INSTITUTO POLITÉCNICO NACIONAL.

ESCUELA SUPERIOR DE CÓMPUTO.

APLICACIONES PARA COMUNICACIONES EN RED.

PRÁCTICA 2.

PROFESOR: AXEL ERNESTO MORENO CERVANTES.

GRUPO 6CM1.

ALUMNA: SÁNCHEZ VALERIANO ALEXANDRA.

FECHA: 13/10/2025

INTRODUCCIÓN.

En la era digital, el consumo de contenido multimedia bajo demanda se ha convertido en una de las aplicaciones de red más predominantes. Desde servicios de video hasta plataformas de música, la capacidad de transmitir grandes volúmenes de datos de manera eficiente y confiable es fundamental. Este tipo de sistemas presenta desafíos únicos que van más allá de las simples interacciones de solicitud-respuesta, como las que se encuentran en el comercio electrónico.

El principal reto en la transmisión de archivos de gran tamaño es asegurar la integridad de los datos sin sacrificar la velocidad. Mientras que protocolos como TCP ofrecen confiabilidad de forma nativa, su estricto control de errores puede introducir latencia, lo cual es indeseable para aplicaciones en tiempo real. Por otro lado, UDP ofrece una comunicación más rápida y ligera, pero a costa de no garantizar la entrega ni el orden de los paquetes.

Con esta motivación, el presente proyecto desarrolla un sistema de streaming de música cliente-servidor que opera sobre UDP. El objetivo es implementar un protocolo de transferencia de datos confiable a nivel de aplicación, utilizando una técnica de ventana deslizante (Sliding Window) para gestionar el flujo de paquetes y la confirmación de su recepción. Este enfoque permite combinar la velocidad de UDP con un mecanismo de fiabilidad personalizado.

Este reporte documenta una aplicación de streaming de música desarrollada en Java. El sistema está compuesto por un servidor, que gestiona y transmite una colección de archivos MP3, y un cliente, que solicita una canción, la recibe en fragmentos y la reproduce. La comunicación se establece a través de DatagramSockets, y la confiabilidad se logra mediante la serialización de objetos de paquete, el secuenciamiento de datos y un sistema de confirmaciones (ACKs) con manejo de timeouts para la retransmisión de paquetes perdidos.

DESARROLLO.

El sistema se articula en torno a tres componentes de software principales que trabajan en conjunto para lograr la transmisión de audio sobre UDP de manera confiable. A continuación, se detalla el funcionamiento técnico de cada uno.

1. **SERVIDOR.JAVA - EL PROVEEDOR DE CONTENIDO.**

La clase **Servidor** actúa como el corazón del sistema, gestionando el catálogo de música y orquestando la transmisión de archivos. Su lógica se puede descomponer en los siguientes pasos:

- **Inicialización y Descubrimiento de Contenido**: Al arrancar, el servidor crea un **DatagramSocket** y lo vincula al puerto **4000** para escuchar las peticiones entrantes. Inmediatamente después, escanea el directorio de escritorio del usuario (**System.getProperty("user.home") + "/Desktop/"**) en busca de archivos con la extensión **.mp3**. Estos archivos se cargan en un arreglo (**File[] mp3s**), conformando el catálogo de canciones que se ofrecerá a los clientes.

- **Protocolo de Comunicación a Nivel de Aplicación**: El servidor opera dentro de un bucle infinito (**while (true)**), esperando datagramas. Su comportamiento depende del mensaje recibido:

    - **Solicitud LISTA**: Si un cliente envía el mensaje "LISTA", el servidor recorre su arreglo de archivos MP3 y construye una cadena de texto. Cada canción se formatea como **"índice:nombre_archivo;"**. Esta cadena se convierte en bytes y se envía de vuelta al cliente, permitiéndole ver el catálogo disponible.

    - **Solicitud de Streaming**: Si recibe un mensaje con el formato **"índice:TAM_PAQUETE:VENTANA"**, el servidor lo interpreta como una solicitud para transmitir una canción específica. Parsea estos tres valores para seleccionar el archivo, determinar el tamaño de cada paquete y configurar el tamaño de la ventana deslizante.

- **Fragmentación y Empaquetado de Datos**: Una vez que se selecciona una canción, el servidor la abre con un **FileInputStream**. El archivo se lee en fragmentos cuyo tamaño máximo es **TAM_PAQUETE**. Cada fragmento de bytes se encapsula en una instancia de la clase **Paquete**. Este objeto no solo contiene los datos del archivo, sino también metadatos vitales: el número de secuencia (**siguiente**), el

número total de paquetes, el nombre del archivo y un booleano (**fin**) que indica si es el último fragmento.

- **Implementación de Ventana Deslizante (Sliding Window)**: Esta es la pieza clave para la fiabilidad.
    - El servidor utiliza dos variables principales: **base** (el número del paquete más antiguo que aún no ha sido confirmado) y **siguiente** (el número del próximo paquete a enviar).
    - Entra en un bucle que envía paquetes mientras **siguiente** esté dentro de la ventana (**siguiente < base + VENTANA**) y no se haya llegado al final del archivo.
    - Para enviar un paquete, serializa el objeto **Paquete** a un arreglo de bytes usando **ObjectOutputStream** y lo envía en un **DatagramPacket**.
    - Después de enviar los paquetes de la ventana, establece un tiempo de espera de 5 segundos (**socket.setSoTimeout(5000)**) y espera un **ACK**.
    - **Si recibe un ACK**: Parsea el número de paquete confirmado y "desliza" la ventana, actualizando **base** a **ackNum + 1**. Esto significa que todos los paquetes hasta **ackNum** fueron recibidos correctamente.
    - **Si ocurre un Timeout**: Si no se recibe un **ACK** a tiempo, se lanza una **SocketTimeoutException**. El servidor asume que los paquetes se perdieron, imprime un mensaje de "Timeout" y resetea **siguiente** al valor de **base** para reenviar toda la ventana de paquetes no confirmados.

2. CLIENTE.JAVA - EL REPRODUCTOR MULTIMEDIA.

El **Cliente** gestiona la interacción con el usuario, solicita las canciones y se encarga de recibir y reconstruir el archivo de audio para su posterior reproducción.

- **Interfaz y Configuración de Usuario**: El cliente opera a través de la consola. Primero envía un mensaje "LISTA" para obtener y mostrar las canciones disponibles. Luego, solicita al usuario que elija el número de la canción, el **tamaño del paquete** (controlando la granularidad de la transferencia) y el **tamaño de la ventana** (determinando cuántos paquetes se pueden enviar antes de requerir una confirmación).

- **Recepción y Deserialización de Paquetes**: El cliente entra en un bucle de recepción (**while (!fin)**).

    1. Crea un buffer de bytes y espera a recibir un **DatagramPacket** del servidor.

    2. Una vez recibido, utiliza un **ByteArrayInputStream** y un **ObjectInputStream** para deserializar el arreglo de bytes y reconstruir el objeto **Paquete** original. Este paso es crucial, ya que recupera tanto los datos del archivo como todos los metadatos de control (número, total, etc.).

- **Ensamblaje Ordenado y Confirmaciones (ACKs)**:

    1. El cliente mantiene una variable **esperado** que lleva la cuenta del número de secuencia del próximo paquete que debe recibir.

    2. Si el número del paquete recibido (**paquete.numero**) coincide con **esperado**, significa que el paquete llegó en el orden correcto. En este caso, los datos del paquete (**paquete.datos**) se escriben en un **FileOutputStream**, reconstruyendo el archivo en el disco local (**"recibido_" + nombreArchivo**). El contador **esperado** se incrementa.

    3. Independientemente de si el paquete estaba en orden o no, el cliente envía un **ACK** acumulativo al servidor. El mensaje es **"ACK " + (esperado - 1)**, confirmando la recepción de todos los paquetes hasta ese número. Esto informa eficientemente al servidor hasta qué punto la transmisión ha sido exitosa.

- **Finalización y Reproducción**: El bucle de recepción termina cuando se recibe un paquete con la bandera **fin** activada. Una vez que el archivo se ha guardado por completo, el cliente utiliza dos librerías externas:

    1. **mp3agic**: Para leer y mostrar los metadatos del archivo MP3 (título, artista, duración, etc.), enriqueciendo la experiencia del usuario.

    2. **JLayer**: Para tomar el archivo **.mp3** recién creado y reproducirlo a través de los altavoces del sistema.

3. PAQUETE.JAVA - EL MODELO DE DATOS.

Esta clase, aunque simple, es fundamental para la arquitectura del sistema. Actúa como un **Data Transfer Object (DTO)**, una estructura de datos diseñada para transportar información entre el cliente y el servidor.

- **Encapsulación de Datos**: Reúne en un solo lugar tanto los **datos** (el fragmento del archivo en **byte[] datos**) como los **metadatos** necesarios para el protocolo de control de flujo (**numero, total, fin, nombre, tamArchivo**). Esto evita tener que enviar información de control y datos de archivo en paquetes separados, simplificando la lógica.

- **Serialización**: La clase implementa la interfaz **java.io.Serializable**. Esta es una interfaz marcadora que le indica a la JVM que los objetos de esta clase pueden ser convertidos en un flujo de bytes. Gracias a esto, el servidor puede tomar un objeto **Paquete** completo, con todos sus atributos, y convertirlo en un **byte[]** para ser enviado en un único **DatagramPacket**. El cliente puede entonces revertir este proceso para reconstruir el objeto idéntico en su lado, garantizando la integridad de la información transmitida.

PRUEBAS.

- **SERVIDOR.**

Practica2 (run) × Practica2 (run) #2 ×

```
run:
Servidor iniciado. Esperando mensajes del cliente...

------------------------------------------------------------
------------ Informaci�n del MP3 recibido: ----------------
 -- Archivo: Life Is Real.mp3
 -- Duraci�n: 2 min 53 seg
 -- Artista: LEO
 -- Álbum: Life Is Real
 -- A�o: null
 -- T�tulo: Life Is Real
------------------------------------------------------------
------------------------------------------------------------
```

Practica2 (run) × Practica2 (run) #2 ×

```
------------------------------------------------------------

Iniciando env�o...
Total paquetes: 56
Ventana: 1 | Tama�o paquete: 50000 bytes


------------------------------------------------------------
------------ METAINFORMACI�N DEL PAQUETE #1 ---------------
 -- Archivo: Life Is Real.mp3
 -- Total paquetes: 56
 -- N� de paquete: 1
 -- Tama�o datos: 50000 bytes
```

Practica2 (run) × Practica2 (run) #2 ×

```
------------------------------------------------------------
------------ METAINFORMACI◆N DEL PAQUETE #1 ---------------
 -- Archivo: Life Is Real.mp3
 -- Total paquetes: 56
 -- N◆ de paquete: 1
 -- Tama◆o datos: 50000 bytes
 -- Tama◆o total archivo: 2780139 bytes
Enviado paquete #1
ACK recibido: 1
------------------------------------------------------------
------------------------------------------------------------
```

Practica2 (run) × Practica2 (run) #2 ×

```
------------ METAINFORMACI◆N DEL PAQUETE #56 ---------------
 -- Archivo: Life Is Real.mp3
 -- Total paquetes: 56
 -- N◆ de paquete: 56
 -- Tama◆o datos: 30139 bytes
 -- Tama◆o total archivo: 2780139 bytes
Enviado paquete #56
ACK recibido: 56
------------------------------------------------------------
------------------------------------------------------------

Transmisi◆n completa de Life Is Real.mp3
```

- **CLIENTE.**

Practica2 (run) × Practica2 (run) #2 ×

```
run:
Canciones disponibles:
[1] Golden Horizon.mp3
[2] In The Changing World.mp3
[3] Life Is Real.mp3
[4] Where Do We Go.mp3

Elige el n◆mero de la canci◆n: 3
Tama◆o de paquete (bytes, ej. 60000): 50000
Tama◆o de ventana (ej. 5): 1

Esperando transmisi◆n del servidor...
```

Practica2 (run) × Practica2 (run) #2 ×

```
--------------- METAINFORMACI◆N RECIBIDA ------------------
 -- Archivo: Life Is Real.mp3
 -- Total paquetes: 56
 -- No. de paquete: 1
 -- Tama◆o datos: 50000 bytes
 -- Tama◆o total archivo: 2780139 bytes
Iniciando recepci◆n de: Life Is Real.mp3
Progreso: 1.79%
------------------------------------------------------------
------------------------------------------------------------

------------------------------------------------------------
```

Practica2 (run) × Practica2 (run) #2 ×

```
------------------------------------------------------------
------------- Informaci◆n del MP3 recibido: ----------------
 -- Archivo: Life Is Real.mp3
 -- Duraci◆n: 2 min 53 seg
 -- Artista: LEO
 -- ◆lbum: Life Is Real
 -- A◆o: null
 -- T◆tulo: Life Is Real
------------------------------------------------------------
------------------------------------------------------------

Reproduciendo...
```

Practica2 (run) × Practica2 (run) #2 ×

```
 -- Album: Life Is Real
 -- A�o: null
 -- T�tulo: Life Is Real
------------------------------------------------------------
------------------------------------------------------------

Reproduciendo...

Reproducci�n finalizada.

�Deseas escuchar otra canci�n? (s/n): n
BUILD SUCCESSFUL (total time: 4 minutes 22 seconds)
```

Practica2 (run) × Practica2 (run) #2 ×

```
 -- Total paquetes: 56
 -- N� de paquete: 56
 -- Tama�o datos: 30139 bytes
 -- Tama�o total archivo: 2780139 bytes
Enviado paquete #56
ACK recibido: 56
------------------------------------------------------------
------------------------------------------------------------

Transmisi�n completa de Life Is Real.mp3
Cliente solicit� cerrar la conexi�n. Cerrando servidor...
BUILD SUCCESSFUL (total time: 4 minutes 24 seconds)
```

CONCLUSIONES.

La realización de este proyecto ha sido una experiencia de aprendizaje sumamente valiosa, ya que me ha permitido implementar en la práctica conceptos teóricos fundamentales de las redes de computadoras. La principal lección fue comprender a fondo las diferencias entre los protocolos TCP y UDP y, más importante aún, entender *por qué* y *cómo* se construye la confiabilidad sobre un protocolo inherentemente no confiable.

En la práctica anterior, al utilizar TCP, daba por sentada la entrega ordenada y sin errores de los datos. En este proyecto, enfrentarme a la posibilidad real de paquetes perdidos o desordenados me obligó a diseñar un protocolo a nivel de aplicación. La implementación de la ventana deslizante fue el mayor desafío y, a la vez, el mayor logro. Ver cómo el sistema era capaz de recuperarse de un Timeout y retransmitir los datos perdidos para finalmente ensamblar un archivo MP3 funcional fue increíblemente gratificante. Me hizo apreciar la complejidad que opera silenciosamente detrás de las aplicaciones de streaming que usamos a diario.

Otro aspecto revelador fue el uso de la serialización de objetos. En lugar de diseñar un complejo protocolo basado en texto para enviar metadatos y datos binarios por separado, encapsular toda la información en un objeto Paquete y serializarlo simplificó enormemente la lógica de comunicación. Esta técnica demostró ser una herramienta poderosa y elegante para mantener la coherencia de los datos entre el emisor y el receptor, asegurando que cada fragmento de información llegara con todo el contexto necesario para ser procesado correctamente.

Finalmente, este proyecto reforzó mi comprensión sobre el diseño de sistemas distribuidos. El éxito de la aplicación no dependía solo de que el código del cliente o del servidor funcionara de forma aislada, sino de que ambos siguieran el mismo protocolo de manera impecable. Depurar problemas de comunicación, donde un ACK no llegaba o un paquete se deserializaba incorrectamente, me enseñó a pensar en el sistema como una única entidad colaborativa. Esta experiencia me ha proporcionado una base sólida para abordar problemas más complejos en el desarrollo de aplicaciones de red, donde la eficiencia y la fiabilidad son críticas.

REFERENCIAS BIBLIOGRÁFICAS.

- Kurose, J. F., & Ross, K. W. (2016). *Computer Networking: A Top-Down Approach*. Pearson Education.
- Oracle. (s. f.). *Java SE Documentation - DatagramSocket*. Obtenido de https://docs.oracle.com/javase/8/docs/api/java/net/DatagramSocket.html
- Postel, J. (1980). *RFC 768: User Datagram Protocol*. Obtenido de https://tools.ietf.org/html/rfc768
- The mp3agic library. (s. f.). *GitHub Repository*. Obtenido de https://github.com/mpatric/mp3agic
- JavaZOOM. (s. f.). *JLayer MP3 Library*. Obtenido de http://www.javazoom.net/javalayer/javalayer.html