress = InetAddress.getBy_name(args[0]); // IP Servidor
        int servPort = Integer.parseInt(args[1]);
    }
}
```

Para iniciar el cliente tenemos que correrlo de la siguiente manera “java clientUDP <Server> [<Port>]”, ya que le pasamos por parametros la direccion del servidor y el puerto donde esta.

```

//Creamos un BufferedReader para leer desde la terminal
BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in));
//Creamos mensaje para guardar lo leído por la terminal
String mensaje="";
//Numero de intentos actual al mandar mensaje
int tries=0;
//Socket para enviar los mensajes
DatagramSocket socket = new DatagramSocket();

```

Tras esto simplemente creamos algunas variables útiles, como una para leer los mensajes desde la terminal, otra para tener estos mensajes como String. Además del número de intentos actual y el socket para enviar los datos.

```

//Bucle infinito, hasta que no escribamos "." no terminara
while(true){
    //Mostramos el prompt y esperamos que el usuario introduzca una línea
    System.out.print(s: "> ");
    mensaje=stdIn.readLine();
    //Si escribimos "." salimos del bucle
    if(mensaje.equals(anObject: ".")){
        break;
    }
    //Transformamos el mensaje en bytes y creamos un datagramPacket con el
    byte[] bytesToSend = mensaje.getBytes();
    DatagramPacket sendPacket = new DatagramPacket(bytesToSend, //Datagrama a enviar
    bytesToSend.length, serverAddress, servPort);
    //Lo enviamos y le damos un valor al timeout
    socket.send(sendPacket);
    socket.setSoTimeout(timeout: 500);
}

```

Ahora entramos en un bucle infinito para que podamos enviar los mensajes que queramos. Donde imprimo por pantalla "> " para indicar que se puede escribir un mensaje. Tras esto el código espera que el usuario introduzca algún mensaje. Si este es igual a "." terminamos el bucle, si no es así, introducimos el mensaje en un array de bytes y creamos un datagramPacket, el cual enviamos usando el socket y ponemos un timeout.

```

boolean recibido = false;
while(!recibido){

```

Ahora entramos en un bucle donde mandaremos el paquete hasta que recibamos una respuesta.

A partir de aquí, el código se divide en dos casos, si se recibe el paquete o no.

Vamos a empezar por el caso positivo:

```
try {
    //Creamos el datagramPacket para recibir la respuesta y lo esperamos
    DatagramPacket receivePacket = //Datagrama a recibir
    new DatagramPacket(new byte[bytesToSend.length], bytesToSend.length);
    socket.receive(receivePacket); // Podria no llegar nunca el datagrama de ECO
    //Si el mensaje recibido es de la direccion ip correcta, lo mostramos por pantalla
    if(receivePacket.getAddress().equals(serverAddress)){
        recibido = true;
        System.out.println("ECO:" + new String(receivePacket.getData()));
    }
}
```

Creamos un datagramPacket para recibir los datos el cual sera del tamaño del paquete enviado, y con el socket recibimos la respuesta. Si no la recibieramos en el tiempo puesto para timeout se lanzaria una excepcion.

En caso de recibirlo, comprobamos que el mensaje haya llegado desde la direccion correcta y lo imprimimos por pantalla. Ademas de poner como true la condicion usada en el while.

Si el paquete no llegara pasaria esto:

```
//En caso de timeout
}catch(SocketTimeoutException e) {
    //Aumentamos el numero de intentos
    tries++;
    System.out.println("Intento "+tries+" de "+maxtries);
    //Si el numero de intentos llega al maximo, salimos del bucle y cerramos la conexion
    if(tries==maxtries){
        System.out.println(x: "Maximo numero de intentos alcanzado");
        System.out.println(x: "Cerrando conexion...");
        break;
    }
}
```

Se lanzaria una excepcion que atraparíamos con este código, aumentaríamos el número de intentos y lo mostraríamos por pantalla.

Si el número de intentos es igual al número máximo de intentos, acabaríamos el bucle, en caso contrario volveríamos a enviar el mensaje.

```
if(!recibido){
    break;
}
```

Si no recibimos el ACK o escribimos ".", salimos del bucle y cerramos el socket.

Por ultimo, vamos a ver su funcionamiento. Primero abrimos ambos programas de la siguiente manera:

```
PS > java concurrentServerUDP
PS > java clientUDP localhost 3000
> |
```


Tras esto, todo esta listo para mandar peiticiones:

<pre>PS > java concurrentServerUDP Nueva peticion de servicio Cerrando peticion Nueva peticion de servicio Cerrando peticion</pre>	<pre>PS > java clientUDP localhost 3000 > hola ECO:hola > adios ECO:adios</pre>
---	--

Cada vez que lanzamos un mensaje desde el cliente, se inicia una nueva peticion junto a thread para contestar.

Si escribimos "." se cierra el cliente:

<pre>PS > java concurrentServerUDP Nueva peticion de servicio Cerrando peticion Nueva peticion de servicio Cerrando peticion </pre>	<pre>PS > java clientUDP localhost 3000 > hola ECO:hola > adios ECO:adios > . PS > </pre>
---	---

Si lanzamos una peticion con servidor ocurre lo siguiente:

<pre>PS > java concurrentServerUDP Nueva peticion de servicio Cerrando peticion Nueva peticion de servicio Cerrando peticion PS ></pre>	<pre>PS > java clientUDP localhost 3000 > hola Intento 1 de 5 Intento 2 de 5 Intento 3 de 5 Intento 4 de 5 Intento 5 de 5 Maximo numero de intentos alcanzado Cerrando conexion...</pre>
---	--

Ejercicio 2

Para este ejercicio realizaremos algo parecido al ejercicio anterior pero sobre el protocolo TCP. Para este caso usaremos la interfaz Executor para hacerlo concurrente

Vamos a empezar por la clase Executor:

```
public class executor {
    Run | Debug
    public static void main(String[] args) throws IOException {
        // Creamos un socket pasivo, un logger y un objeto de la clase Executor
        ServerSocket servSock = new ServerSocket(port: 7890);
        Logger logger = Logger.getLogger(name: "servidorDST-C3");
        Executor service = Executors.newCachedThreadPool();
        // Bucle principal
        while (true) {
            Socket clntSock = servSock.accept(); // Espera una peticion
            // Ejecuta una hebra y le asigna el servicio de eco
            service.execute(new EchoProtocol(clntSock, logger));
        }
    }
}
```

Esta clase usa la interfaz Executor, la cual nos permite controlar la gestion de hebras automaticamente, lo que nos da que si por alguna razon una hebra se elimina se cree otra igual, que si esta inactiva cierto tiempo se borre, etc.

En este codigo basicamente creamos un ServerSocket al que conectarnos, un logger y una instancia de Executor. Tras esto, entramos en un bucle el cual va aceptando conexiones y asignandole servicios echo.

Ahora toca la clase EchoProtocol:

```
public class EchoProtocol implements Runnable {
    private Socket clntSock; // Socket de datos
    private Logger logger; // Logger del servidor

    //Constructor del objeto al que le pasamos el socket del cliente y el servidor
    public EchoProtocol(Socket clntSock, Logger logger) {
        this.clntSock = clntSock;
        this.logger = logger;
    }
}
```

Esta clase implementa Runnable para que podamos crear hebras desde esta. Creamos las variables globales que vamos a usar, el socket del cliente y el logger. Ademas del constructor al que simplemente le pasamos estos ultimos.

Para crear la hebra simplemente usaremos:

```
//Iniciamos la hebra y lanzamos el metodo que se encarga del protocolo
public void run() {
    handleEchoClient(clntSock, logger);
}
```

Esto simplemente crea la hebra y lanza el metodo principal.

Ahora el metodo importante:

```
public static void handleEchoClient(Socket clntSock, Logger logger) {
    //Creamos un String que almacena los mensajes
    String line;
}
```

A este metodo le vamos a pasar por argumentos las variables de clase y creamos un String que usaremos luego.

```
try {
    System.out.println("Nuevo cliente, socket " + clntSock);
    //Creamos los objetos que se encargan de recibir los mensajes y de enviarlos
    BufferedReader in = new BufferedReader(new InputStreamReader(clntSock.getInputStream()));
    PrintWriter out = new PrintWriter(
        new BufferedWriter(new OutputStreamWriter(clntSock.getOutputStream()), autoFlush: true));
    //Mientras sea falso, nos quedamos en el bucle
    boolean salir = false;
}
```

Entramos en un try/catch por si acaso y mostramos que tenemos un nuevo cliente. Ahora creamos un objeto BufferedReader, el cual se encargara de leer los mensajes que lleguen al servidor, y un objeto PrintWriter, el cual se encargara de enviar los mensajes.

Por ultimo, creamos un booleano inicializado a false para el siguiente bucle:

```
while (!salir) {
    line = in.readLine(); // lectura socket cliente
    //Si la lectura es nula, es momento de salir del bucle
    if(line==null) {
        System.out.println(x: "Cerrando conexion");
        salir = true;
    } //Si no es nula, mostramos por pantalla el mensaje recibido y mandamos el mensaje de vuelta
    else{
        System.out.println("Nuevo mensaje: "+line);
        out.println(line);
    }
}
```

El cual va leyendo lineas hasta que una sea nula, en ese caso salimos del bucle. Si la linea leida no es nula, lo mostramos por pantalla y lo enviamos de vuelta.

Por ultimo cerramos el socket:

```
//Tras salir del bucle, cerramos el socket
clntSock.close();
//Tras esto la hebra se cerrara tambien
} catch (Exception e) {
    //e.printStackTrace();
}
```

Ahora toca el cliente, el cual solo contendra un metodo main.

```
public static void main(String[] args) throws IOException {
    //Creamos direccion del servidor, lo necesitamos aqui fuera para los catch
    InetAddress serverAddr = null;
    try {
        //Si la entrada no es igual a 2, lanzamos excepcion
        if ((args.length < 1) || (args.length > 2)) {
            throw new IllegalArgumentException(s: "Parameter(s): <Server> [<Port>]");
        }
        //Le damos valores a la direccion del servidor y puerto
        serverAddr = InetAddress.getByName(args[0]); // IP Servidor
        int serverPort = Integer.parseInt(args[1]);
    }
```

Primero leemos los args de entrada para darle valores a la direccion del servidor y el puerto, si esta entrada no es igual a 2 lanzamos excepcion.

```
//Creamos un socket con estos
Socket sockfd = new Socket(serverAddr, serverPort);
System.out.println("Conexión local" + serverAddr);
//Creamos un BufferedReader para que se encargue de recibir mensajes
//y un PrintWriter para que se encargue de enviarlos
BufferedReader in = new BufferedReader(
    new InputStreamReader(sockfd.getInputStream()));
PrintWriter out = new PrintWriter(new BufferedWriter(new OutputStreamWriter(sockfd.getOutputStream())),
    autoFlush: true);
```

Ahora creamos un socket con las variables anteriores y los objetos para enviar y recibir mensajes. En este caso, un BufferedReader para recibir y un PrintWriter para enviar.

```
//Con este BufferedReader leemos los mensajes de la terminal y los escribimos en el String
BufferedReader stdIn = new BufferedReader(
    new InputStreamReader(System.in));
String userInput; // Entrada por teclado
//Prompt
System.out.print(s: "> ");
//Leemos entrada por teclado, si no es nula seguimos
```

Lo siguiente es crear el BufferedReader que se encargara de leer la entrada por teclado, el string donde guardar este y mostrar el prompt

```
//Leemos entrada por teclado, si no es nula seguimos
while ((userInput = stdIn.readLine()) != null) {
    //Si escribimos punto salimos del bucle
    if (userInput.equals(anObject: "."))
        break; // finaliza con el "."
    //Enviamos mensaje
    out.println(userInput);
    //Recibimos mensaje
    System.out.println("echo: " + in.readLine());
    System.out.print(s: "> ");
}
```

Lo siguiente que tenemos es un bucle, el cual funcionara mientras la entrada no sea nula. Si escribimos un "." salimos el bucle, en caso contrario, enviamos el mensaje con "out.println(userInput)" y mostramos el mensaje recibido de vuelta.

```
//Cerramos tanto socket como BufferedReader y PrintWriter
out.close();
in.close();
stdIn.close();
sockfd.close();
```

Por ultimo, al salir del bucle anterior, cerramos todo para que no haya leaks de memoria.

Ahora nos queda ver el funcionamiento, iniciamos el servidor y cliente de la siguiente manera:

<pre>PS > java executor Nuevo cliente, socket Socket[addr=/127.0.0.1,port=4989 5,localport=7890]</pre>	<pre>PS > java clientTCP localhost 7890 Conexión locallocalhost/127.0.0.1 ></pre>
---	---

Al iniciar el servidor no vemos nada, pero tras lanzar el cliente se inicia una nueva hebra y se lanza el mensaje que vemos.

Si escribimos un mensaje vemos:

<pre>PS > java executor Nuevo cliente, socket Socket[addr=/127.0.0.1,port=4989 5,localport=7890] Nuevo mensaje: hola Nuevo mensaje: adios</pre>	<pre>PS > java clientTCP localhost 7890 Conexión locallocalhost/127.0.0.1 > hola echo: hola > adios echo: adios ></pre>
--	---

Y si escribimos ".", lo que significa que hemos cerrado el cliente vemos:

<pre>PS > java executor Nuevo cliente, socket Socket[addr=/127.0.0.1,port=4989 5,localport=7890] Nuevo mensaje: hola Nuevo mensaje: adios Cerrando conexion </pre>	<pre>PS > java clientTCP localhost 7890 Conexión locallocalhost/127.0.0.1 > hola echo: hola > adios echo: adios > . PS > </pre>
--	---

Lo que significa que el cliente se ha cerrado, y el servidor lo ha recibido.

II. Desarrollo de servicios estandar

Ejercicio 1

Para este ejercicio vamos a tener que implementar el protocolo TFTP, es decir, un protocolo de transferencia de ficheros trivial. Para realizar esto lo haremos creando un servidor y cliente sobre el protocolo UDP

Vamos a empezar por el servidor:

```
public class serverTFTP {  
    Run | Debug  
    public static void main(String[] args) throws IOException {  
        //Creamos el socket y el datagrama para recibir los paquetes  
        DatagramSocket socket = new DatagramSocket(port: 6789);  
        DatagramPacket paquete = new DatagramPacket(new byte[516], length: 516);  
        System.out.println(x: "-----");  
        System.out.println(x: "Servidor iniciado");  
        System.out.println(x: "-----");  
        while (true) {
```

Para iniciar simplemente lanzamos el programa, y lo primero que hacemos es crear un socket para recibir datos y un datagrama donde guardarlos.

Tras esto entramos en un bucle infinito y empezamos a recibir datos:

```
        //Limpiamos el paquete  
        paquete = new DatagramPacket(new byte[516], length: 516);  
        System.out.println(x: "Esperando entrada");  
        System.out.println(x: "-----");  
        socket.receive(paquete); // Recibe un datagrama del cliente  
        //Vemos que modo es  
        int mode = packet.getMode(paquete.getData());  
        //Si es 1 tenemos que enviar al cliente
```

Recibimos el paquete y vemos si es RRQ o WRQ la petición con el método “packet.getMode”, que es una clase que he creado con métodos que me sirven de ayuda:

```
        //Sacamos modo(WRQ O RRQ)  
        public static int getMode(byte[] mensaje){  
            return mensaje[1];  
        }
```

En este caso, simplemente miramos un byte y lo devolvemos como int.

A partir de aquí el programa se divide en dos opciones “RRQ” o “WRQ”. Voy a empezar por RRQ:

```
        //Si es 1 tenemos que enviar al cliente  
        if(mode==1){  
            try{  
                System.out.println(x: "RRQ");  
                //Guardamos la dirección y el puerto del cliente  
                InetAddress serveAddress = paquete.getAddress();  
                int servPort = paquete.getPort();  
                //Cargamos archivo  
                System.out.println(packet.getNameFile(paquete.getData()));  
                File archivo = new File(packet.getNameFile(paquete.getData()));  
                //Creamos el primer ACK falso solo para continuar con el código  
                DatagramPacket receivePacket = packet.ACK(bloque: 0, paquete.getAddress(), paquete.getPort());  
                socket.setSoTimeout(timeout: 1000);
```

Primero, vemos que modo es, en este caso RRQ.

Dentro de este modo, lo primero que hacemos es guardar la dirección del cliente, abrir el archivo y crear un ACK falso para que el programa continúe. En caso de que el archivo no exista se lanzaría una excepción que controlamos más abajo.

```
//Empezamos a enviar el archivo
if(packet.compACK(receivePacket.getData())==0){
    //Creamos un array de bytes y un FileInputStream para leer el archivo en bloques
    byte[] bytesToSend = new byte[512];
    FileInputStream fi = new FileInputStream(archivo);
    int bloque=0, tam=512;
    boolean terminado = false;
```

Como con "RRQ" nos toca enviar a nosotros el archivo, comenzamos creando las variables que usaremos para leerlo y controlar por que bloque vamos. Tras esto entramos en un bucle que no parara hasta que hayamos terminado de enviar el archivo:

```
while(!terminado){
    //Si lo que nos queda por enviar es mayor a 512 no lo enviamos
    if(fi.available()<512){
        tam=fi.available();
    }
    //Si hemos recibido ACK del ultimo bloque leido continuamos leyendo el archivo
    if(packet.compACK(receivePacket.getData())==bloque){
        bloque++;
        bytesToSend = new byte[512];
        fi.read(bytesToSend, off: 0, tam);
    }
}
```

Donde vemos si nos queda mas de 512 bytes por enviar, si no cambiamos el tamaño de lo que vamos a leer.

Despues comprobamos si el ultimo ACK recibido es igual al paquete mandado, en caso afirmativo, leemos el siguiente bloque del archivo. En otro caso lo reenviamos. Tras esto, enviamos el paquete:

```
//Enviamos el paquete
DatagramPacket send = packet.dataPacket(bytesToSend, bloque, serveAddress, servPort);
socket.send(send);
//Recibimos el ACK
try{
    socket.receive(receivePacket);
    socket.setSoTimeout(timeout: 1000);
    System.out.println("ACK "+packet.compACK(receivePacket.getData()));
    //Si nos da timeout, es que se ha perdido
    //Creamos ACK falso con un bloque anterior para que reenvie
} catch (IOException e){
    receivePacket = packet.ACK(bloque-1, serveAddress, servPort);
}
```

Creamos un datagrama con el metodo de packet:

```
//Creamos datagrama con los datos del archivo, el bloque y la direccion del servidor o cliente
public static DatagramPacket dataPacket(byte[] datos, int bloque, InetAddress serverAddress, int port) throws IOException{
    ByteArrayOutputStream outputStream = new ByteArrayOutputStream( );
    outputStream.write(new byte[]{0,3}, off: 0, len: 2);
    outputStream.write((byte)((bloque>>>8)&0xFF));
    outputStream.write((byte)(bloque&0xFF));
    outputStream.write(datos, off: 0, datos.length);

    return new DatagramPacket(outputStream.toByteArray(), outputStream.toByteArray().length, serverAddress, port);
}
```

Donde simplemente le pasamos los datos, el boque y la direccion de donde queremos enviarlo, y nos devuelve un datagrama listo para enviar.

Ahora enviamos el datagrama y esperamos la respuesta. En caso de que no la recibamos, significa que se ha perdido el paquete, así que creamos un ACK para que no continúe la lectura del archivo.

Por último, vemos como funciona la finalización de enviar:

```
//Si recibimos ACK del ultimo bloque y no nos queda nada por leer hemos terminado
if(packet.compACK(receivePacket.getData())==bloque && fi.available()==0){
    terminado = true;
}
```

Donde como vemos, si hemos recibido el ACK del último bloque y no nos queda nada por leer, significa que hemos terminado.

```
//Si el archivo no existe se lanza esto
} catch (FileNotFoundException e){
    System.out.println("Archivo "+packet.getNameFile(paquete.getData())+" no encontrado");
}
```

Para finalizar, si el archivo no existe, simplemente mostramos un mensaje de error.

Ahora nos toca ver el modo “WRQ”, lo que significa que nos toca recibir datos desde el cliente:

```
//Si es 2 recibimos del cliente
}else if(mode==2){
    System.out.println(x: "WRQ");
    //Enviamos el ACK 0
    socket.send(packet.ACK(bloque: 0, paquete.getAddress(), paquete.getPort()));
    boolean wrq = true;
    //Numero de paquetes totales recibidos(Incluidos los perdidos) y paquetes perdidos
    int paquetesTotales = 0, paquetesPerdidos=0;
    //Bucle infinito para recibir
```

Si en el primer paquete recibido, tenemos un 2, significa que es WRQ, así tenemos que enviar ACK 0 al cliente. También creamos un par de variables para luego mostrar las estadísticas de la transmisión.

```
while(wrq){
    try {
        //Recibimos paquetes
        socket.receive(paquete);
        Random a = new Random();
        paquetesTotales++;
        //Si el numero aleatorio es mayor a 0.05 hemos recibido el paquete
        if(a.nextDouble(origin: 0, bound: 1)>0.05){
            //Si es menor a 512 bytes hemos terminado
            if(packet.dataLength(paquete.getData())<512){
                wrq=false;
                System.out.println("DATA "+packet.getBloque(paquete.getData())+" \ "<"+512+" b\");
            }else{
                System.out.println("DATA "+packet.getBloque(paquete.getData())+" \ "<"+packet.dataLength(paquete.getData())+" b\");
            }
        }
        //Enviamos ACK
        socket.send(packet.ACK(packet.getBloque(paquete.getData()), paquete.getAddress(), paquete.getPort()));
    }
```

Ahora entramos en este bucle infinito, donde recibimos el archivo, creamos un objeto Random y aumentamos el número de paquetes recibidos.

Tras esto tenemos un if, el cual comprueba si el siguiente número creado por el objeto random es mayor a 0.05 significa que hemos recibido el paquete.

Dentro de este if tenemos dos posibilidades:

- Que el paquete sea menor a 512 bytes, lo que significa que hemos terminado
- Que el paquete sea igual a 512 bytes, por lo que vamos a continuar recibiendo paquetes.

Por último, enviamos el ACK del bloque.

Si hemos perdido el paquete, mostramos lo siguiente por pantalla:

```
//Hemos perdido el paquete
}else{
    System.out.println("DATA "+packet.getBloque(paquete.getData()) +" (p)");
    paquetesPerdidos++;
}
```

Antes de terminar, mostramos las estadísticas si hemos recibido algún paquete:

```
//Si hemos recibido algún paquete, mostramos las estadísticas
if(paquetesTotales>0){
    System.out.println("%"+Integer.toString((paquetesPerdidos*100)/paquetesTotales)+":paquetes perdidos; %"+
    Integer.toString((100-(paquetesPerdidos*100)/paquetesTotales))+": paquetes retransmitidos");
}
```

Lo siguiente por explicar es el cliente, el cual lo he basado en leer desde la terminal con lo siguiente:

```
String mensaje="";
BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in));
System.out.println(x: "Usa \"help\" para ver informacion sobre los comandos");
//Entramos en el bucle principal donde tenemos todas las opciones
while(!mensaje.equals(anObject: "quit")){
```

Y entrar en un bucle infinito con varios if dependiendo de donde entremos, vamos en orden. Primero toca el connect:

```
//Mostramos el prompt y leemos
System.out.print(s: "> ");
mensaje=stdIn.readLine();
//Si el mensaje es connect entramos aqui
if(mensaje.contains(s: "connect")){
    //Separamos el mensaje por el espacio
    String[] aux = mensaje.split(regex: " ");
    //Si el mensaje tiene menos de 2 args
    if(aux.length<2){
        System.out.println(x: "Para conectarte a un servidor usa: connect <Server>");
    }
    //Si el mensaje tiene 2 args, ponemos como direccion el segundo argumento y creamos un socket
    }else{
        servPort = 6789;
        serveAddress = InetAddress.getByName(aux[1]);
        socket = new DatagramSocket();
        System.out.println("Conectado a servidor "+serveAddress+": "+servPort);
        System.out.println(x: "Si quiere conectarse a otro servidor vuelva a usar connect");
    }
}
```

En cada ciclo del bucle, mostramos el prompt y leemos el mensaje que haya escrito el usuario. Si este contiene “connect” entramos en la primera opción, donde tenemos que si el mensaje no tiene al menos dos argumentos, mostramos el mensaje de error. Si tiene los argumentos necesarios guardamos la información del servidor al que vamos a conectarnos y creamos un socket.

El siguiente que tenemos es el comando “help”:

```
//Si el mensaje es help entramos aqui, para ver todos los comandos disponibles
}else if(mensaje.contains(s: "help")){
    System.out.println(x: "- Para conectarte a un servidor usa: connect <Server>");
    System.out.println(x: "- Para recibir un archivo usa: get <Server>");
    System.out.println(x: "- Para mandar datos al servidor usa: put <fileName>");
    System.out.println(x: "- Para cerrar el cliente usa: quit");
}
```

El cual simplemente muestra todos los posibles comandos del programa.
El siguiente a ver es el comando “get” el cual hace que el servidor nos mande un archivo:

```
//Si el mensaje contiene get entramos aqui
}else if(mensaje.contains(s: "get")){
    //Primero necesitamos conectarnos a un server para usar get
    if(serveAddress==null){
        System.out.println(x: "Primero usa el comando connect");
    }else{
        //Dividimos el mensaje de entrada
        String[] aux = mensaje.split(regex: " ");
        //Si tiene menos de dos argumentos mostramos error
        if(aux.length<2){
            System.out.println(x: "Para recibir un archivo usa: get <Server>");
        }else{
```

Lo primero que tenemos, es que si no nos hemos conectado a ningun servidor, es lo primero que tenemos que hacer.

Tras esto se comprueba si estan los argumentos necesarios, en caso de que falta algo se muestra por pantalla un error. En otro caso continuamos y empezamos con la transmision de datos:

```
//Enviamos el datagrama RRQ
DatagramPacket dp = packet.createRRQ(aux[1], serveAddress, servPort);
socket.send(dp);
socket.setSoTimeout(timeout: 1000);
//Creamos el datagrama para recibir el archivo
DatagramPacket paquete = new DatagramPacket(new byte[516], length: 516);
boolean wrq = true;
//Numero de paquetes totales recibidos(Contado los que se pierden) y numero de paquetes perdidos
int paquetesTotales = 0, paquetesPerdidos=0;
//Bucle infinito para recibir el archivo
while(wrq){
```

Primero enviamos un datagrama RRQ para informar al servidor de lo que queremos.
Tras esto creamos un datagrama para recibir el archivo, unas variables numericas para las estadisticas y entramos en un bucle infinito donde recibimos el archivo:

```
//Recibimos el siguiente bloque
socket.receive(paquete);
Random a = new Random();
paquetesTotales++;
//Si el numero es mayor a 0.05 hacemos como que lo hemos recibido
if(a.nextDouble(origin: 0, bound: 1)>0.05){
    //Si el paquete es menor a 512 bytes hemos terminado
    if(packet.dataLenght(paquete.getData())<512){
        wrq=false;
        System.out.println("DATA "+packet.getBloque(paquete.getData()) +" \<"+512+" b\");
    }else{
        System.out.println("DATA "+packet.getBloque(paquete.getData()) +" \<"+packet.dataLenght(paquete.getData())+" b\");
    }
    //Enviamos el ACK
    socket.send(packet.ACK(packet.getBloque(paquete.getData()), paquete.getAddress(), paquete.getPort());
}
//Se pierde el paquete
}else{
    paquetesPerdidos++;
    System.out.println("DATA "+packet.getBloque(paquete.getData()) +" (p)");
}
```

Como se puede ver, el codigo es exactamente igual al modo WRQ del servidor, asi que se puede leer mas arriba.

En caso de que el archivo no existe, recibiremos timeout y lo controlaremos con:

```
//Si recibimos timeout es que el archivo no existe
} catch (IOException e) {
    System.out.println(x: "Archivo no encontrado");
    wrq=false;
}
```

Por ultimo, mostramos las estadísticas si hemos recibido algun paquete:

```
//Si hemos recibido algun paquete mostramos las estadísticas
if(paquetesTotales>0){
    System.out.println("%"+Integer.toString((paquetesPerdidos*100)/paquetesTotales)+":paquetes perdidos; %"
    +Integer.toString((100-(paquetesPerdidos*100)/paquetesTotales))+": paquetes retransmitidos");
}
System.out.println(x: "Transmision finalizada");
```

Ahora nos queda la ultima parte del cliente, el comando “put” el cual se encarga de enviar archivos:

```
//Para usar este comando primero necesitamos un servidor al que enviar los datos
if(serveAddress==null){
    System.out.println(x: "Primero usa el comando connect");
}else{
    //Dividimos el mensaje
    String[] aux = mensaje.split(regex: " ");
    //Si tiene menos de dos argumentos mostramos error
    if(aux.length<2){
        System.out.println(x: "Para mandar datos al servidor usa: put <fileName>");
    }else{
```

Primero hacemos las mismas comprobaciones que para el comando “get”.

Tras esto, hacemos lo siguiente:

```
//Cargo el archivo
File archivo = new File(aux[1]);
//Mando WRQ
DatagramPacket dp = packet.createWRQ(aux[1], serveAddress, servPort);
socket.send(dp);
socket.setSoTimeout(timeout: 1000);
//Creo el datagrama de los ACK
DatagramPacket receivePacket = new DatagramPacket(new byte[4], length: 4);
//Recibimos el primero datagrama que sera el ACK 0
socket.receive(receivePacket);
System.out.println("ACK "+packet.compACK(receivePacket.getData()));
//Si hemos recibido el ACK 0 empezamos a enviar el archivo
```

Cargamos el archivo, y creamos un datagrama con la estructura WRQ. Esto lo enviamos al servidor y esperamos que nos envíe un ACK 0 el cual guardamos en receivePacket.

Si lo recibimos:

```
//Si hemos recibido el ACK 0 empezamos a enviar el archivo
if(packet.compACK(receivePacket.getData())==0){
    //Creamos el array de bytes donde vamos a guardar el archivo y
    //FileInputStream para ir leyendo el archivo por cachitos
    byte[] bytesToSend = new byte[512];
    FileInputStream fi = new FileInputStream(archivo);
    int bloque=0, tam=512;
    boolean terminado = false;
```

Ahora creamos un array de bytes para guardar el archivo y un FileInputStream con el cual podemos leer el archivo por bloques. Ademas de algunas variables de ayuda.

Empezamos a enviar el archivo:

```
//Bucle de enviar el archivo
while(!terminado){
    //Mientras nos quede mas de 512bytes por leer del archivo no cambiamos el tamaño
    if(fi.available()<512){
        tam=fi.available();
    }
    //Si hemos recibido el ack con el ultimo bloque mandado, leemos el siguiente
    if(packet.compACK(receivePacket.getData())==bloque){

        bloque++;
        bytesToSend = new byte[512];
        fi.read(bytesToSend, off: 0, tam);
    }
}
```

Donde entramos en un bucle infinito, y vamos comprobando si tenemos que cambiar el tamaño de lectura a uno menor a 512. Tras esto, si el anterior ACK recibido es igual al bloque enviado, continuamos leyendo el archivo.

Ahora toca enviar el paquete:

```
//Creamos el datagrama a enviar y enviamos el paquete
DatagramPacket send = packet.dataPacket(bytesToSend, bloque, serveAddress, servPort);
socket.send(send);
//Esperamos un segundo a recibir el paquete, en caso contrario se ha perdido y creamos
//un ACK falso para que no continúe leyendo el archivo y reenvíe
try{
    socket.receive(receivePacket);
    socket.setSoTimeout(timeout: 1000);
    System.out.println("ACK "+packet.compACK(receivePacket.getData()));
} catch (IOException e){
    receivePacket = packet.ACK(bloque-1, serveAddress, servPort);
}
//Si nos queda 0 bytes por enviar y hemos recibido el ultimo ACK hemos terminado
if(packet.compACK(receivePacket.getData())==bloque && fi.available()==0){
    terminado = true;
}
```

Creamos un datagrama con los datos y el bloque, el cual enviamos.

Ahora esperamos a recibir el ACK de respuesta. Si no lo recibimos creamos uno falso para que no continúe leyendo el archivo y lo reenvíe.

Si el ACK recibido es igual al paquete enviado y no nos queda nada por leer del archivo, hemos terminado.

A lo largo del código, se ha podido ver una clase llamada “packet” con numerosos métodos. Es una clase que he creado de ayuda la cual simplemente contiene métodos que voy a explicar.

```
//Creamos datagrama para el RRQ simplemente pasando el nombre del archivo y la dirección del servidor
public static DatagramPacket createRRQ(String filename, InetAddress serverAddress, int port){
    ByteArrayOutputStream outputStream = new ByteArrayOutputStream( );
    outputStream.write(new byte[]{0,1}, off: 0, len: 2);
    outputStream.write(filename.getBytes(), off: 0, filename.getBytes().length);
    outputStream.write(new byte[]{0,0,0,0}, off: 0, len: 4);
    return new DatagramPacket(outputStream.toByteArray(), outputStream.toByteArray().length, serverAddress, port);
}

//Creamos datagrama para el WRQ simplemente pasando el nombre del archivo y la dirección del servidor
public static DatagramPacket createWRQ(String filename, InetAddress serverAddress, int port){
    ByteArrayOutputStream outputStream = new ByteArrayOutputStream( );
    outputStream.write(new byte[]{0,2}, off: 0, len: 2);
    outputStream.write(filename.getBytes(), off: 0, filename.getBytes().length);
    outputStream.write(new byte[]{0,0,0,0}, off: 0, len: 4);
    return new DatagramPacket(outputStream.toByteArray(), outputStream.toByteArray().length, serverAddress, port);
}
```

Con esos métodos, pasándole el nombre del archivo y la dirección del servidor creo los datagramas WRQ y RRQ.

```
//Creamos datagrama con los datos del archivo, el bloque y la dirección del servidor o cliente
public static DatagramPacket dataPacket(byte[] datos, int bloque, InetAddress serverAddress, int port) throws IOException{
    ByteArrayOutputStream outputStream = new ByteArrayOutputStream( );
    outputStream.write(new byte[]{0,3}, off: 0, len: 2);
    outputStream.write((byte)((bloque>>>8)&0xFF));
    outputStream.write((byte)(bloque&0xFF));
    outputStream.write(datos, off: 0, datos.length);

    return new DatagramPacket(outputStream.toByteArray(), outputStream.toByteArray().length, serverAddress, port);
}
```

A este le paso los datos del archivo a mandar, el bloque y la dirección, y me devuelve un datagrama listo para enviar.

```
//Sacamos número de ACK
public static int compACK(byte[] ack){
    return Integer.parseInt(Integer.toString(ack[2])+Integer.toString(ack[3]));
}

//Sacamos modo(WRQ o RRQ)
public static int getMode(byte[] mensaje){
    return mensaje[1];
}

//Sacamos bloque
public static int getBloque(byte[] mensaje){
    return Integer.parseInt(String.valueOf(mensaje[2])+String.valueOf(mensaje[3]));
}
```

Con estos métodos, consigo el número de ACK, el modo y el bloque para cuando sea necesario.

```
//Creamos ACK pasándole el bloque y la dirección del servidor o cliente
public static DatagramPacket ACK(int bloque, InetAddress serverAddress, int port) throws IOException{
    ByteArrayOutputStream outputStream = new ByteArrayOutputStream( );
    outputStream.write(new byte[]{0,4}, off: 0, len: 2);
    outputStream.write((byte)((bloque>>>8)&0xFF));
    outputStream.write((byte)(bloque&0xFF));
    return new DatagramPacket(outputStream.toByteArray(), outputStream.toByteArray().length, serverAddress, port);
}
```

Con este creo datagramas con el bloque que necesite de manera fácil.

```

//Sacamos la longitud de los datos del archivo enviados
//Copiamos los ultimos 512bytes a un array, y contamos hasta que haya 0 seguidos
public static int dataLenght(byte[] data){
    data = Arrays.copyOfRange(data, from: 4, data.length);
    int ceros = 0;
    for(byte e : data){
        if(e==0){
            ceros++;
        }else if(e!=0){
            ceros=0;
        }
    }

    return Arrays.copyOfRange(data, from: 0, data.length-ceros).length;
}

```

Con este metodo consigo saber cuantos bytes han sido enviados, ya que estos pueden ser menores a 512 y es necesario saberlo para saber si he terminado de recibir el archivo.

```

//Sacamos nombre de archivo desde un datagrama
public static String getNameFile(byte[] data){
    data = Arrays.copyOfRange(data, from: 2, data.length);
    int ceros = 0;
    for(byte e : data){
        if(e==0){
            ceros++;
        }else if(e!=0){
            ceros=0;
        }
    }

    return new String(Arrays.copyOfRange(data, from: 0, data.length-ceros));
}

```

Por ultimo, con este consigo el nombre del archivo de los paquetes recibidos.

Ahora nos toca ver lo mas importante, su funcionamiento.

Iniciamos el servidor y cliente de la siguiente manera:

<pre> PS > java serverTFTP ----- Servidor iniciado ----- Esperando entrada ----- </pre>	<pre> PS > java clientTFTP Usa "help" para ver informacion sobre los comandos > </pre>
--	--

Antes que nada, vamos a ver el comando help:

<pre> PS > java serverTFTP ----- Servidor iniciado ----- Esperando entrada ----- </pre>	<pre> PS > java clientTFTP Usa "help" para ver informacion sobre los comandos > help - Para conectarte a un servidor usa: connect <Server> - Para recibir un archivo usa: get <Server> - Para mandar datos al servidor usa: put <fileName> > </pre>
--	--

De esta manera podemos ver todos los comandos del cliente.

No hemos usado aun el comando “connect”, así que vamos a ver que ocurre si usamos get o put:

```
PS > java clientTFTP
Usa "help" para ver informacion sobre los comandos
> help
- Para conectarte a un servidor usa: connect <Server>
- Para recibir un archivo usa: get <Server>
- Para mandar datos al servidor usa: put <fileName>
> get hola.txt
Primero usa el comando connect
> put hola.txt
Primero usa el comando connect
> |
```

Ahora, vamos a conectarnos al servidor:

```
PS > java clientTFTP
Usa "help" para ver informacion sobre los comandos
> connect localhost
Conectado a servidor localhost/127.0.0.1:6789
Si quiere conectarse a otro servidor vuelva a usar connect
> |
```

Primero vamos a ver el comando “put” en funcionamiento:

<pre>PS > java serverTFTP ----- Servidor iniciado ----- Esperando entrada ----- WRQ DATA 1 "512 b" DATA 2 "512 b" DATA 3 "512 b" DATA 4 "512 b" DATA 5 "512 b" DATA 6 "512 b" DATA 7 "512 b" DATA 8 "512 b" DATA 9 "512 b" DATA 10 "512 b" DATA 11 "512 b" %0:paquetes perdidos; %100: paquetes retransmitidos Transmision finalizada</pre>	<pre>PS > java clientTFTP Usa "help" para ver informacion sobre los comandos > connect localhost Conectado a servidor localhost/127.0.0.1:6789 Si quiere conectarse a otro servidor vuelva a usar connect > put loren.txt ACK 0 ACK 1 ACK 2 ACK 3 ACK 4 ACK 5 ACK 6 ACK 7 ACK 8 ACK 9 ACK 10 ACK 11 ></pre>
--	---

Como vemos, podemos ver todos los ACK recibidos por el cliente y todos los datos recibidos por el servidor. Además, al final de la transmisión en el servidor se nos muestran las estadísticas de la transmisión.

Por ultimo nos queda el comando “get”:

<pre>DATA 11 "<512 b" %0:paquetes perdidos; %100: paquetes retransmitidos Transmision finalizada ----- Esperando entrada ----- RRQ lorem.txt ACK 0 ACK 1 ACK 2 ACK 3 ACK 4 ACK 5 ACK 6 ACK 7 ACK 8 ACK 9 ACK 10 ACK 11</pre>	<pre>ACK 11 > get lorem.txt DATA 1 "512 b" DATA 2 "512 b" DATA 3 "512 b" DATA 4 (p) DATA 4 (p) DATA 4 "512 b" DATA 5 "512 b" DATA 6 "512 b" DATA 7 "512 b" DATA 8 "512 b" DATA 9 "512 b" DATA 10 "512 b" DATA 11 "<512 b" %15:paquetes perdidos; %85: paquetes retransmitidos Transmision finalizada ></pre>
---	---

A la izquierda vemos los ACK recibidos por el servidor, ya que es el que envia archivos y a la derecha los datos recibidos por el cliente ya que es el que los recibe. Ademas, podemos ver que algunos paquetes se han perdido (En concreto el 4) y las estadísticas.

III. Patrones de diseño

Ejercicio 2

Para este ejercicio hay que implementar un servidor multi-protocol el cual funcionara igual que los servidores echo realizados en el primer bloque de la practica. Con servidor multi-protocolo nos referimos a un servidor capaz de aceptar peticiones por protocolo UDP y TCP de manera concurrente.

Para implementar esto hay que usar el paquete NIO (Selector) como programa para el servidor. Para los clientes usaremos los usados en los Ejercicios 1 y 2 del primero bloque de la practica.

Vamos con el servidor, la cual solo contendra un metodo main. Empezemos por el principio:

```
//Creo el selector y un buffer
Selector selector = Selector.open();
//ByteBuffer buffer = ByteBuffer.allocate(512);
//Creo canal de TCP
ServerSocketChannel tcp = ServerSocketChannel.open();
tcp.socket().bind(new InetSocketAddress(port: 9000));
tcp.configureBlocking(block: false);
SelectionKey serverKey = tcp.register(selector, SelectionKey.OP_ACCEPT);
//Creo canal de UDP
DatagramChannel udp = DatagramChannel.open();
udp.socket().bind(new InetSocketAddress(port: 9000));
udp.configureBlocking(block: false);
SelectionKey channelKey = udp.register(selector, SelectionKey.OP_READ, new ClientRecord());
```

Creamos un objeto de la clase Selector. Tras esto creamos un ServerSocketChannel, el cual usaremos para las conexiones TCP. A este le asignamos el puerto que vamos a usar. Por ultimo, lo configuramos como lectura no bloqueante y lo registramos junto a la key "OP_ACCEPT" la cual se usa para canales pasivo.

Para la parte UDP, creamos un DatagramChannel, lo bindeamos al puerto y lo configuramos como lectura no bloqueante. Ademas de registrarlo junto a una key de lectura y un buffer de datos donde guardar lo leido.

En este caso, he usado un objeto de la clase ClientRecord, la cual contiene:

```
public class ClientRecord {
    public ByteBuffer buffer = ByteBuffer.allocate(capacity: 512);
    public SocketAddress clientAddress;
}
```

Como vemos, simplemente contiene un ByteBuffer de tamaño 512, donde vamos a escribir y un SocketAddress donde guardaremos la direccion del cliente.

Lo siguiente a hacer es iterar sobre los canales que hemos creado de la siguiente manera:

```
for (;;) {
    selector.select();
    Set keys = selector.selectedKeys();
    for(Iterator i = keys.iterator(); i.hasNext();) {
        SelectionKey key = (SelectionKey) i.next();
        i.remove();
    }
}
```

Creamos un bloque infinito, donde con la primera intruccion esperamos hasta que al menos un canal este activo, creamos un Set con las keys de los canales activos e iteramos sobre este. Conforme vamos avanzando, guardamos la key actual en una vaiable y la borramos.

Lo siguiente que tenemos en el programa es ir comprobando que tipo de key es. El primer caso es para TCP:

```
//Si recibo una conexcion TCP entro aqui
if(key == serverKey){
    //Si la conexcion es aceptada, le creo un canal de lectura al selector
    if(key.isAcceptable()) {
        SocketChannel client = tcp.accept();
        client.configureBlocking(block: false);          You, last week * FTP y NIO
        System.out.println(x: "Conexion TCP aceptada");
        SelectionKey clientKey = client.register(selector, SelectionKey.OP_READ, ByteBuffer.allocate(capacity: 1024));
    }
    //Si no la rechazo
} else {
    System.out.println(x: "Conexion TCP no aceptada");
    if(!key.isReadable()){
        continue;
    }
}
```

Aqui vemos si la key es una serverKey, el tipo de TCP. Ahora vemos si el cliente no esta asignado a un canal, en ese caso, creamos un SocketChannel con la direccion del cliente y creamos un canal con la key "OP_READ" para que sea de lectura. Ademas, le asignamos un ByteBuffer para guardar los mensajes recibidos. En caso de estar asignado, simplemente no lo aceptamos.

```
//Entro aqui si recibo un mensaje TCP lo leo
}else if(key.isReadable()){

    SocketChannel client = (SocketChannel) key.channel();
    ByteBuffer buffer2 = (ByteBuffer) key.attachment();
    buffer2.clear();
    int bytesread = client.read(buffer2);
    //Con esto cierro la conexcion si el cliente se cierra
    if(bytesread==-1){
        key.cancel();
        client.close();
        System.out.println(x: "Conexion TCP cerrada");
        continue;
    }
    key.interestOps(SelectionKey.OP_READ | SelectionKey.OP_WRITE);
}
```

Si la key es de lectura, es decir, de un mensaje TCP, entramos ahí. Aquí creamos un socket con la direccion del cliente y un buffer, estos dos con los datos que tiene el key. Limpiamos el buffer por si hay basura y leemos del socket al buffer. Ademas, guardamos la cantidad de bytes leídos para usarlos mas adelante.

Si esta cantidad leída es igual a -1, eliminamos la key del programa y cerramos el socket.

En otro caso, cambiamos el tipo del key a escritura y lectura.

Cuando tengamos una key valida y de escritura, entrariamos aqui:

```
//Aquí envío el mensaje TCP
}else if(key.isValid() && key.isWritable()){

    ByteBuffer buf = (ByteBuffer) key.attachment();
    buf.flip();
    SocketChannel client = (SocketChannel) key.channel();
    client.write(buf);
    System.out.println("Mensaje TCP: "+new String(buf.array()).trim());
    if(!buf.hasRemaining()) {
        key.interestOps(SelectionKey.OP_READ);
    }
    buf.compact();
}
```

Creamos un ByteBuffer con el que hay asignado a la key y usamos el metodo flip en este. Lo que hace basicamente que pase a estar leyendo a escribir.

Tras esto, creamos un socket con la direccion del cliente y enviamos el mensaje. Si tras enviar los datos, el buffer se queda vacio, cambiamos el tipo de key a solo lectura.

Antes de terminar de enviar, usamos el metodo "compact" en el buffer, lo cual simplemente reinicia la posicion de escritura.

Ahora toca el codigo para las peticiones UDP:

```
//Si es una conexion UDP entro aqui
}else if((key.isValid()) && (key==channelKey)){
    //Leo el mensaje recibido
    if(key.isReadable()){
        DatagramChannel channel = (DatagramChannel) key.channel();
        ClientRecord cr = (ClientRecord) key.attachment();
        cr.buffer.clear();
        cr.clientAddress = channel.receive(cr.buffer);
        if(cr.clientAddress!=null){
            key.interestOps(SelectionKey.OP_WRITE);
        }
    }
}
```

Si la key es valida y channelKey sabemos que es una peticion UDP.

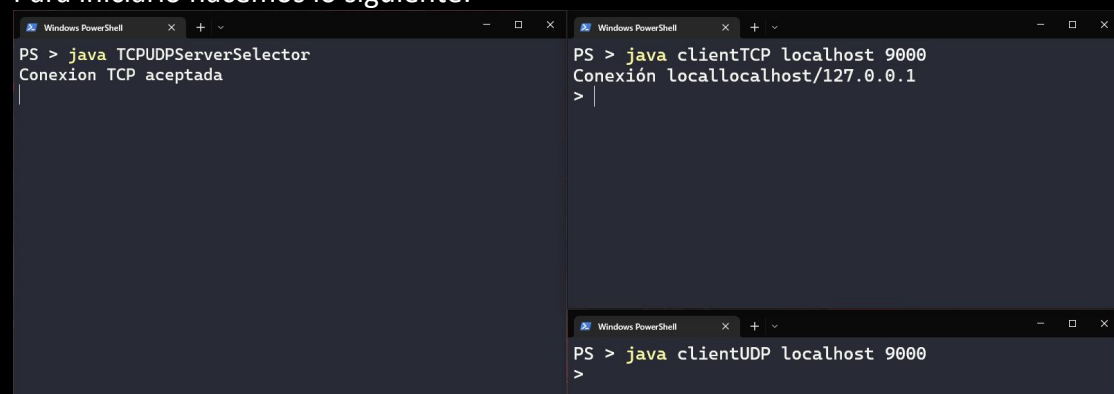
Tras esto, comprobamos si la key es de lectura, es decir, si hemos recibido un mensaje. Con lo cual creamos un DatagramChannel y un ClientRecord con los que hay asignados a la key. Limpiamos el buffer por si hubiera basura y leemos el mensaje recibido al basura, ademas, a la vez guardamos la direccion del cliente. Si esta direccion no es nula, significa que hemos recibido un mensaje y podemos pasar a enviarlo cambiando la key a tipo escritura.

```
//Envio el mensaje de vuelta
if(key.isWritable()){
    DatagramChannel channel = (DatagramChannel) key.channel();
    ClientRecord cr = (ClientRecord) key.attachment();
    cr.buffer.flip();
    System.out.println("Mensaje UDP: "+new String(cr.buffer.array()));
    int bs = channel.send(cr.buffer, cr.clientAddress);
    if(bs!=0){
        key.interestOps(SelectionKey.OP_READ);
    }
}
```

Creamos un DatagramChannel y un ClientRecord como en el caso anterior. Usamos el metodo “flip” como para tcp y enviamos el mensaje. Junto a esto guardamos la cantidad de bytes enviados, si este es distinto de 0, cambiamos el tipo de key a lectura. Por ultimo, usamos el metodo “compact”.

Los clientes que voy a usar para comprobar ya han sido explicados en los apartados pertinentes [UDP](#) y [TCP](#).

Vamos a ver como funciona nuestro programa.
Para iniciarlo hacemos lo siguiente:



The image shows three separate Windows PowerShell terminal windows. The top-left window runs 'java TCPUDPServerSelector' and outputs 'Conexion TCP aceptada'. The top-right window runs 'java clientTCP localhost 9000' and outputs 'Conexion locallocalhost/127.0.0.1'. The bottom window runs 'java clientUDP localhost 9000' and shows a prompt character '>'.

A la izquierda tenemos el servidor, el cual inicia sin ningun mensaje.
Tras esto inicie el clienteTCP, el cual al conectar hace que se muestre un mensaje en el servidor.

Y por ultimo el cliente UDP abajo a la derecha.

Si enviamos mensajes desde los clientes vemos lo siguiente:


```
PS > java TCPUDPServerSelector
Conexion TCP aceptada
Mensaje TCP: hola
Mensaje UDP: hola
|
```

```
PS > java clientTCP localhost 9000
Conexión locallocalhost/127.0.0.1
> hola
echo: hola
>
```

```
Windows PowerShell
PS > java clientUDP localhost 9000
> hola
EC0:hola
> |
```

Se puede ver que podemos enviar los mensajes desde ambos cliente a la vez sin problemas. Además de poder ver que nos envía cada uno de los clientes.

Si cerramos la conexión desde el cliente vemos lo siguiente:

```
PS > java TCPUDPServerSelector
Conexion TCP aceptada
Mensaje TCP: hola
Mensaje UDP: hola
Conexion TCP cerrada
|
```

```
PS > java clientTCP localhost 9000
Conexión locallocalhost/127.0.0.1
> hola
echo: hola
> .
PS > |
```

Con lo que sabemos que el programa elimina el canal asignado al cliente.