# Tecnologías del Sector Financiero


*Práctica Lección 2*

# Mensajería de Baja Latencia I

## CONCEPTOS BÁSICOS

*PRÁCTICAS*

Universidad Carlos III de Madrid

Comunytek
CONSULTING & SOLUTIONS

# Contenido

# 1. Introducción

El proyecto de práctica está escrito en Java con Maven para resolver las dependencias, lo podéis abrir en cualquier IDE de programación Java (Eclipse, IntelliJ, Netbeans) o utilizar la línea de comandos y un editor de texto si lo preferís.

El objetivo de la práctica consiste tanto en resolver los ejercicios como en analizar y comprender los resultados obtenidos.

Podéis encontrar el código de todas las prácticas en GitHub:
https://github.com/cnebrera/MasterUC3Practices

Se necesita tener instalado Java8 para poder trabajar con ellas.

En estas prácticas vamos a tratar mensajería a "bajo nivel" manejando nosotros los sockets y los flujos binarios de datos. Normalmente se suele trabajar con librerías de mensajería que intentan simplificar y ocultar los detalles de bajo nivel, pero es interesante conocer estos últimos para poder entender y depurar problemas que podemos encontrar utilizando estas librerías.

Una vez hayamos realizado algunas pruebas tanto con sockets TCP como multicast procederemos a realizar un pequeño análisis del tráfico de red en cada uno de los casos con Wireshark.

**Entregar el código de la práctica y las capturas que se pidan, así como respuestas a las preguntas que se planteen en cada uno de los ejercicios.**

# 2. Práctica 1: Creación de comunicaciones TCP

En esta práctica crearemos un canal de comunicación simple TCP en Java, probando el envío y recepción y la serialización de mensajes. Cada mensaje representa un precio de mercado, con su instrumento, cantidad y precio.

Las comunicaciones TCP como se explicó en la clase presencial son punto a punto entre dos nodos. Al iniciar la comunicación se establece una sesión entre ellos que se mantiene mientras uno de ellos permanezca conectado.

En esta comunicación uno de los nodos siempre actúa como "servidor" y el otro como "cliente". El servidor debe estar arrancado para que el cliente pueda conectarse a el e iniciar el proceso de comunicación y establecimiento de la sesión. Una vez está lista la sesión la comunicación TCP puede ser unidireccional o bidireccional.

TCP convierte las comunicaciones en un flujo continuo de binario, un "stream" de datos. Para poder trabajar con mensajes individuales debemos ser nosotros los que hagamos el particionado de los mismos al enviar y reconstrucción al recibir.

## a. Mensajes de tamaño fijo

El esqueleto de la práctica lo podéis encontrar en las clases TCPFixSizeClient.java y TCPFixSizeServer.java. El servidor enviará un mensaje de tamaño fijo cada segundo y el cliente lo recibirá y pintará en pantalla.

Las clases de utilidad FixSizeMessage.java y VariableSizeMessage.java representan un mensaje que al serializar a binario tiene tamaño fijo y uno que no lo tiene respectivamente.

Vamos a realizar el proceso mas sencillo que es enviar mensajes de tamaño fijo. Al ser fijo el ensamblado es mucho mas sencillo.

En cada una de las clases tenéis una serie de TODO numerados que será lo que vayamos rellenando en la práctica.

**Creando el servidor**

TODO 1

Lo primero es crear el servidor, para ello utilizaremos la clase ServerSocket de java.net, utilizaremos el constructor con un único argumento dándole el puerto, por ejemplo 6789. Esta clase escucha conexiones de cliente, creando un socket específico de comunicación por cliente.

```
final ServerSocket welcomeSocket = new ServerSocket(6789);
```

TODO 2

El siguiente paso es aceptar una conexión de cliente. Esta llamada se quedara “bloqueada” hasta que algún cliente intente conectarse. Es práctica habitual rodearlo de un “while(true)” y procesar cada socket en un hilo aparte. Esta técnica la utilizan los servidores web para poder procesar multiples peticiones de cliente. Por simplicidad únicamente aceptaremos una conexión.

```
final Socket connectionSocket = welcomeSocket.accept();
```

TODO 3

Una vez tenemos la conexión llamamos al método que va a enviar mensajes.

```
sendMessagesToClient(connectionSocket);
```

Si seguís el código podéis ver como del socket de cliente se obtiene el “outPutStream” es decir el stream de salida de datos para poder escribir en el y enviar información al cliente.

Después se crea un buffer donde se almacenará el mensaje ya serializado en binario para enviarlo, como tienen tamaño fijo, hacemos el buffer del mismo tamaño que el mensaje.

Generamos un mensaje aleatorio y lo convertimos a binario, escribiendo el contenido en el buffer. Si entráis en la clase FixSizeMessage veréis que el código es muy sencillo y se apoya en ByteBuffer para serializar.

TODO 4

Escribimos el mensaje binario ya serializado en el stream de salida.

```
outputStream.write(sendBuffer.array());
```

TODO 5

Si no hacemos nada mas, el mensaje no tiene por que enviarse inmediatamente, el protocolo TCP va a intentar meter tanta información como pueda en un mismo Datagrama TCP para ser lo mas eficiente posible. Como estamos interesados en la latencia, vamos a obligarle a mandar lo que tenga en el stream usando flush.

```
outputStream.flush();
```

Pintamos que hemos mandado un mensaje y esperamos 1 segundo antes de mandar el siguiente.

**Creando el cliente**

El código del cliente es mas sencillo, ya que se conecta directamente con el servidor, no necesita aceptar conexiones.

TODO 1

Creamos el socket de cliente, usando la IP y puerto donde se encuentra el servidor. En este caso de IP usaremos localhost que es la dirección local de loopback.

```
final Socket clientSocket = new Socket("localhost", 6789);
```

TODO 2

Igual que en el servidor obtuvimos el stream de salida, en este caso vamos a obtener el de entrada para recibir mensajes. Los sockets TCP son bidireccionales, aunque para la práctica por simplicidad únicamente utilizaremos una dirección.

```
final InputStream inputStream = clientSocket.getInputStream();
```

TODO 3

Llamamos al método para leer el siguiente mensaje usando el stream de entrada.

```
readMessage(inputStream);
```

En el método de lectura creamos un buffer de bytes para escribir el mensaje binario que viene del socket

TODO 4

Como hemos dicho al principio los streams me dan un flujo continuo de bytes y no sabemos cuantos hay realmente almacenados en el buffer del socket en cada momento pendientes de ser leídos. Dado que los mensajes son de tamaño fijo, basta con preguntar al stream cuantos tiene disponible y esperar a que haya un mensaje completo.

```
while(inputStream.available() < msgBytes.length);
```

TODO 5

Una vez tenemos suficientes bytes en el buffer podemos leerlos.

```
inputStream.read(msgBytes);
```

Por último parseamos el mensaje a partir del contenido binario y ya lo tenemos. Dado que tiene implementado el método toString() podemos pintarlo en pantalla.

**Ejecutando**

Si ejecutáis únicamente el cliente veréis que da una excepción “ConnectionRefused” ya que no encuentra al servidor en la IP y puerto indicada.

Adjuntar la captura de la ejecución al resultado de la práctica.

Hagámoslo ahora en el orden correcto, primero el servidor. Veremos que se queda esperando a que llegue algún cliente. Si ejecutamos ahora el cliente veremos como el servidor escribe en el socket de cliente y el cliente los recibe, 1 mensaje por segundo hasta que no los paremos.

Adjuntar la captura de la ejecución al resultado de la práctica.

Si ejecutáis un segundo cliente, este se no se rompe pero tampoco recibe ningún mensaje. Se queda esperando que el servidor le atienda, como en el código que hemos hecho el servidor únicamente atiende la primera conexión que recibe, no llegan mensajes al segundo cliente.

### b. Mensajes de tamaño variable

El esqueleto de la práctica lo podéis encontrar en las clases TCPVarSizeClient.java y TCPVarSizeServer.java. El servidor enviará un mensaje de tamaño variable cada segundo y el cliente lo recibirá y pintará en pantalla.

Esta vez mandaremos mensajes de tipo VariableSizeMessage.

En este caso el ensamblado no es tan sencillo, debemos saber cuanto ocupa el mensaje para poder esperar a tener suficientes bytes en el stream. Para ello vamos a incluir en cada mensaje que enviamos 4 bytes con el tamaño del mensaje.

En cada una de las clases tenéis una serie de TODO numerados que será lo que vayamos rellenando en la práctica.

**Creando el servidor**

TODO 1, TODO 2, TODO 3

Exactamente igual que en el ejercicio anterior.

Obtenemos el output stream como hicimos anteriormente y creamos un buffer para ir almacenando la cabecera que vamos a escribir. La cabecera contendrá el tamaño del resto del mensaje en bytes.

Dentro del bucle lo primero es limpiar el buffer con la cabecera. Después creamos un mensaje aleatorio, como podéis ver en el parámetro del método vamos a darle un valor para crear mensajes mas grandes o mas pequeños. Comencemos por uno pequeño dándole un valor de 8 como está en el código.

Convertimos el mensaje a binario y guardamos en el buffer de la cabecera el tamaño del mensaje.

TODO 4

Escribimos la cabecera en el stream del socket.

```
outputStream.write(headerBuffer.array());
```

TODO 5

Escribimos el resto del mensaje en el stream del socket.

```
outputStream.write(binaryMessage.array(), 0, binaryMessage.limit());
```

TODO 6

Igual que en la práctica anterior llamamos a flush para forzar el envío sin batching.

Pintamos que está enviado y esperamos antes de enviar el siguiente.

**Creando el cliente**

TODO 1, TODO 2, TODO 3

Exactamente igual que en el ejercicio anterior.

TODO 4

Igual que hicimos en el ejercicio anterior esperando al mensaje completo, esta vez primero esperamos los 4 bytes de la cabecera para saber cuanto ocupa el mensaje realmente.

```
while(inputStream.available() < 4);
```

TODO 5

Leemos la cabecera con el tamaño del mensaje.

```
inputStream.read(header);
```

Deserializamos los bytes leídos y ya tenemos el tamaño, con el tamaño podemos crear un buffer para almacenar el resto del mensaje.

TODO 6

Igual que esperamos los 4 bytes de la cabecera, esta vez esperamos al resto del mensaje dado que ya sabemos cuanto va a ocupar.

```
while(inputStream.available() < msgBytes.length);
```

TODO 7

Leemos los bytes del mensaje.

```
inputStream.read(msgBytes);
```

Igual que en el ejercicio anterior deserializamos el mensaje y lo pintamos.

**Ejecutando**

Ejecutad los dos y a adjuntar la captura de la ejecución al resultado de la práctica.

### c. Mensajes de tamaño variable grandes

Usando el mismo código de la práctica anterior, incrementad el número que se da al método:

*VariableSizeMessage.generateRandomMsg()*

En el ejercicio anterior usamos un 8, creando un mensaje pequeño. Si lo incrementáis a 80000 y volvéis a ejecutar veréis que el servidor envía el primer mensaje y se bloquea justo antes de enviar el segundo. El cliente recibe la cabecera con el tamaño del mensaje, pero se queda en bucle infinito esperando que haya suficientes bytes en el stream.

Mandar una captura de pantalla de la ejecución

**¿Qué está pasando?**

Lo que sucede en el cliente es que el mensaje que envía el servidor es mas grande que la capacidad del buffer intermedio del socket, esto da lugar que nunca haya suficientes bytes disponibles para que los lea el cliente en el stream. Luego veremos como resolverlo.

El servidor también tiene problemas y se bloquea. Recordad que TCP se adapta a la velocidad del cliente, si el cliente no está leyendo datos por defecto va a bloquear el servidor hasta que el cliente esté listo para seguir recibiendo. Como en este caso el cliente se ha "colgado" esperando, el servidor también va a bloquear.

Los mensajes de control siguen fluyendo y la conexión no se rompe, pero no hay envío de mensajes de datos útiles.

Vamos a corregirlo leyendo desde el cliente en pequeños bloques en lugar de esperar a todo el mensaje.

**Creando el cliente**

El cliente con soporte para mensajes grandes está en la clase TCPVarSizeClientBigMsgs.java

TODO1, TODO 2, TODO 3, TODO 4, TODO 5

Exactamente igual que en el ejercicio anterior.

Al igual que antes vamos a crear un array de bytes para almacenar el mensaje completo. Esta vez como leeremos en pequeños bloques, vamos a llevar la cuenta de cuantos bytes llevamos leídos y en nuestro bucle seguiremos leyendo hasta que los tengamos todos.

Calculamos el número de bytes que vamos a leer, como vamos a usar bloques de 128 bytes, será 128 o menos en caso de que quede poco por leer del mensaje.

TODO 6

Leemos el siguiente bloque de bytes y actualizamos los byes leídos.

```
numBytesRead += inputStream.read(msgBytes, numBytesRead, bytesToRead);
```

Ya únicamente falta deserializar el mensaje y pintarlo igual que hicimos antes.

**Ejecutando**

Esta vez deberíamos ser capaces de leer y enviar los mensajes sin problemas.

Adjuntar una captura de la ejecución

## 3. Práctica 2: Creación de comunicaciones Multicast

En esta práctica crearemos un canal de comunicación simple Multicast en Java, probando el envío y recepción y la serialización de mensajes. Cada mensaje representa un precio de mercado, con su instrumento, cantidad y precio.

A diferencia de TCP, en Multicast las comunicaciones son UDP, es decir no hay sesión, ni control de flujo, ni de orden, etc.

El cliente de la comunicación necesita "unirse" a un grupo multicast, es decir registrar que quiere escuchar mensajes enviados a ese grupo. A efectos prácticos un grupo no es mas que la dirección multicast a la que nos subscribimos.

El servidor es mucho mas sencillo, únicamente tiene que enviar a esa dirección multicast, ni siquiera necesita mantener una conexión abierta.

A diferencia de TCP no tenemos un "stream" continuo de datos binarios donde el protocolo envía ventanas de distintos tamaños y se encarga de meter mas o menos mensajes en un mismo datagrama para optimizar la transferencia.

En este caso dado que no hay control ninguno, somos nosotros los que tenemos que decidir cuanta información va en un datagrama y el datagrama se envía completo. Lo mismo sucede en recepción donde llega el datagrama completo que se envía.

Veamos un ejemplo sencillo con mensajes de tamaño variable pequeños.

### a. Mensajes de tamaño variable

El esqueleto de la práctica lo podéis encontrar en las clases McastClient.java y McastServer.java

En cada una de las clases tenéis una serie de TODO numerados que será lo que vayamos rellenando en la práctica.

**Servidor**

Definimos la IP multicast y el puerto que vamos a utilizar. Y creamos un objeto java de tipo InetAddr para la dirección.

TODO 1

Creamos un objeto de tipo DatagramSocket que nos va a permitir enviar datagramas a la IP y puerto que le digamos. En el constructor no indicamos nada, recordad que no tenemos conexión o sesión fija con lo que no hay que ligar el socket a ninguna de ellas.

```
final DatagramSocket serverSocket = new DatagramSocket()
```

TODO 2

Llamamos a sendMessage con el socket que hemos creado y la dirección a la que vamos a enviar para no tener que volver a crearla en cada mensaje.

```
sendMessage(addr, serverSocket);
```

Igual que en las prácticas anteriores generamos un mensaje aleatorio y lo convertimos a binario.

TODO 3

Creamos un objeto de tipo DatagramPacket con el mensaje binario que vamos a enviar.

```
DatagramPacket msgPacket = new DatagramPacket(binaryMessage.array(), binaryMessage.limit(), addr,
PORT);
```

Si os fijáis es aquí donde indicamos la dirección y el puerto, es decir podemos utilizar el mismo objeto socket para enviar mensajes a múltiples IPs multicast.

TODO4

Enviamos el datagrama.

```
serverSocket.send(msgPacket);
```

Esperamos un poco antes de enviar de nuevo.

**Cliente**

Igual que en el servidor creamos un objeto InetAddress con la dirección a la que vamos a subscribirnos. También crearemos un buffer que reutilizaremos para leer el binario de los mensajes que recibamos.

TODO 1

Creamos el socket multicast donde vamos a recibir, indicando cual es el puerto.

```
final MulticastSocket clientSocket = new MulticastSocket(PORT)
```

TODO 2

Nos subscribimos a una dirección multicast, es decir nos unimos a un grupo multicast. Podemos utilizar el mismo socket para recibir de múltiples grupos multicast si queremos.

```
clientSocket.joinGroup(address);
```

TODO 3

Con el buffer y el socket, vamos a entrar en un bucle de recepción de mensajes.

```
receiveMessage(receiveBuffer, clientSocket);
```

Limpiamos el buffer de recepción para reutilizarlo.

TODO 4

Creamos el datagrama que vamos a rellenar con el próximo mensaje que recibamos, vamos a utilizar el buffer que creamos al principio para que el binario se almacene en el.

```
final DatagramPacket msgPacket = new DatagramPacket(receiveBuffer.array(), receiveBuffer.capacity());
```

TODO 5

Llamamos al socket para recibir un mensaje y rellenar el datagrama que hemos creado.

```
clientSocket.receive(msgPacket);
```

TODO 6

Como los mensajes son de tamaño variable, antes de leer el binario necesitamos saber cuantos bytes han llegado realmente, para ello establecemos el limite del buffer con el tamaño de bytes recibidos en le datagrama.

```
receiveBuffer.limit(msgPacket.getLength());
```

Ya podemos deserializar el mensaje binario e imprimirlo en pantalla.

**Ejecución**

A diferencia de TCP esta vez no importa el orden de ejecución. Si lanzamos primero el cliente no recibirá nada, pero no se romperá ya que no necesita mantener una sesión con el servidor, simplemente se quedará escuchando a que alguien envíe algo a la dirección multicast.

Del mismo modo el servidor no espera a que haya un cliente conectado, envía mensajes a la dirección multicast y si no hay nadie escuchando simplemente se pierden.

Las comunicaciones multicat no son punto a punto, si el servidor envía el mensaje llegará a cualquier cliente que esté escuchando en esa dirección multicast. Para probarlo no tenéis mas que lanzar múltiples instancias del cliente y ver que todas reciben los mensajes.

Mandar capturas de pantalla de las distintas ejecuciones.

### b. Mensajes grandes

Al igual que en la práctica de TCP, podemos incrementar el tamaño del mensaje que envía el servidor. Si lo incrementamos a 80000, veremos que el servidor se rompe al intentar enviar.

Mandar una captura de la ejecución

El problema es que a diferencia de TCP donde tenemos un “flujo de bytes”, ahora nosotros controlamos los datagramas, y estos tienen un tamaño máximo.

Para poder solucionar esto tenemos que partir los mensajes que enviamos si son demasiado grandes y “reensamblarlos” en el cliente. El concepto es sencillo, pero recordemos que en Multicast los mensajes se pueden perder, llegar desordenados, etc. Esto complica mucho las cosas y obliga a mantener números de secuencia de los datagramas y otros mecanismos de control para reensamblar mensajes y recuperar el orden.

No vamos a solucionar este problema, en este caso el objetivo es ilustrar que la solución no es trivial y para eso los distintos fabricantes han creado sus protocolos de “reliable multicast” o utilizan el estándar PGM y tienen sus propios mecanismos de particionamiento y ensamblado de mensajes..

## 4. Práctica 3: Analizando con Wireshark

En el material adicional vimos como utilizar Wireshark para analizar el flujo de red. Vamos a hacer un pequeño análisis sobre algunas de las prácticas anteriores.

### a. Flujo TCP normal

Vamos a utilizar la práctica de mensajería TCP de tamaño fijo. Como vimos en el material adicional abriremos wireshark y esta vez vamos a capturar el interfaz de loopback que es

donde estamos mandado los mensajes en la práctica. El interfaz loopback es un interfaz virtual para llamadas internas que nunca sale a la red real. Para evitar ruido de otras llamadas por DNS y otros sistemas internos vamos a filtrar por el puerto que estamos utilizando y por protocolo introduciendo en el campo de filtrado:

*tcp && tcp.port==6789*

Primero abrimos únicamente el cliente, va a fallar porque no hay servidor. Deberíais ver dos entradas en la lista. Una primera del cliente en un puerto aleatorio llamando al puerto del servidor de tipo SYNC, y una respuesta de rechazo por parte del SO.

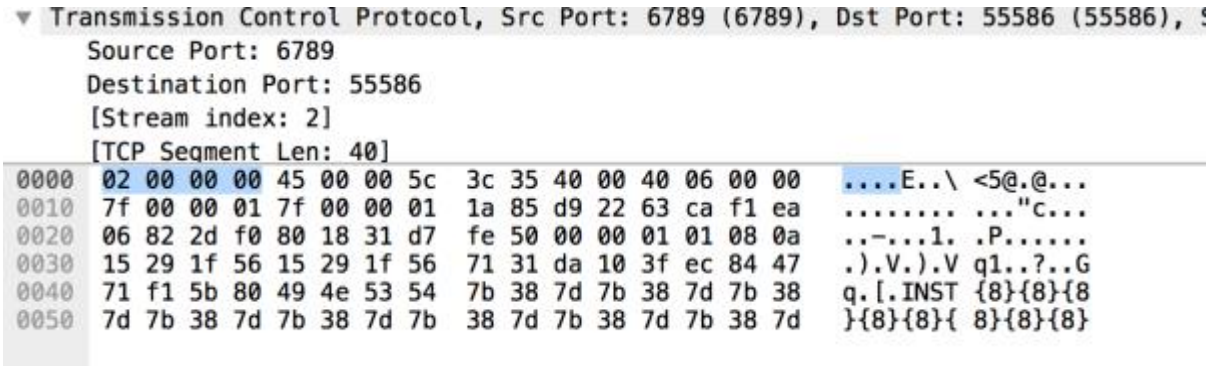
Adjuntar la captura

Si repetimos con el servidor arrancado veremos que además de los mensajes de sincronización iniciales ahora además tenemos mensajes de datos, ACK de control y mensajes de actualización de tamaño de ventana.

Adjuntar la captura

Si abrimos uno de los mensajes de datos de tipo PSH y vamos al contenido veremos que es puro binario y poco podemos discernir. Si quisiéramos analizarlo tendríamos que poder deserializarlo de la misma forma que hicimos en código. Es posible escribir un plugin de wireshark para hacerlo pero no vamos a entrar en ello.

Vamos a repetir lo mismo con los mensajes de tamaño variable, esta vez pequeños de tamaño 8 como estaba en la práctica original. Si volvemos a ir a los mensajes de datos, veremos que esta vez al estar enviando Strings en UTF-8 wireshark es capaz de decodificar parte del mensaje y debemos ver algo así:




Podéis ver el INST {8}{8}… que es la cadena que estamos mandando como instrumento.

Adjuntar una captura de la pantalla completa de wireshark con el análisis.

### b. Flujo TCP bloqueado

Esta vez vamos a repetir los mensajes de tamaño variable grandes que dejaban bloqueado el servidor y el cliente.

Si lo dejáis correr un poco veréis mensajes del tipo TCP Window FULL del servidor al cliente y TCP Zero Window del cliente al servidor. La comunicación de control no se detiene, pero ambas partes están avisando que hay un problema, el servidor que tiene la ventana llena y el cliente que no tiene espacio para recibir nada mas.

Adjuntar una captura de la pantalla completa de wireshark con el análisis.

### c. Flujo Multicast

Vamos a repetir el análisis pero con un flujo de mensajería multicast utilizando la práctica anterior de multicast.

Esta vez la elección de la interfaz de red no es tan sencilla, si os fijáis en el código no estamos usando IPs multicast, pero estas IPs no tienen nada que ver con IPs locales lo que dificulta la elección de que interfaz de red usar. Es posible asignar una interfaz de red para que los mensajes utilicen la interfaz que nosotros queramos.

Si no estáis conectados a internet utilizará la interfaz de loopback, si estáis conectados utilizará la interfaz normal de red, por ejemplo la Wifi. Si no estáis seguros basta con utilizar el filtro sobre cada una de las interfaces y ver donde llegan los mensajes.

El filtro en este caso será UDP (Multicast es UDP) y el puerto:

*udp && udp.port==8888*

Esta vez veréis que no hay mensajes de sesión ni control, únicamente un mensaje por Datagrama con el binario que se está enviando.

Si hacéis igual que en el análisis de TCP podéis ver el contenido del mensaje y buscar la cadena INT con el instrumento.

Adjuntar la captura del análisis