Trabajo Práctico 1
File Transfer - Grupo 9

[75.43] Introducción a los Sistemas Distribuidos
Segundo cuatrimestre de 2025

ALUMNO	PADRON	CORREO
BARTOCCI, Camila	105781	cbartocci@fi.uba.ar
PATIÑO, Franco	105126	fpatino@fi.uba.ar
RETAMOZO, Melina	110065	mretamozo@fi.uba.ar
SAGASTUME, Matias	110530	csagastume@fi.uba.ar
SENDRA, Alejo	107716	asendra@fi.uba.ar

Índice

1. Introducción

Este trabajo implementa un protocolo RDT (Reliable Data Transfer) implementado para transferencia confiable de archivos sobre UDP. El protocolo soporta operaciones de upload y download con dos estrategias de recuperación de errores: Stop & Wait y Selective Repeat, garantizando entrega confiable con hasta 10 % de pérdida de paquetes.

El diseño del protocolo se basa en los principios de transferencia confiable de datos, implementando mecanismos de:

- Detección de errores mediante checksums
- Numeración de secuencia para detectar duplicados
- Retransmisión automática ante pérdida de paquetes
- Control de flujo mediante ventanas deslizantes (Selective Repeat)
- 3-Way-Handshake para establecimiento de sesión
- Manejo concurrente de múltiples transferencias

Contexto del Problema

Objetivos del Trabajo Práctico

Alcance y Limitaciones

Estructura del Informe

2. Hipótesis y suposiciones realizadas

Durante el desarrollo del protocolo, se realizaron las siguientes hipótesis y suposiciones:

2.1. Sobre el Entorno de Red

1. **Pérdida de paquetes**: Se asume que la pérdida de paquetes es aleatoria y no supera el 10 %. No se consideran escenarios con pérdida masiva o sistemática.

2. **Reordenamiento**: Se asume que los paquetes pueden llegar desordenados, pero que el reordenamiento no es extremo. El protocolo maneja esto con números de secuencia y buffers.

3. **Duplicación**: Se asume que puede haber paquetes duplicados debido a retransmisiones, y el protocolo debe ser idempotente.

4. **Corrupción**: Se asume que UDP puede entregar paquetes corruptos, aunque es poco común. Se implementa checksum para detección.

2.2. Sobre la Aplicación

1. **Tamaño de archivos**: Los archivos a transferir no superan los 5 MB. Esta limitación permite mantener estructuras de datos simples y eficientes.

2. **Nombres de archivo**: Se asume que los nombres de archivo son válidos en UTF-8 y no contienen caracteres especiales que puedan causar problemas en el sistema de archivos.

3. **Unicidad de archivos**: El servidor sobrescribe archivos con el mismo nombre sin previo aviso. No se implementó versionado.

4. **Recursos del sistema**: Se asume que el sistema tiene suficiente memoria para mantener buffers de ventanas y archivos en memoria durante la transferencia.

2.3. Sobre el Protocolo

1. **Session IDs**: Con un espacio de 255 IDs posibles y un límite de 10 sesiones concurrentes, se asume que siempre hay IDs disponibles.

2. **Números de secuencia**: El espacio de 256 números (0-255) es suficiente para ventanas de tamaño 20-30 paquetes con el mecanismo de wrap-around.

3. **Timeouts**: Se asumieron valores fijos de timeout basados en una red de baja latencia (LAN/localhost). En una WAN real, se necesitaría un mecanismo de timeout adaptativo.

4. **Orden de operaciones**: Se asume que el handshake siempre se completa antes de comenzar la transferencia de datos. No se consideran casos de transferencia parcial con re-handshake.

2.4. Sobre la Concurrencia

1. **Thread-safety**: Se asume que las estructuras de datos compartidas (como `active_sessions`) son accedidas con locks apropiados para evitar race conditions.
2. **Puertos disponibles**: Se asume que el sistema operativo siempre puede asignar un puerto dinámico para cada nueva sesión.
3. **Límite de concurrencia**: Se estableció un límite de 10 transferencias simultáneas como balance entre capacidad y uso de recursos.

2.5. Decisiones de Diseño Basadas en Suposiciones

Estas suposiciones llevaron a las siguientes decisiones:

- **Tamaño de paquete de 4096 bytes**: Balance entre eficiencia y fragmentación IP.
- **Timeout de 50ms (Stop & Wait) y 200ms (Selective Repeat)**: Optimizado para LAN.
- **Ventana de 20 paquetes**: Suficiente para mantener el pipeline lleno sin abrumar al receptor.
- **Máximo 20 reintentos**: Suficiente para recuperarse de pérdidas transitorias.

2.6. Marco teórico

2.7. Hipótesis

2.8. Suposiciones

2.9. Arquitectura General

La implementación se dividió en varios módulos con responsabilidades bien definidas, siguiendo principios de diseño orientado a objetos. El sistema está compuesto por tres componentes principales:

1. **Cliente (upload.py / download.py):** Aplicaciones CLI independientes
2. **Servidor (start-server.py):** Servidor concurrente multi-thread
3. **Librería RDT (lib/):** Módulos compartidos para el protocolo

2.10. Estructura de paquetes

Todos los paquetes siguen una estructura de header fijo de 14 bytes seguido de un payload variable.

Campo	Tamaño	Descripción
`seq_num`	1 byte	Número de secuencia (0-255, con wrap-around)
`checksum`	1 byte	Suma de verificación para detección de errores
`ack_num`	1 byte	Número de ACK (usado en paquetes ACK)
`payload_length`	4 bytes	Longitud del payload en bytes
`file_size`	4 bytes	Tamaño total del archivo (solo en INIT)
`packet_type`	1 byte	Tipo de paquete (ver tabla siguiente)
`protocol`	1 byte	Protocolo utilizado (1=Stop&Wait, 2=Selective Repeat)
`session_id`	1 byte	Identificador de sesión (1-255)

Cuadro 1: Campos del Header (14 bytes)

2.11. Tipos de paquetes

En el protocolo implementado, cada paquete tiene un tipo específico que determina su función dentro de la transferencia de archivos. A continuación se describen los tipos de paquetes utilizados y su propósito principal:

- **DATA (1)**: Transferencia de datos, contiene los fragmentos del archivo.
- **ACK (2)**: Acknowledgment de paquetes DATA recibidos correctamente.
- **UPLOAD_INIT (3)**: Inicia una sesión de subida (upload) de archivo.
- **DOWNLOAD_INIT (4)**: Inicia una sesión de descarga (download) de archivo.
- **ACCEPT (5)**: Acepta la transferencia en respuesta a un paquete INIT.
- **ACCEPT_ACK (6)**: Confirma la finalización del handshake.
- **FIN (7)**: Indica la finalización de la transferencia de datos.

- **FIN_ACK (8)**: Confirma la finalización de la sesión.
- **ERROR (9)**: Mensaje de error, enviado por el servidor ante problemas o rechazos.

Tipo	Dirección	Payload	Campos Relevantes
UPLOAD_INIT (3)	Cliente → Servidor	Nombre del archivo (UTF-8)	`file_size` (bytes), `protocol`, `session_id=0`
DOWNLOAD_INIT (4)	Cliente → Servidor	Nombre del archivo (UTF-8)	`file_size=0`, `protocol`, `session_id=0`
ACCEPT (5)	Servidor → Cliente	Puerto dedicado en ASCII (ej: "52341") o vacío	`session_id` (1–255 asignado)
ACCEPT_ACK (6)	Cliente → Servidor	Vacío	`session_id`, `ack_num=0`
DATA (1)	Bidireccional	Fragmento de archivo (hasta 4096 bytes)	`seq_num`, `session_id`, `payload_length`
ACK (2)	Receptor → Emisor	Vacío	`ack_num`, `session_id`
FIN (7)	Emisor → Receptor	Vacío	`session_id`
FIN_ACK (8)	Receptor → Emisor	Vacío	`session_id`
ERROR (9)	Servidor → Cliente	Mensaje de error en UTF-8	Texto descriptivo (ej: "File not found")

Cuadro 2: Tipos de Paquetes del Protocolo

2.12. Flujos del Protocolo

2.12.1. Handshake (upload)

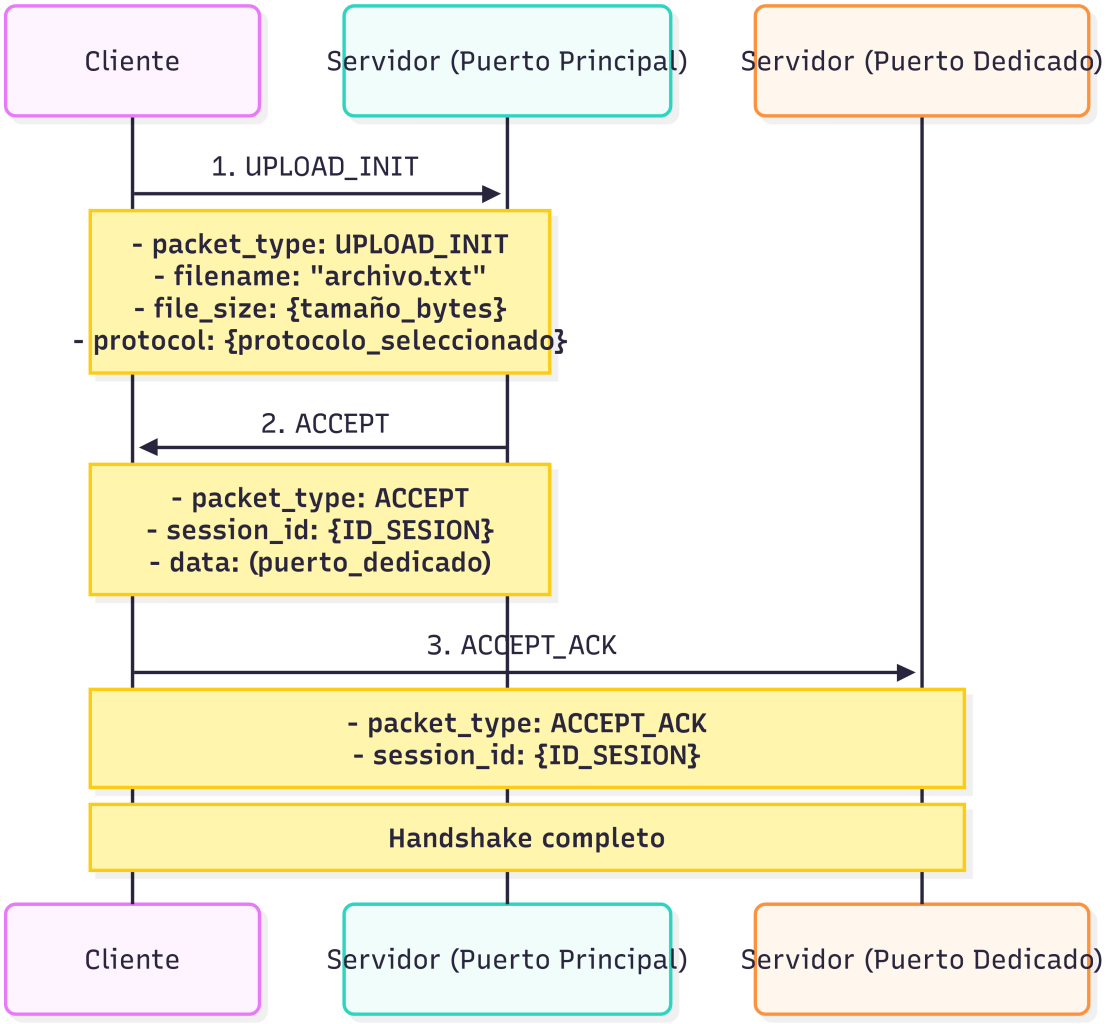

Figura 1: Handshake inicial para subida de archivo

2.12.2. Handshake (download)

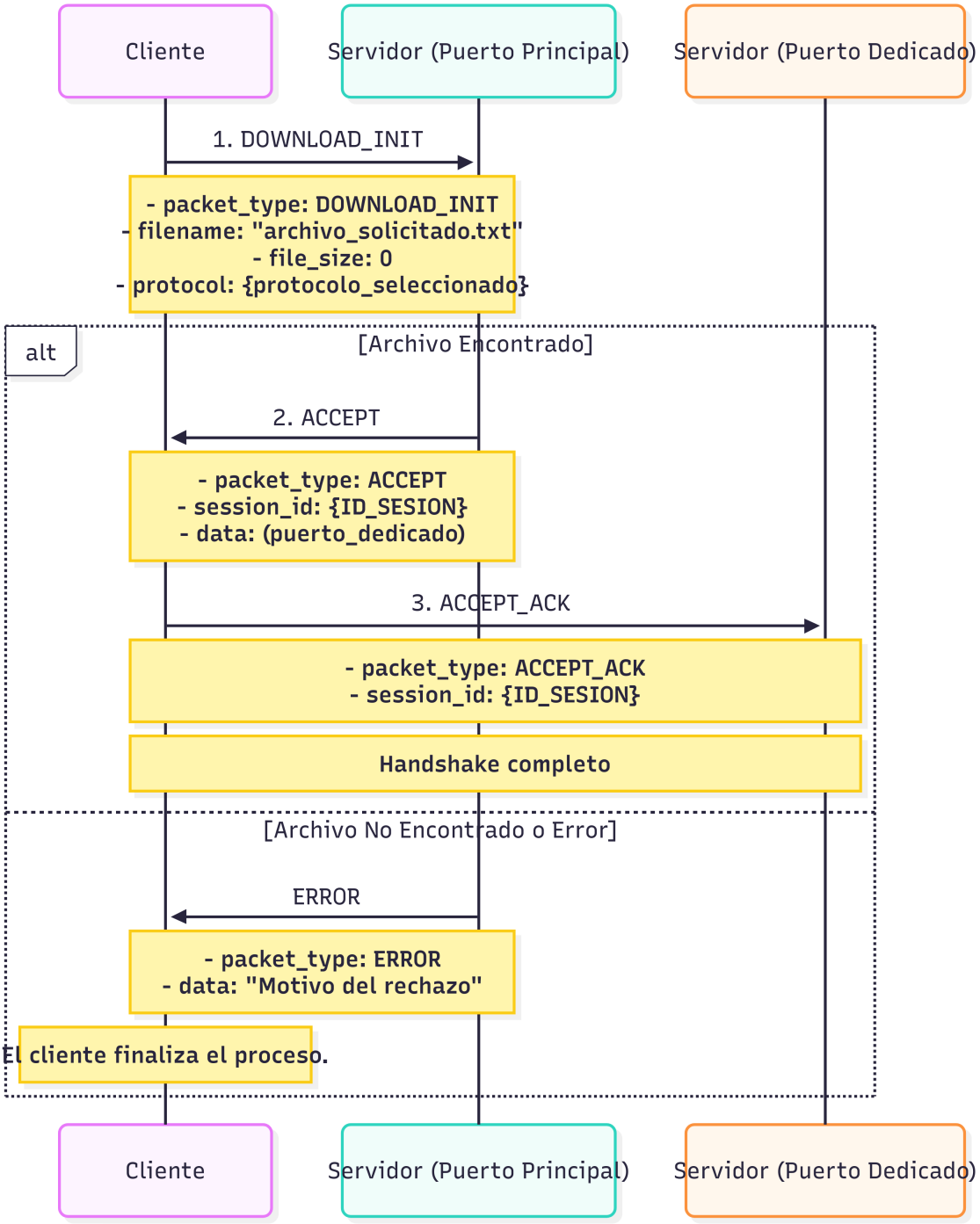

Figura 2: Handshake inicial para descarga de archivo

2.12.3. Cierre de conexion

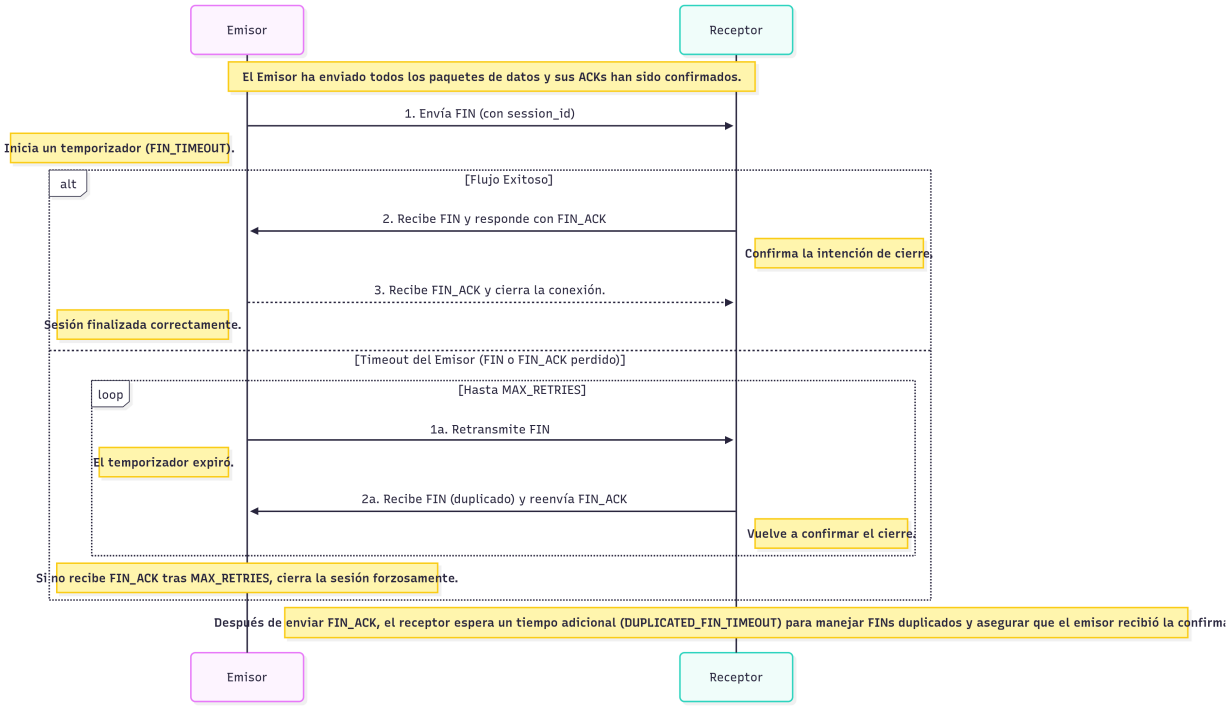

Figura 3: Subida / Descarga de archivo con Selective Repeat

2.12.4. Subida / Descarga de archivo

En el caso de una subida de archivo, el emisor es el Cliente y el receptor el Servidor. En el caso de una descarga, el emisor es el Servidor y el receptor el Cliente. El flujo de envio de paquetes DATA es el mismo para ambos casos.

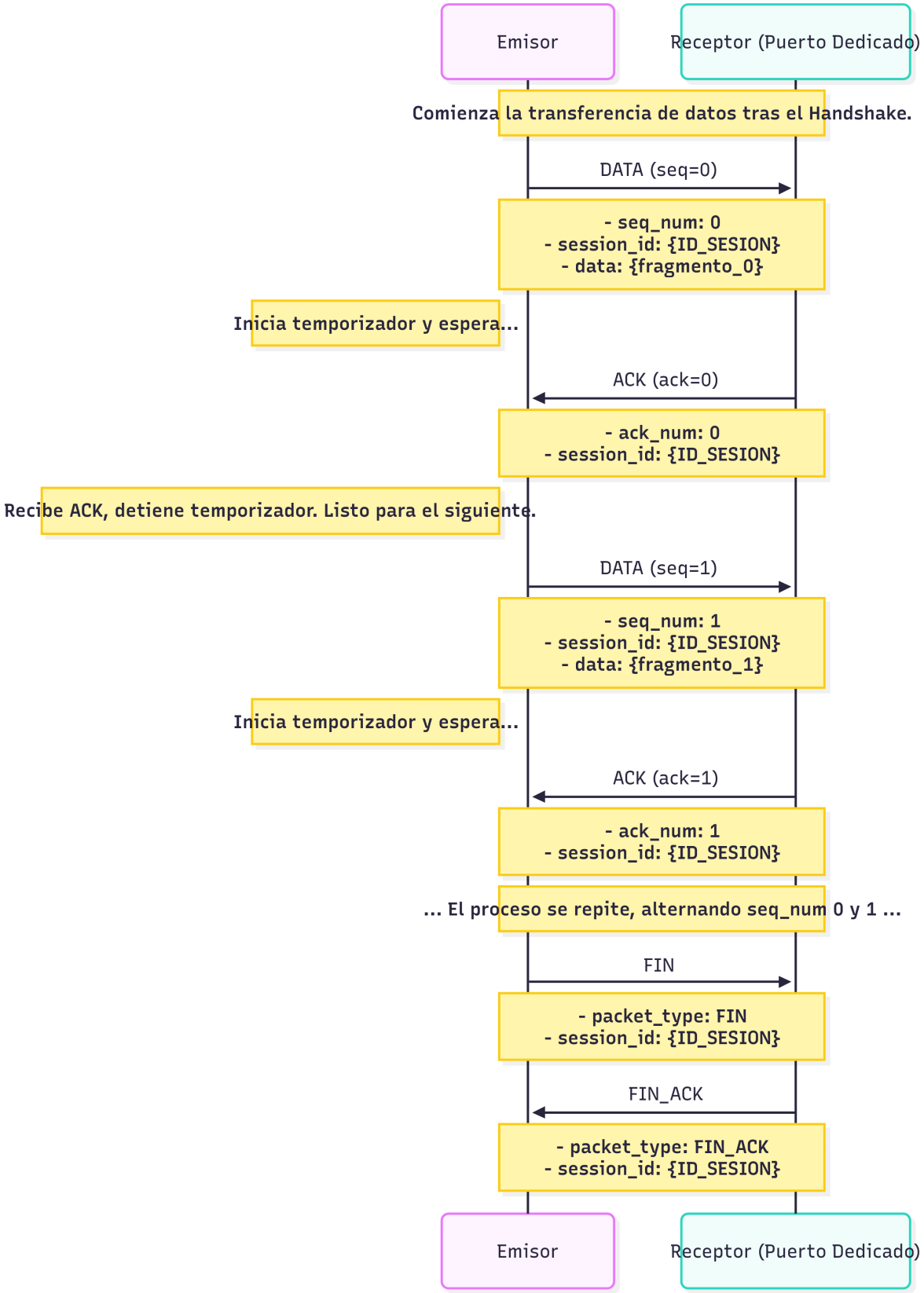

Figura 4: Subida / Descarga de archivo con Stop & Wait

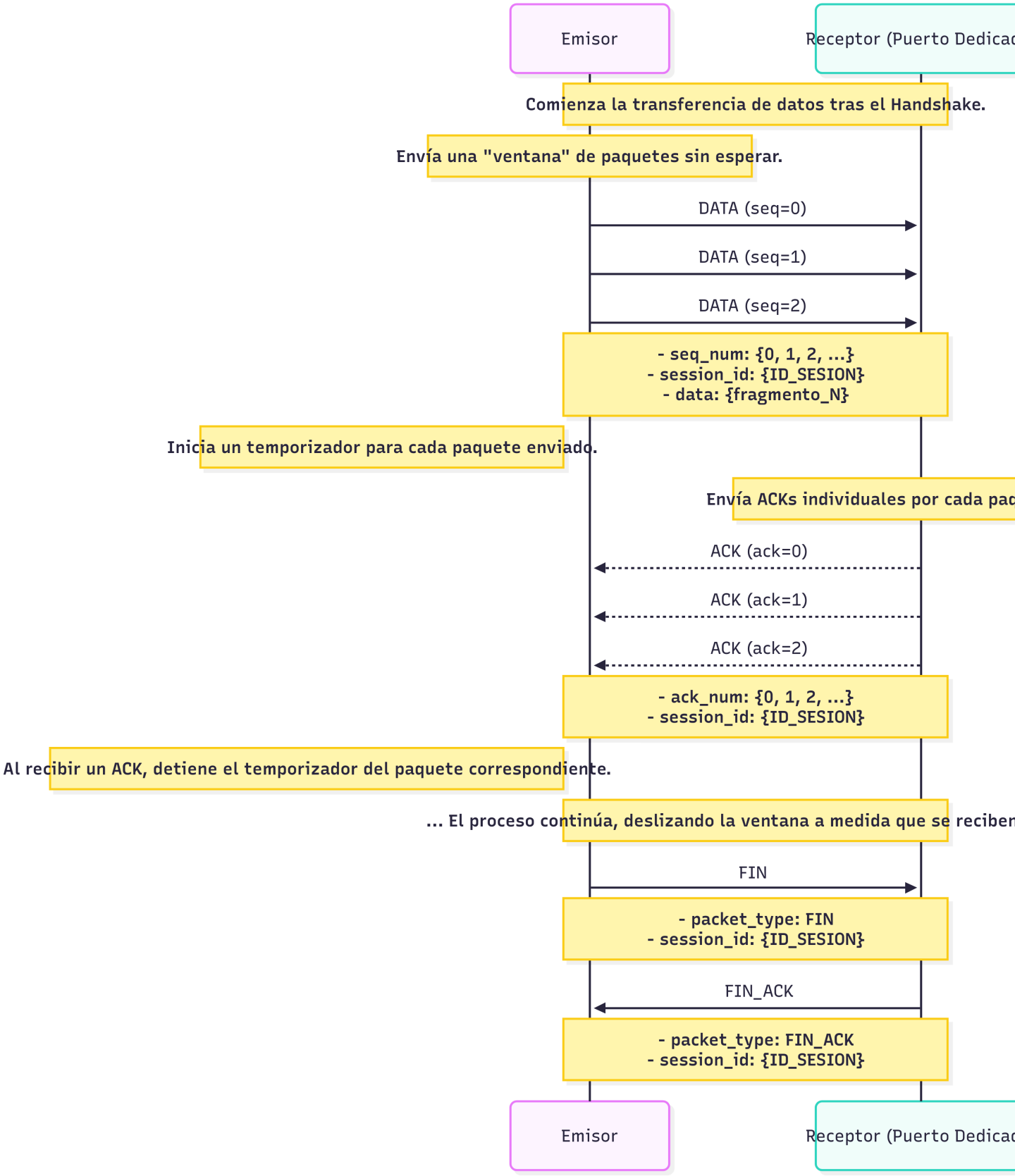

Figura 5: Subida / Descarga de archivo con Selective Repeat

2.12.5. Manejo de errores

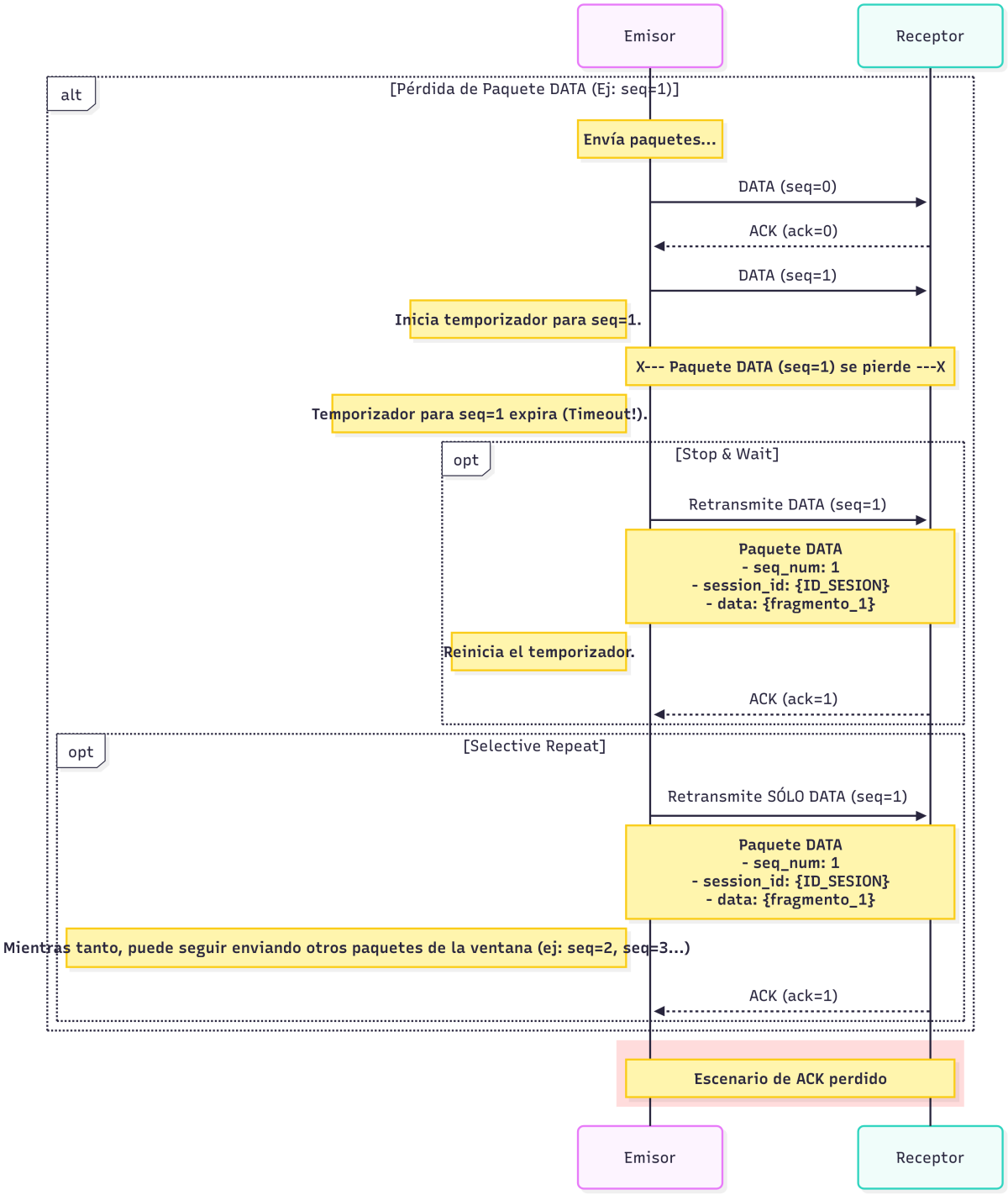

Figura 6: Retramision de paquetes por timeout

3. Manejo de Concurrencia

3.1. Arquitectura Concurrente

El servidor implementado es capaz de atender múltiples clientes simultáneamente sin que las transferencias interfieran entre sí. El servidor utiliza un modelo de concurrencia basado en threads con recursos dedicados por sesión. La arquitectura se puede describir en tres niveles:

3.1.1. Thread Principal (Main Thread)

El thread principal del servidor ejecuta un loop infinito que:

1. Escucha en el puerto principal (49153) solicitudes de inicio de transferencia
2. Valida que haya capacidad disponible
3. Crea recursos dedicados para cada nueva transferencia
4. Delega el manejo de la transferencia a un thread dedicado
5. Registra la sesión en el diccionario de sesiones activas

Este diseño permite que el thread principal continúe aceptando nuevas conexiones mientras las transferencias en curso se ejecutan en paralelo.

3.1.2. Threads Dedicados (Worker Threads)

Cada transferencia corre en su propio thread con recursos completamente aislados:

- **Socket UDP dedicado:** Cada sesión tiene su propio socket escuchando en un puerto dinámico único
- **Session ID único:** Identificador de 1-255 que no colisiona con otras sesiones activas
- **Estado independiente:** Variables de protocolo (ventanas, buffers) no compartidas entre threads
- **File descriptor:** Cada thread maneja su propio archivo sin compartir recursos de I/O

Este aislamiento garantiza que un error o problema en una transferencia no afecta a las demás.

3.2. Gestión de Recursos Compartidos

Aunque cada transferencia está mayormente aislada, existen algunos recursos compartidos que requieren sincronización cuidadosa:

3.2.1. Diccionario de Sesiones Activas

El servidor mantiene un diccionario global de todas las sesiones activas, que contiene para cada sesión:

- `session_id`: Identificador único (1-255)
- `dedicated_port`: Puerto UDP asignado
- `thread`: Referencia al thread worker
- `dedicated_socket`: Socket UDP dedicado
- `client_addr`: Dirección IP y puerto del cliente
- `created_at`: Timestamp de creación

Para garantizar consistencia, todas las operaciones sobre este diccionario están protegidas por un lock (`sessions_lock`). Las operaciones sincronizadas incluyen:

- Agregar nueva sesión
- Remover sesión completada
- Verificar cantidad de sesiones activas
- Generar session ID único

3.3. Puertos Dedicados

Cada transferencia obtiene su propio puerto UDP, lo que proporciona aislamiento a nivel de socket del sistema operativo.

3.3.1. Asignación Dinámica de Puertos

El servidor crea un socket UDP y lo vincula al puerto 0, lo que indica al sistema operativo que asigne automáticamente un puerto disponible del rango de puertos dinámicos (típicamente 49152-65535). Este mecanismo tiene varias ventajas:

1. **Aislamiento completo:** Los paquetes de diferentes transferencias nunca se mezclan, ya que cada una tiene su propio socket
2. **Sin multiplexación:** No hay necesidad de demultiplexar paquetes por session_id en el socket principal, simplificando la lógica
3. **Simplicidad:** Cada thread puede ejecutar `recvfrom()` sin condiciones de carrera ni coordinación con otros threads
4. **Escalabilidad:** El sistema operativo maneja eficientemente la distribución de puertos y el enrutamiento de paquetes
5. **Robustez:** Si un socket falla o se bloquea, no afecta a los demás

3.4. Control de Capacidad

El servidor implementa un límite configurable de transferencias concurrentes para prevenir agotamiento de recursos del sistema.

Antes de aceptar una nueva transferencia, el servidor verifica que el número de sesiones activas no supere el máximo configurado (10 por defecto). Si el servidor está a capacidad:

1. Se registra una advertencia en los logs
2. Se envía un paquete ERROR al cliente con el mensaje "Server at capacity, try again later"
3. Se rechaza la conexión sin crear recursos
4. El cliente puede reintentar más tarde

3.5. Flujo Completo de una Sesión Concurrente

A continuación se describe el ciclo de vida completo de una sesión en el servidor concurrente:

```
1. Cliente envia UPLOAD_INIT/DOWNLOAD_INIT al puerto 49153
   |
   +---> Thread Principal recibe solicitud
   |     |
   |     +---> Adquiere lock sobre sessions_lock
   |     +---> Verifica capacidad (< 10 sesiones)
   |     +---> Libera lock
   |
   +---> Si capacidad disponible:
   |     |
   |     +---> Genera session_id unico (con lock)
   |     +---> Crea socket dedicado en puerto N
   |     +---> Envia ACCEPT(session_id, puerto N)
   |     |
   |     +---> Crea thread dedicado T
   |     +---> Registra sesion en active_sessions (con lock)
   |     +---> Inicia thread: T.start()
   |
   +---> Si capacidad completa:
         +---> Envia ERROR("Server at capacity")

2. Thread Dedicado T ejecuta:
   |
   +---> Espera ACCEPT_ACK en socket dedicado
   +---> Crea instancia de protocolo (Stop&Wait o Selective Repeat)
   +---> Ejecuta transferencia (bloqueante solo para este thread)
   +---> Guarda archivo en disco
   |
   +---> Finally (siempre ejecuta):
         +---> Adquiere lock sobre sessions_lock
         +---> Elimina sesion de active_sessions
         +---> Libera lock
         +---> Cierra socket dedicado

3. Thread Principal continua:
   +---> Loop: Espera proxima solicitud (no bloqueado por T)
```

Figura 7: Ciclo de vida de una sesión concurrente

4. Implementación

4.1. Arquitectura del sistema

4.2. Realiable Data Transfer

4.2.1. Estructuras compartidas

Creación de paquetes

Numeración de secuencia

Acknowledgments

Timers y retransmisiones

4.2.2. Stop & Wait

4.2.3. Selective Repeat

4.3. Concurrencia

4.4. Manejo de errores

4.5. Interfaces de usuario

4.6. Configuración de parámetros

5. Pruebas

5.1. Métricas de rendimiento

Para evaluar el rendimiento de la implementación y poder comparar ambos protocolos, se han utilizado las siguientes métricas:

- **RTT promedio:** Tiempo entre el envío de un paquete y la recepción de su acuse de recibo.
- **Jitter:** Desviación estándar del RTT.
- **Throughput:** Cantidad de bytes recibidos sobre el tiempo transcurrido.
- **Overhead:** Cantidad de retransmisiones sobre la cantidad de paquetes enviados.
- **Uso promedio de las ventanas de envío y recepción:** Promedio de tamaño de las ventanas de envío y recepción utilizadas durante la transmisión.

5.2. Herramientas utilizadas

Se utilizó la herramienta *mininet* para simular una LAN simple con tres hosts conectados a través de un switch central. El primer host simula un cliente con una red problemática (10 % de pérdida de paquetes). Los dos primeros hosts están limitados a un ancho de banda de 10 Mbps y a una cola de hasta 1000 paquetes, mientras que el tercer host simula un servidor con una red estable y sin limitaciones de ancho de banda.

Para el cálculo de las métricas mencionadas en la sección anterior, se empleó una estructura de datos que permite almacenar la información necesaria para cada una de ellas. Esta estructura se encuentra en el archivo `lib/stats/stats_structs.py` y fue utilizada tanto por el servidor como por los clientes.

5.3. Escenarios de prueba

Para evaluar el rendimiento de los protocolos implementados, se generó un conjunto de archivos de prueba con tamaños entre 1 KB y 10 MB. Cada archivo fue transferido utilizando ambos protocolos (Stop & Wait y Selective Repeat) en distintas condiciones de red simuladas por *mininet*.

Cada prueba se repitió cinco veces por archivo, protocolo y condición de red, y se registraron las métricas descritas previamente. Además, las pruebas se llevaron a cabo tanto de forma secuencial como concurrente. En este último caso, se enviaron múltiples archivos de manera simultánea desde diferentes clientes.

5.4. Resultados obtenidos

6. Preguntas a responder

- Describa la arquitectura Cliente-Servidor
- ¿Cuál es la función de un protocolo de capa de aplicación?
- Detalle el protocolo de aplicación desarrollado en este trabajo.
- La capa de transporte del stack TCP/IP ofrece dos protocolos: TCP y UDP. ¿Qué servicios proveen dichos protocolos? ¿Cuáles son sus características? ¿Cuándo es apropiado utilizar cada uno?

7. Dificultades encontradas

8. Conclusiones