Instituto Politécnico Nacional

Escuela Superior de Computo

Practica 2

CONTROL DE FLUJO

MATERIA

Aplicaciones para comunicaciones de red

PROFESOR

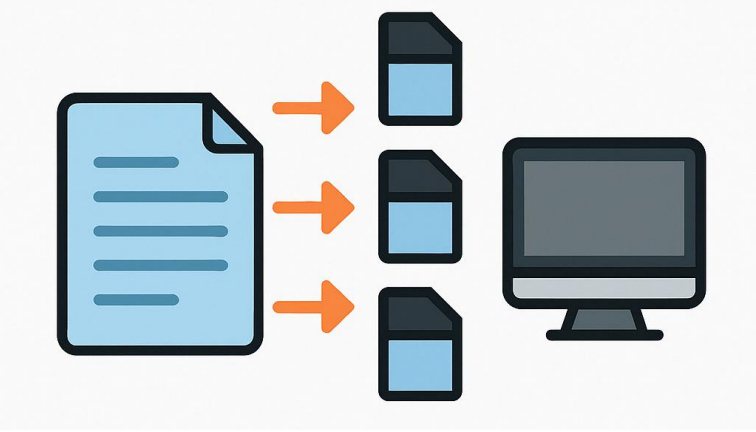
Moreno Cervantes Axel Ernesto

GRUPO

6CM2

ALUMNOS

- Guevara Badillo Areli Alejandra
- Ramírez Martínez Alejandro



FECHA

lunes, 14 de abril de 2025

INTRODUCCIÓN

En las comunicaciones de red, el protocolo UDP permite enviar datos rápidamente, pero no garantiza que lleguen de forma correcta, ordenada o sin duplicados. Por eso, es necesario implementar mecanismos adicionales para asegurar una transmisión confiable de datos. En esta práctica se desarrolla una aplicación cliente-servidor que simula la transferencia de archivos por medio de sockets de datagrama (UDP), incorporando técnicas de control de flujo y control de errores que permitan asegurar la entrega correcta de los paquetes, incluso en presencia de pérdidas o duplicados.

OBJETIVOS

- Implementar control de flujo usando una ventana deslizante.
- Aplicar un mecanismo de control de errores para asegurar la entrega correcta.
- Simular el envío de un archivo confiable usando sockets UDP.
- Evaluar el comportamiento ante pérdida, duplicación o desorden de paquetes.

DESARROLLO

Esta práctica implementa una transferencia confiable de archivos entre un cliente y un servidor utilizando **sockets UDP**, que por naturaleza no garantizan la entrega, ni el orden correcto de los datos. Para solucionar esto, se agregaron dos mecanismos clave:

- **Control de flujo**: con el algoritmo de **ventana deslizante**.
- **Control de errores**: mediante el esquema de **rechazo selectivo**.

Ambos mecanismos permiten manejar pérdida de paquetes, duplicados y desorden en la entrega.

La aplicación está dividida en Cliente y Servidor, cada uno con responsabilidades claras para garantizar él envió de archivos.

- **Cliente.java:** Se encarga de leer un archivo, dividirlo en fragmentos y enviarlo al servidor utilizando una ventana deslizante.
- **Servidor.java:** Escucha los paquetes, verifica duplicados y guarda cada parte usando su número de secuencia.

Las partes más importantes giran en torno al control de flujo, donde el cliente usa una ventana deslizante para enviar paquetes sin esperar confirmación de todos, optimizando el envío; al manejo de ACKs, ya que el cliente espera confirmaciones de cada paquete y reenvía los no confirmados; al control de errores, en el que el servidor valida y confirma los paquetes con ACKs para evitar duplicados y pérdidas; y a la reconstrucción del archivo, donde el servidor junta los paquetes recibidos para formar el archivo original. Todo esto garantiza una transferencia confiable del archivo.

CLIENTE.JAVA

Selecciona un archivo, lo divide en paquetes y los envía al servidor usando UDP, esperando confirmaciones (ACK) por cada paquete. Si no recibe un ACK, reenvía los paquetes no confirmados. Una vez terminado, envía un paquete especial para indicar el fin de la transmisión.

Selección del archivo:

Cuando se ejecuta el programa cliente, abre una ventana para que el usuario elija un archivo. Luego lee todos los bytes del archivo.

```
byte[] fileData = Files.readAllBytes(file.toPath());
```

Divide el archivo en paquetes

Se calcula cuántos paquetes se van a enviar.

```
int totalPackets = (int) Math.ceil((double) fileData.length / PACKET_SIZE);
```

Luego, se usa una ventana deslizante que mantiene controlados qué paquetes se envían y cuáles están esperando ACK.

Envío con ventana deslizante

El cliente envía WINDOW_SIZE paquetes (4 por defecto) y espera sus ACKs.

```
while (nextSeqNum < base + WINDOW_SIZE && nextSeqNum < totalPackets) {
    // Se arma el paquete con 4 bytes de número de secuencia + datos
    byte[] packetData = new byte[end - start + 4];
    byte[] seqBytes = intToBytes(nextSeqNum);
    System.arraycopy(seqBytes, 0, packetData, 0, 4);
    System.arraycopy(fileData, start, packetData, 4, end - start);

    DatagramPacket sendPacket = new DatagramPacket(packetData, packetData.length, IPAddress, SERVER_PORT);
    socket.send(sendPacket);
    nextSeqNum++;
}
```

Espera los ACKs:

Luego escucha el número de secuencia del ACK que viene del servidor.

```
DatagramPacket ackPacket = new DatagramPacket(ackData, ackData.length);
socket.receive(ackPacket);

int ackNum = bytesToInt(ackPacket.getData());
```

Cuando llega un ACK, se actualiza el array acked[] para marcar ese paquete como recibido.

Si se recibe ACKs consecutivos, se mueve la base de la ventana (ventana deslizante).

```
while (base < totalPackets && acked[base]) base++;
```

Reenvío selectivo en caso de pérdida:

Si no se recibe un ACK antes del TIMEOUT, el cliente reenvía todos los paquetes a partir del último no confirmado (rechazo selectivo).

```
catch (SocketTimeoutException e) {
    System.out.println("Timeout. Reenviando desde paquete #" + base);
    nextSeqNum = base;  // ← vuelve a enviar desde el primero que no ha sido ACKed
}
```

SERVIDOR.JAVA

Recibe los paquetes, los confirma con ACKs y, al finalizar la transmisión, reconstruye el archivo a partir de los fragmentos recibidos.

Espera paquetes:

Se queda escuchando el socket en un bucle.

```
DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
socket.receive(receivePacket);
```

Recibe y analiza cada paquete:

Cada paquete tiene 4 bytes de número de secuencia y el resto son datos del archivo.

```
int seqNum = ((data[0] & 0xFF) << 24) |
             ((data[1] & 0xFF) << 16) |
             ((data[2] & 0xFF) << 8) |
             (data[3] & 0xFF);
```

Luego se saca solo la parte de datos.

```
byte[] filePart = Arrays.copyOfRange(data, 4, data.length);
```

Evita duplicados y guarda los datos:

Usa un HashMap<Integer, byte[]> para almacenar solo una vez cada paquete.

```
if (!fileChunks.containsKey(seqNum)) {
    fileChunks.put(seqNum, filePart);
    System.out.println("Recibido paquete #" + seqNum);
} else {
    System.out.println("Duplicado paquete #" + seqNum + " ignorado");
}
```

Manda ACKs al cliente:

Cada vez que recibe un paquete válido, responde con un ACK que incluye el número de secuencia recibido.

```
byte[] ackBytes = {
    (byte) (seqNum >> 24),
    (byte) (seqNum >> 16),
    (byte) (seqNum >> 8),
    (byte) seqNum
};
DatagramPacket ackPacket = new DatagramPacket(ackBytes, ackBytes.length,
receivePacket.getAddress(), receivePacket.getPort());
socket.send(ackPacket);
```

Reconstrucción del archivo:

Cuando llega un paquete con seqNum == -99, se termina la recepción y se ensamblan los fragmentos en orden.

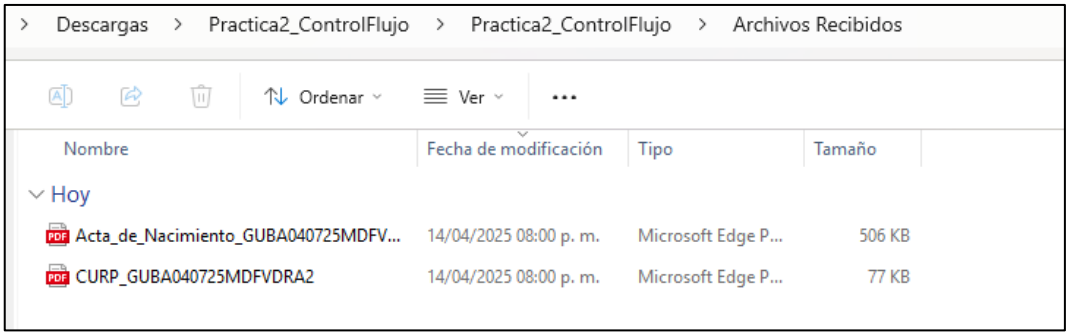
```
for (int i = 0; i < fileChunks.size(); i++) {
    fos.write(fileChunks.get(i));
}
```

PRUEBAS Y RESULTADOS

Para verificar el correcto funcionamiento de la aplicación de control de flujo, se realizaron pruebas en un entorno simulado, donde el cliente en vía el archivo separado en fragmentos por medio de la ventana deslizante y el servidor recibe estos fragmentos y los acomoda para obtener el archivo completo.

Carpeta Archivos Recibidos:

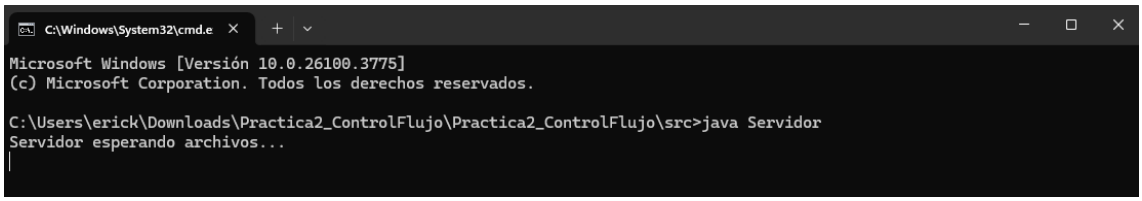
Se encuentra alojada en nuestro proyecto, aquí se guarda el archivo enviado desde el cliente.



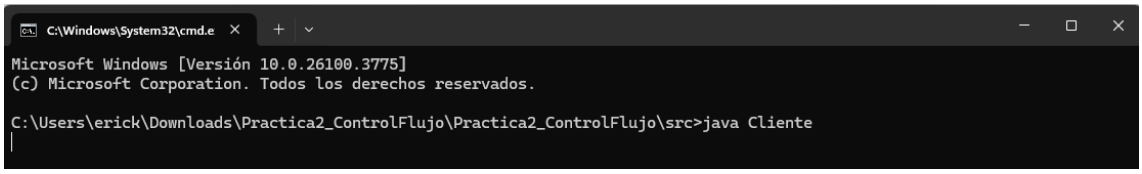
Conexión Cliente-Servidor:

Primero debemos ejecutar el Servidor y luego el Cliente, esto para establecer la conexión entre ambos.

Ejecutamos el Servidor:

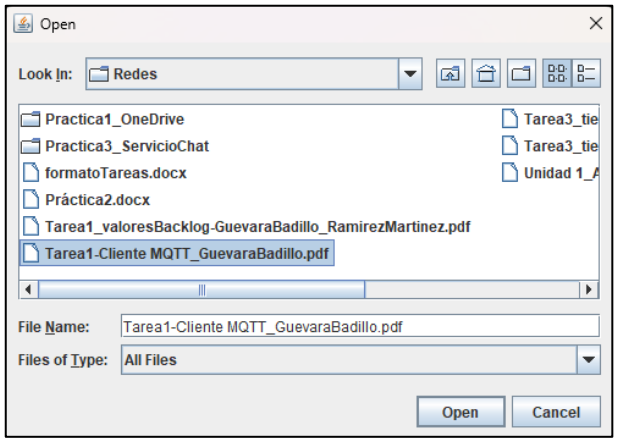


Ejecutamos el Cliente:



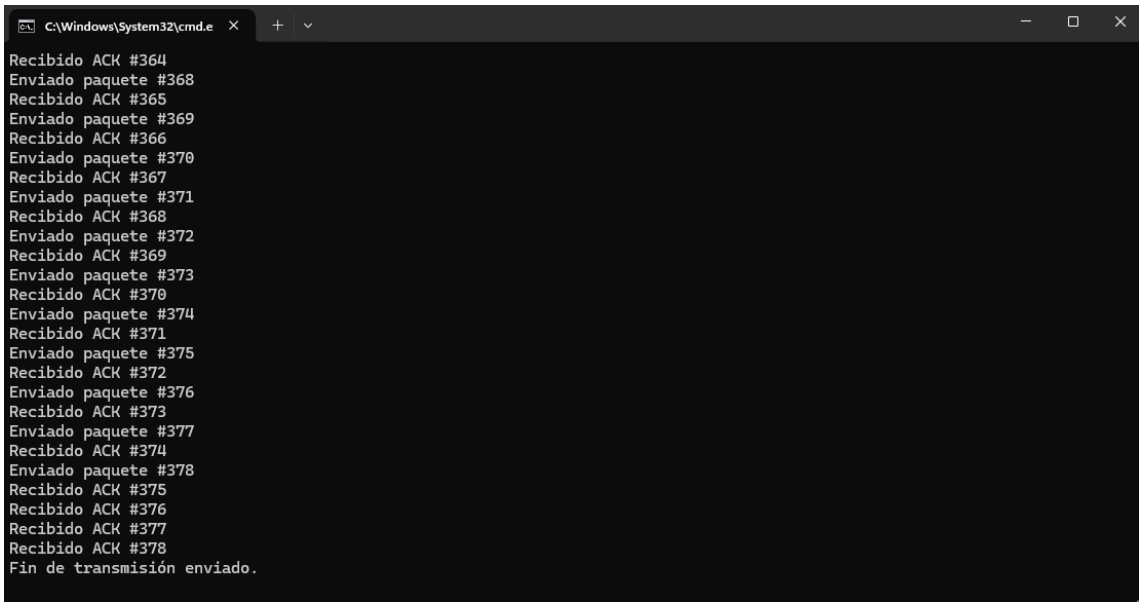
Selección del archivo:

Al ejecutar el Cliente, debemos elegir el archivo que enviaremos mediante socket de datagrama, en este caso **Tarea1-Cliente MQTT_GuevaraBadillo.pdf** este se enviará por fragmentos a la carpeta **Archivos Recibidos** ubicada en la carpeta que aloja nuestro proyecto.

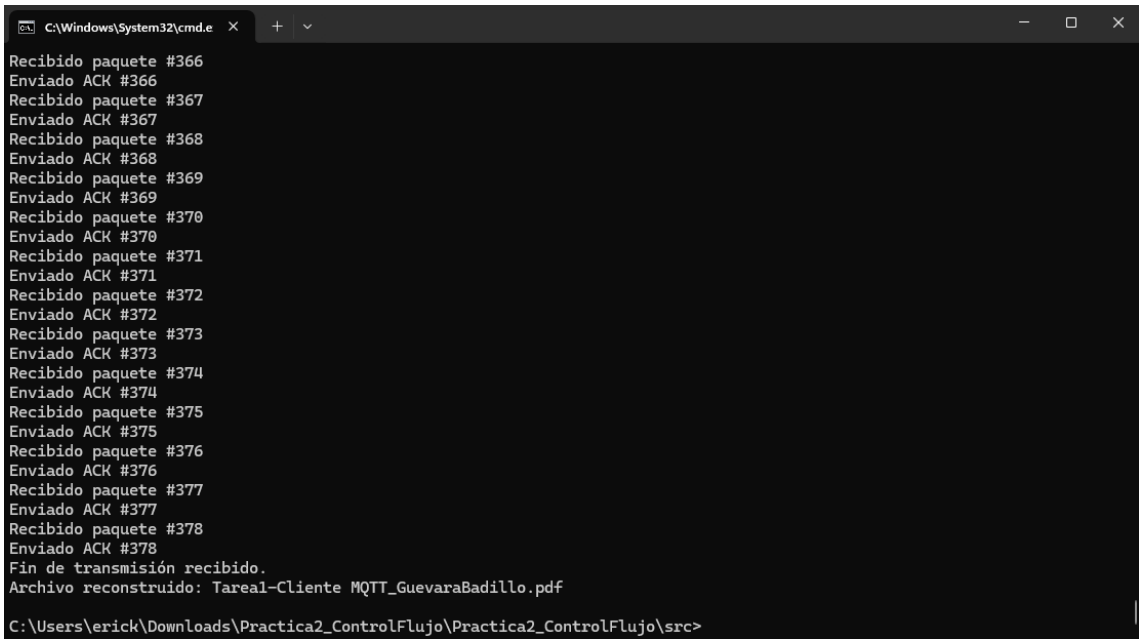


Envió del archivo:

Se muestra cómo se enviaron los diferentes fragmentos del archivo desde el Cliente:

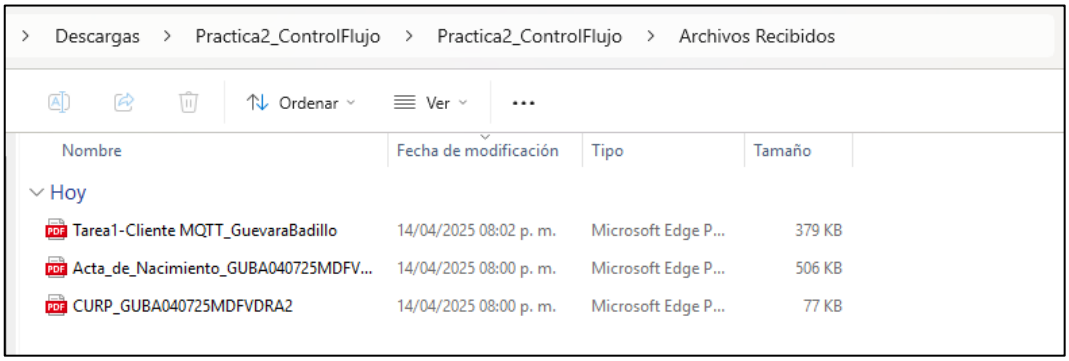


Se muestra cómo se recibieron los fragmentos desde el Servidor:



Carpeta Archivos Recibidos actualizada:

La carpeta donde se guardan los archivos debe actualizarse con el nuevo archivo.



Los resultados de las pruebas demostraron que:

- Es posible lograr una transmisión confiable utilizando el protocolo UDP al implementar mecanismos adicionales de control.
- El uso de ventana deslizante mejoró significativamente el rendimiento al permitir el envío continuo de varios paquetes sin esperar confirmaciones individuales.
- El rechazo selectivo fue efectivo, ya que se reenviaron únicamente los paquetes no confirmados, optimizando el uso del canal.
- Se puedo mantener la entrega ordenada de los datos incluso ante pérdidas, duplicados o paquetes fuera de orden, gracias al uso de números de secuencia.
- El sistema reaccionó correctamente ante la pérdida de paquetes, reenviando solo lo necesario dentro del tiempo definido.

CONCLUSIONES

Durante esta práctica se pudo comprobar que, si bien el protocolo UDP es más rápido, no garantiza la entrega de los datos, lo que hizo necesario implementar mecanismos adicionales para asegurar una transmisión confiable. Uno de estos mecanismos fue el **control de flujo** mediante una **ventana deslizante**, que permitió optimizar el envío de paquetes al transmitir varios de manera continua sin esperar una confirmación individual, mejorando así el rendimiento general.

Además, se utilizó el **rechazo selectivo** como técnica de **control de errores**, lo cual permitió que el cliente reenviara únicamente los paquetes que no fueron confirmados, en lugar de toda la ventana, siguiendo una lógica basada en el reenvío desde la base al no recibir confirmación dentro del tiempo establecido. Gracias a la implementación del número de secuencia y las confirmaciones del servidor, se logró una transmisión completa y ordenada del archivo, incluso en escenarios con pérdidas, duplicados o llegada fuera de orden.

Esta experiencia permitió comprender mejor la importancia de los protocolos de nivel de transporte y cómo se pueden implementar manualmente funciones que protocolos como TCP ya traen integradas, reforzando así los conceptos de fiabilidad, control de flujo y manejo de errores.