Sockets UDP y TCP

PRACTICA III

IGNACIO ORTEGA Y ROCIO RUIZ

ÍNDICE

[UDP 2](#_Toc39257682)

[SERVIDOR UDP 2](#_Toc39257683)

[CLIENTE UDP 3](#_Toc39257684)

[TCP 5](#_Toc39257685)

[SERVIDOR TCP 5](#_Toc39257686)

[CLIENTE TCP 6](#_Toc39257687)

[CONCLUSIONES TCP VS UDP 7](#_Toc39257688)

[CONCLUSIONES LATENCIAS TCP VS UDP 7](#_Toc39257689)

# UDP

UDP son las siglas de Protocolo de Datagrama de Usuario. Es un protocolo de nivel de transporte, capa 4 del modelo OSI. Está basado en el intercambio de datagramas a través de la red sin que se haya establecido previamente una conexión, ya que el propio datagrama incorpora suficiente información de direccionamiento en su cabecera.

UDP/IP proporciona muy pocos servicios de recuperación de errores, ofreciendo en su lugar una manera directa de enviar y recibir datagramas a través una red IP. Se utiliza sobre todo cuando la velocidad es un factor importante en la transmisión de la información, por ejemplo, audio y video en tiempo real.

## SERVIDOR UDP

Vamos a describir el proceso de desarrollo de un servidor UDP en Java. Las librerías de java como java.io nos facilita muchas herramientas para la gestión de Sockets y transmisión de mensajes.

Comenzamos creando un socket que realice la función de escucha en el sistema operativo. Dentro de un try-catch (el cual gestionará las excepciones sobre mensajes), inicializamos el socket al cual asignaremos un puerto para la recepción de mensajes.

Creamos un *array* de bytes para contener el mensaje enviado por el cliente. Definimos un puerto para el cliente definimos una dirección IP para el cliente.

En un bucle *while(true)* se inicializa el datagrama (*request*) que contendrá el mensaje enviado por el cliente.

Se utiliza la primitiva receive del sistema operativo para actuar de escucha y pasarle el mensaje al servidor una vez sea enviado por el cliente.

Obtenemos el puerto del cliente, donde se establecerá la conexión de vuelta entre el servidor y el cliente, y la IP del cliente para posteriormente poder mandar un mensaje a la misma.

Creamos el datagrama(*reply*) que va a enviar el servidor al cliente, indicando cual es la longitud de este mensaje, la dirección IP destino y el puerto destino.

Utilizando la primitiva *send* del sistema, el servidor envía el datagrama al cliente.

Al tratarse de UDP, no cerramos la conexión.

Captura de pantalla de un celular con letras

Descripción generada automáticamente

## CLIENTE UDP

Preparamos el canal generando un *bufferedReader* para poder leer por pantalla lo que vamos a mandar al servidor echo. Creamos el socket de envío, que inicialmente no tiene asignado un puerto destino ni una IP propia. Creamos la dirección IP que será la propia del cliente, desde donde se envía el socket, para que así el servidor sea capaz de enviarnos la respuesta a nuestro equipo. Inicializamos un array de bytes, con el numero de bytes que vamos a enviar el mensaje. El mensaje a mandar en este caso, se lo mandamos al main como argumento, (args[0]). El mensaje que se vaya a enviar por el socket de envío, tendrá la cadena del mensaje convertida en bytes

Creamos una variable que será la correspondiente al puerto destino del servidor, que es el punto donde se conectan el cliente con el servidor, y el cual actúa como si fuera un fichero, donde se escribe el paquete. Creamos un array de bytes de un tamaño de 1000 que contendrá el mensaje en bytes enviado de vuelta por el servidor.

Inicializamos el proceso de comunicación entre el cliente y el servidor. Dentro de un try-catch (el cual gestionará las excepciones sobre mensajes), inicializamos el socket de envío. Lo asignamos a la dirección IP la que tiene el equipo del servidor. Creamos un vector *times* donde se van a almacenar las latencias de cada petición para poder hacer una media de la latencia en UDP.

Creamos el *bind*, que será el encargado de conectar la IP del cliente con el puerto del servidor. Dentro de este *bind* también se incluye el mensaje(en bytes) que quiere mandar el cliente al servidor. Será el encargado de conectar el TSAP(Identificador de transporte(Puerto)) con el NSAP(Identificador de red) que es una dirección IP ya que estamos utilizando el protocolo IP.

Inicializamos un bucle para realizar muchas peticiones y empezamos a medir el tiempo que transcurre desde aquí hasta que lo reciba de vuelta para calcular el tiempo total que tarda en hacerse la comunicación. El cliente utiliza la primitiva de send del sistema operativo, mediante la cual envía al servidor su socket de envío mandándole el *bind* creado anteriormente, para que así el *listener* del servidor reciba esto.

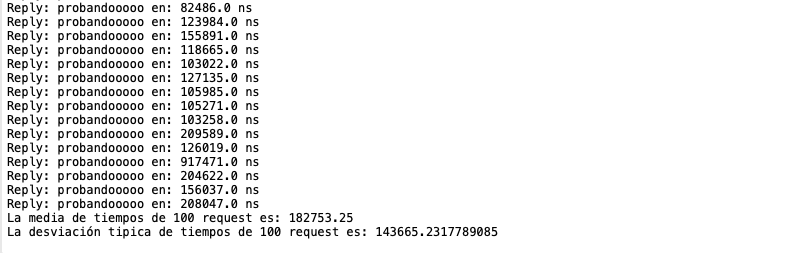
Inicializamos un array de bytes que será el que contenga el mensaje enviado por el servidor. Creamos un datagrama que va a contener lo que mande de vuelta el servidor. Creamos el listener, que va a estar a la espera hasta que le llegue un mensaje de vuelta del servidor. Cuando le llegue, el array creado para el mensaje, ya contendrá el mensaje(en bytes) enviado por el cliente.

Guardamos el tiempo de latencia una vez que llega el mensaje de vuelta al cliente. La primera vez que enviamos un mensaje tarda más que las demás, ya que tiene que obtener la dirección IP y el puerto destino al que va enviado.

Después de las peticiones asignadas, se llama a una función de estadísticas que calcula la media y la desviación típica.

Una captura de pantalla de un celular con letras

Descripción generada automáticamente



Aquí se muestra parte de los resultados y las estadísticas finales.

# TCP

TCP son las siglas de Protocolo de Control de Transmisión. Un sistema de protocolos que hacen posibles múltiples.

Este, permite a dos anfitriones establecer una conexión e intercambiar datos. El TCP garantiza la entrega de datos, es decir, que los datos no se pierdan durante la transmisión y también garantiza que los paquetes sean entregados en el mismo orden en el cual fueron enviados.

## SERVIDOR TCP

Vamos a describir el proceso de desarrollo de un servidor TCP en Java. Las librerías de java como java.io nos facilita muchas herramientas para la gestión de Sockets y transmisión de mensajes.

Declaramos un bloque try-catch para la gestión de errores. Instanciamos un ServerSocket con el puerto que va a estar disponible para la conexión con los clientes. Creamos un socket que servirá para que el listen pueda seguir escuchando mientras que el servidor realiza una operación con el socket que reciba el listen.

Creamos un bucle while(true) Lo hacemos infinito, ya que el servidor siempre va a estar escuchando los posibles mensajes que les lleguen desde los diferentes clientes.

La función *accept* que nos permitirá aceptar conexiones de clientes, haciendo así la función principal del TCP que lo diferencia del UDP. Se realiza la conexión con este nuevo socket.  
Mientras tanto, el socket *listen* seguirá escuchando a otros posibles clientes. Por esta razón creamos otro socket.

En la conexión declaramos e instanciamos el objeto *DataInputStream* y *DataOutputStream* para recibir y enviar los datos del cliente. Así cumple la función echo.

Captura de pantalla de un celular

Descripción generada automáticamente

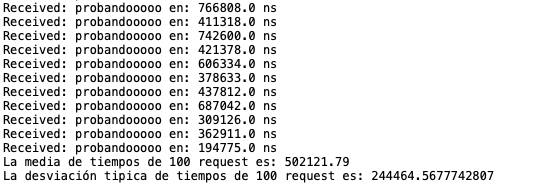
## CLIENTE TCP

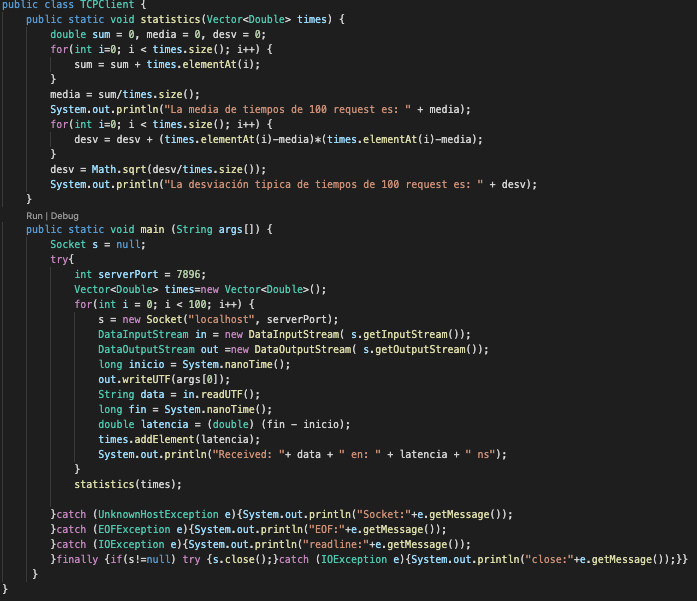
Creamos el socket mediante el cual nos vamos a comunicar. Inicializamos un bucle try-catch que se encargará de gestionar las excepciones del socket. Creamos un vector *times* donde se van a almacenar las latencias de cada petición para poder hacer una media de la latencia en UDP. Inicializamos un bucle para realizar muchas peticiones y empezamos a medir el tiempo que transcurre desde aquí hasta que lo reciba de vuelta para calcular el tiempo total que tarda en hacerse la comunicación. Las comunicaciones en TCP tendrán mayor latencia ya que en una conexión solo se puede mandar un mensaje, y hay que crear una conexión por cada request.

Inicializamos el socket de comunicación con la dirección IP del servidor con la que quiere el cliente realizar la conexión, dicha IP es el NSAP, identificador de red, (es una dirección IP ya que utilizamos protocolo IP)y el puerto del servidor que sería el TSAP, identificador de transporte, que es el punto de conexión entre el servidor y el cliente en el protocolo TCP. Declaramos un objeto *DataInputReader* que nos permitirá recibir la información del servidor en el cliente. Y

Enviamos el mensaje codificado en UTF al servidor. Definimos una variable data que va a contener el mensaje enviado de vuelta por el servidor. Como en este caso es *echo*, el mensaje será el mismo que mandamos.

Una vez llega el mensaje de respuesta, paramos el tiempo. Como es un servicio orientado a la conexión, la latencia será mayor que la de un servicio UDP, ya que es necesario que el servidor confirme que el cliente puede mandar el mensaje y tiene que ser aceptado. También tiene que establecer la conexión cada vez que lo envíe, por lo que el tiempo de latencia aumenta considerablemente.

Cerramos la conexión, ya que es un servicio orientado a la conexión. Imprimimos la latencia de cada envío de mensaje y la media y desviación típica de ambas.



# ASIGNACIÓN DE TRABAJADORES

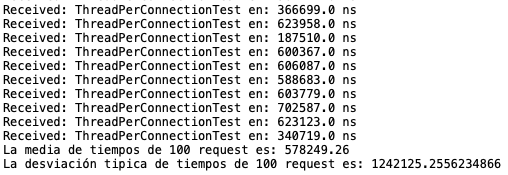
En el código de Servidor UDP anterior, está implementado un Thread per Request por la clase Connection.

Hemos implementado un **Thread-per-Connection** creando una clase que creará una conexión por cada petición que recibía.

Esto, aumenta el rendimiento de la comunicación, ya que, se podrían establecer muchas conexiones al mismo tiempo.

Lo hemos implementado de esta manera:

Clase ThreadPerConnection: Crea cada socket cliente aceptando la petición, y crea una nueva conexión.



Captura de pantalla de un celular con letras

Descripción generada automáticamente

Para hacer una mejor comparativa, hemos calculado cuando tardan en hacer la conexión TCP con Thread-per-Request y Thread-Per-Connection de 1000 peticiones.

1. Thread-per-Request:



1. Thread-per-Connection:



TCP con Thread-per-Connection es mas rápido que el Thread-per-Request.

Si probamos con mas numero de solicitudes se notaría mas aún la diferencia.

# CONCLUSIONES TCP VS UDP

En el cliente servidor UDP, al ser un servicio no orientado a la conexión, el cliente solo asigna una vez la dirección IP y el puerto al que se va a conectar del servidor. No necesita confirmación de conexión y no hace falta liberar la conexión, con lo cual, una vez que el cliente establezca la primera conexión con el servidor, podrá enviar segmentos con mensajes las veces que quiera. Al no tener que liberar la conexión, cuando le mande un mensaje al servidor, como este sigue ejecutándose, va a utilizar los datos de la primera conexión para comunicarse con este.

Sin embargo, en el cliente servidor TCP, al ser un servicio orientado a la conexión, el cliente tiene que mandarle una petición al servidor para poder mandarle mensajes. Una vez el servidor lo recibe, lo acepta y establece una conexión con el cliente. El cliente le envía el mensaje y el receptor se lo devuelve (si echo). Una vez realizada esta operación, se cierra la conexión entre ellos. Para que el cliente se pueda volver conectar al servidor, va a tener que mandarle otra petición. Tendrá que crearse un socket de envío asignado a un determinado puerto.

# CONCLUSIONES LATENCIAS TCP VS UDP

Para medir la latencia en UDP y TCP, medimos el tiempo que transcurre desde que el cliente envía el mensaje hasta que le llega este mismo desde el servidor.

En UDP, la latencia va a variar entre el primer envío de mensaje y los posteriores, dado que en el primero es cuando se aplica el protocolo para establecer la conexión con la dirección IP y el puerto de conexión del servidor, y en las siguientes no es necesario, porque ya utiliza la información de esa conexión, dado que esta no se fue liberada. Los posteriores envíos de mensaje van a tener una latencia similar pero no igual, y esto es debido a que nuestra tarjeta de red no solo está atendiendo a nuestro programa, sino que también está prestando servicio a otras aplicaciones.

En TCP, la latencia va a ser similar entre cada envío, porque cada vez que envía un mensaje tiene que esperar a que el servidor le mande confirmación y lo acepte, y transmitir el mensaje. El TCP tiene mayor latencia que el UDP dado que el servidor tiene que mandarle confirmación una vez el cliente le solicita conexión para enviar el mensaje y tiene que recibirlo.

El uso de un protocolo u otro dependerá del uso.