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

La transferencia de archivos es una operación fundamental en sistemas distribuidos. Sin embargo, cuando se utiliza UDP como protocolo de transporte, no se cuenta con mecanismos de confiabilidad como en TCP. Esto obliga a implementar protocolos de Reliable Data Transfer (RDT) para asegurar la correcta entrega de datos aún en presencia de pérdidas, duplicados o errores.

Objetivos del Trabajo Práctico

- Desarrollar una aplicación cliente-servidor que permita upload y download de archivos.
- Implementar dos variantes de RDT sobre UDP: Stop & Wait y Selective Repeat.
- Validar su funcionamiento bajo condiciones de pérdida de paquetes simuladas en Mininet.
- Comparar el rendimiento de ambos protocolos en distintos escenarios.

Alcance y Limitaciones

- Se soportan archivos binarios de hasta al menos 5 MB, con transferencias menores a 2 minutos.
- El servidor maneja múltiples clientes concurrentes.
- Se simulan pérdidas del 10 % con Mininet.
- No se implementó control de congestión

Estructura del Informe

El presente documento se organiza en:

I. arquitectura del sistema

II. detalles de implementación de ambos protocolos

III. análisis de performance

IV. respuestas a preguntas teóricas

V. dificultades encontradas y conclusiones.

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.

3. Implementación

3.1. 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

3.2. 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)

3.3. 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

3.4. Flujos del Protocolo

3.4.1. Handshake (upload)

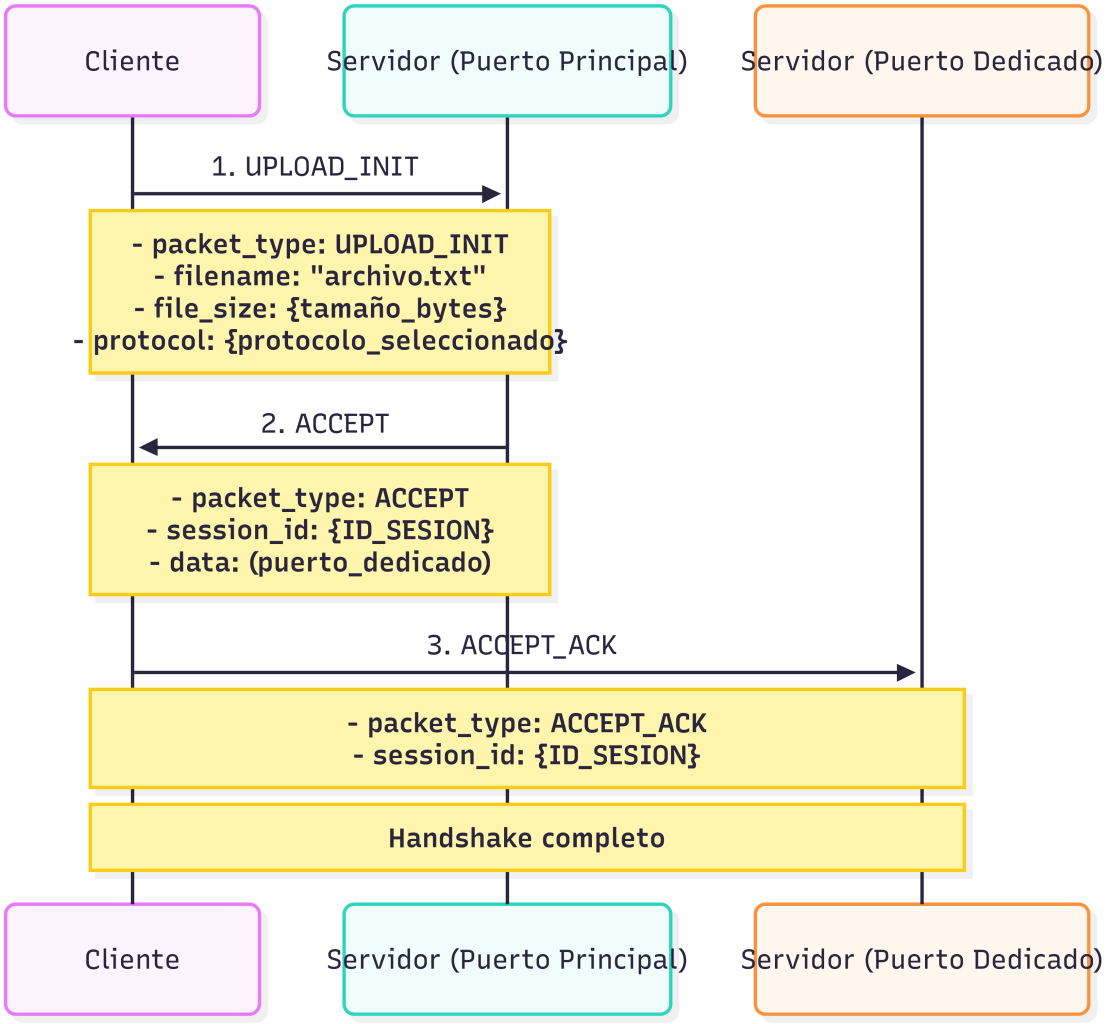

Figura 1: Handshake inicial para subida de archivo

3.4.2. Handshake (download)

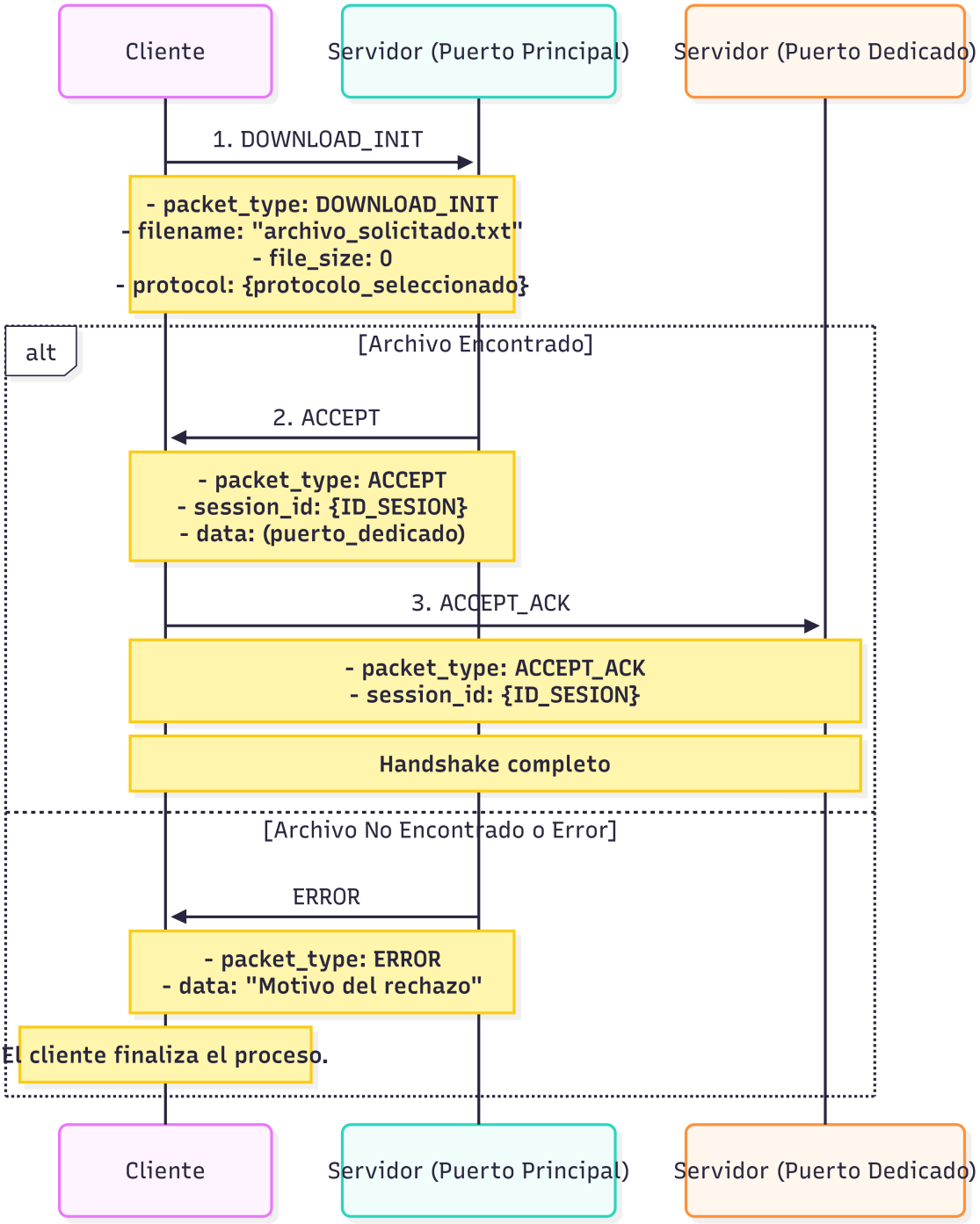

Figura 2: Handshake inicial para descarga de archivo

3.4.3. Cierre de conexion

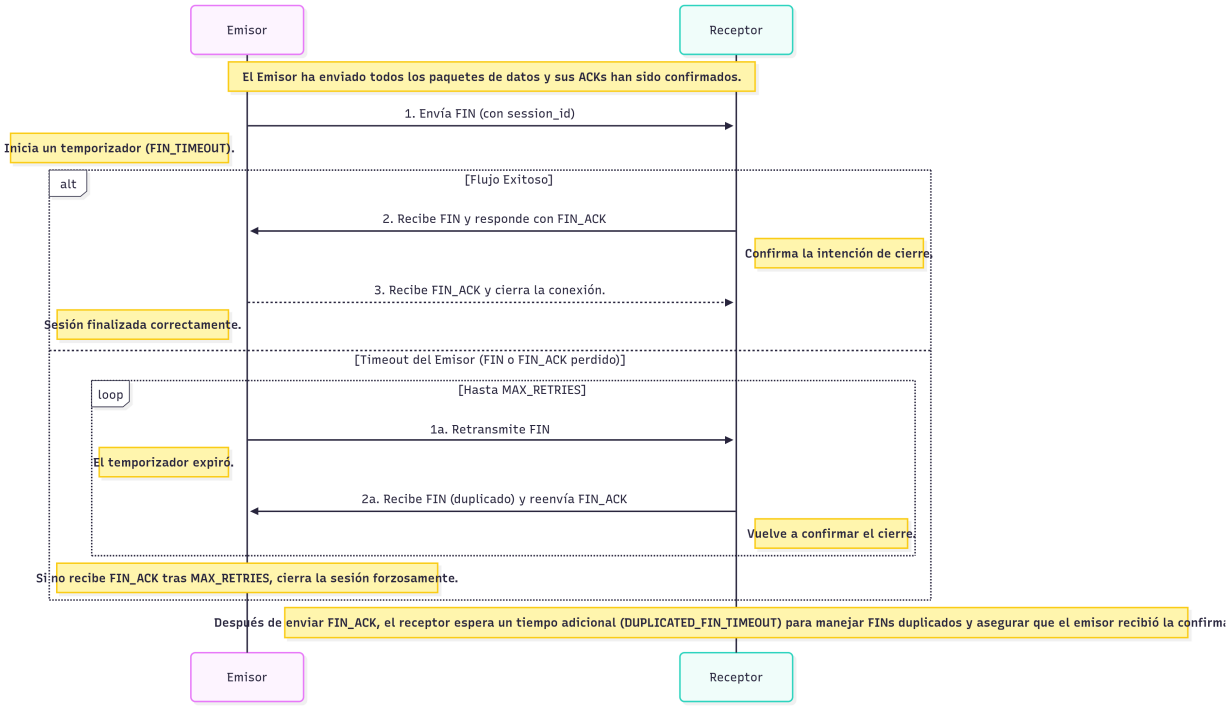

Figura 3: Subida / Descarga de archivo con Selective Repeat

3.4.4. 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.

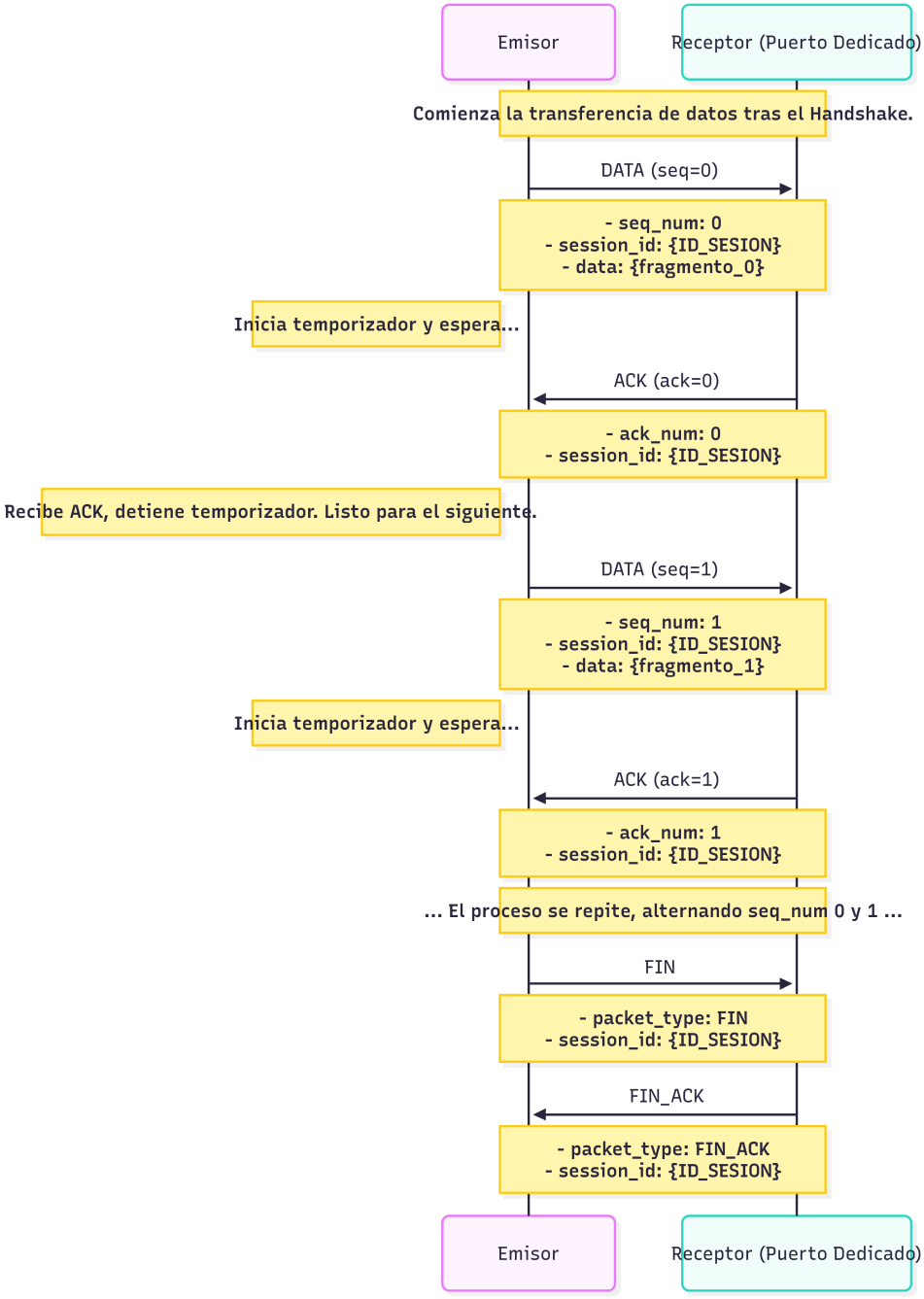

Figura 4: Subida / Descarga de archivo con Stop & Wait

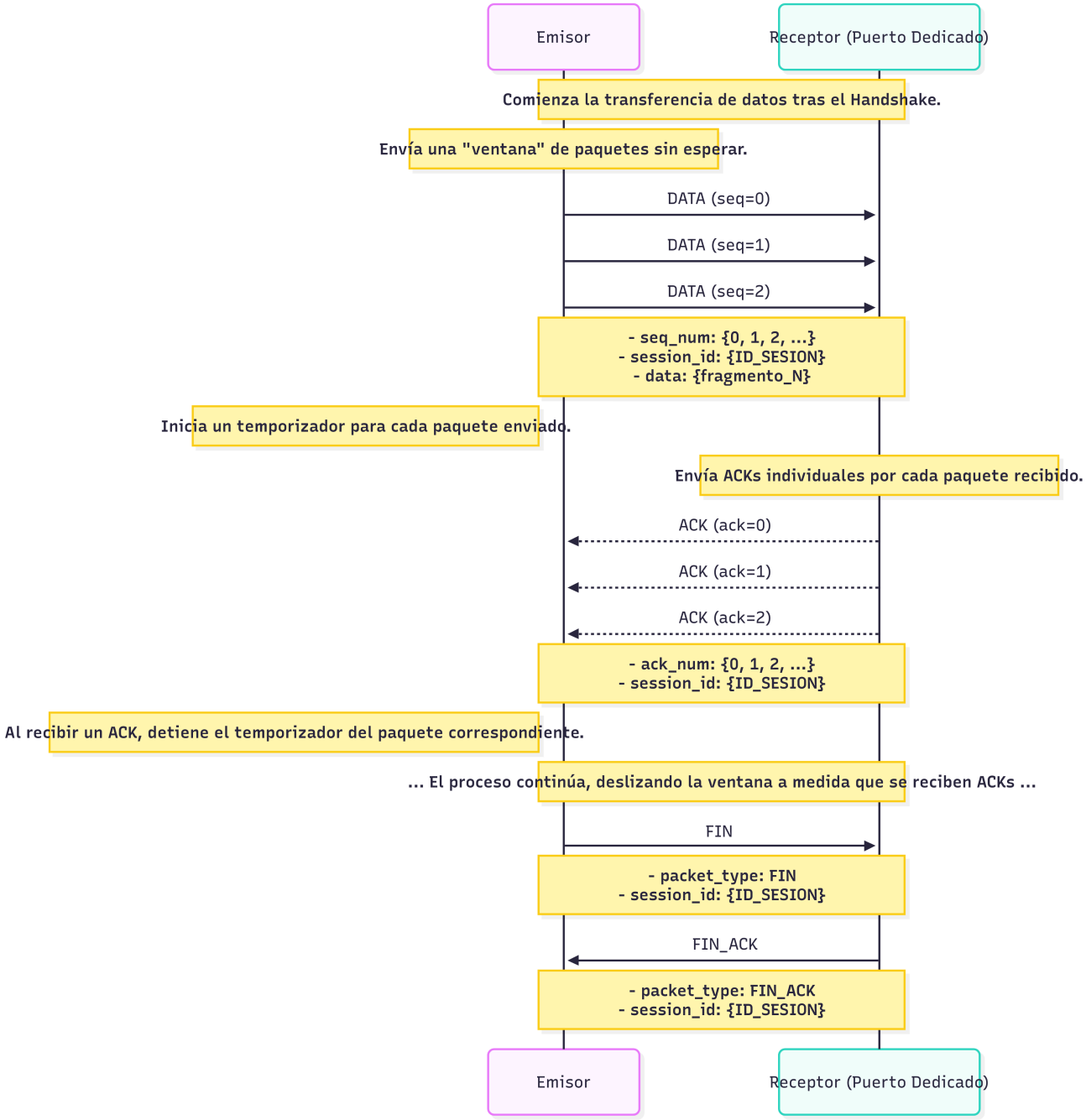

Figura 5: Subida / Descarga de archivo con Selective Repeat

3.5. Manejo de errores

El sistema implementa múltiples capas de manejo de errores:

1. **Validación de Checksums:**

Cada paquete incluye un checksum simple (suma de los bytes del payload, metadatos y encabezado). El receptor valida la integridad recalculando el checksum y comparándolo. En caso de que el paquete este corrupto, se descarta y no se envía confirmación, forzando retransmisión por timeout.

II. **Timeouts y Retransmisiones:**
Cada emisor asocia un temporizador a los paquetes enviados, si no se recibe el ACK correspondiente dentro del tiempo configurado, el paquete se retransmite asegurando la entrega aún bajo pérdidas del 10 % simuladas en Mininet.

III. **Control de duplicados:**
Stop & Wait: gracias al uso de números de secuencia binarios (0/1), el receptor distingue duplicados fácilmente.
Selective Repeat: mantiene una ventana deslizante con buffer y descarta paquetes ya recibidos fuera de ventana.

IV. **Manejo de paquetes fuera de orden:**
Stop & Wait: no admite desorden, cualquier paquete inesperado se ignora.
Selective Repeat: almacena los paquetes válidos recibidos fuera de orden en el buffer, enviando igualmente ACKs, hasta que pueda reconstruir el flujo completo.

V. **Mensajes de control específicos:**
ERROR: permite rechazar solicitudes inválidas (por ejemplo, archivo inexistente o en uso)
FIN / FIN-ACK: asegura el cierre limpio de la sesión, evitando que un corte abrupto deje la conexión en estado inconsistente.

A continuación, se presentan diagramas de secuencia que ilustran lo explicado en algunos de los items anteriores:

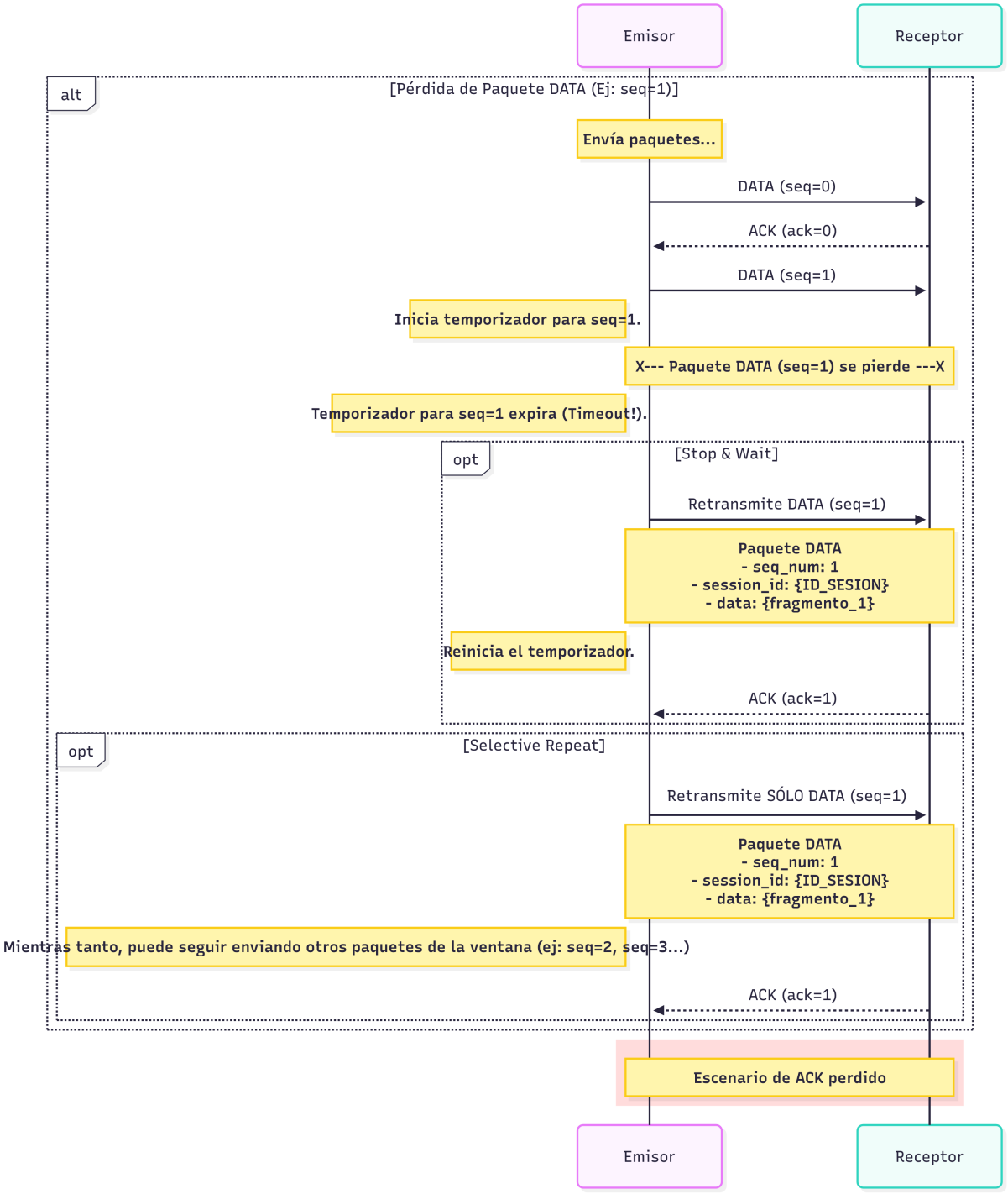

Figura 6: Retramision de paquetes por timeout.

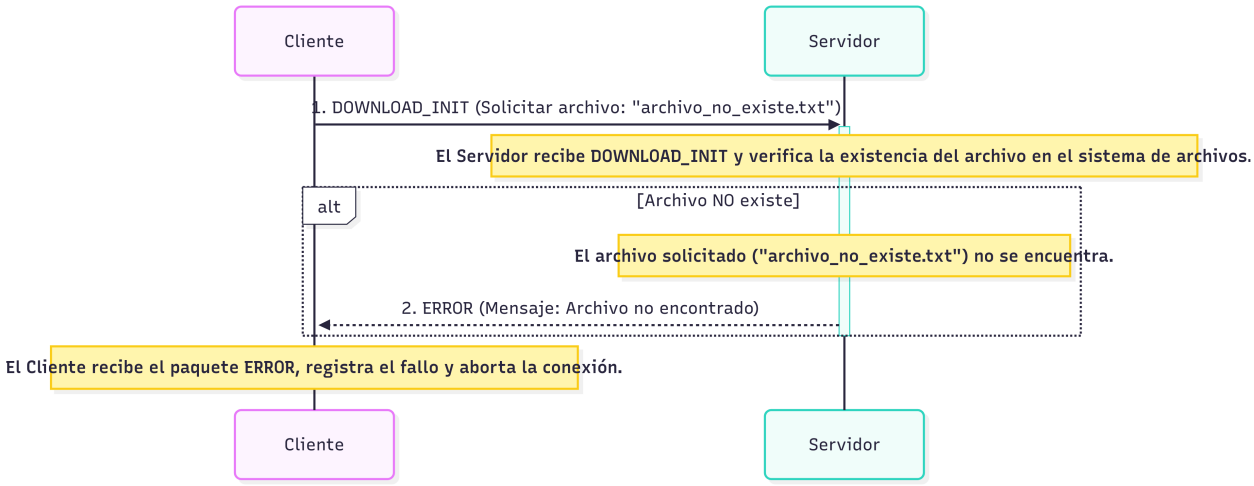

Figura 7: Cierre de conexión y envio de error por solicitud de archivo inexistente.

3.6. Manejo de Concurrencia

3.6.1. 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:

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.

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.

3.6.2. Gestión de Recursos Compartidos

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

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

3.6.3. Puertos Dedicados

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

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

3.6.4. 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

3.6.5. 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 8: Ciclo de vida de una sesión concurrente

3.7. Stop & Wait

Stop & Wait es el protocolo RDT más simple implementado en este trabajo. Su funcionamiento se basa en el envío secuencial de paquetes: el emisor transmite un paquete y espera la confirmación (ACK) antes de enviar el siguiente.

3.7.1. Principio de funcionamiento

El protocolo utiliza números de secuencia alternantes (0 y 1) para distinguir entre paquetes consecutivos y detectar duplicados. El emisor mantiene una variable `seq_num` que alterna entre 0 y 1 con cada paquete enviado exitosamente, mientras que el receptor mantiene una variable `expected_seq` que indica el número de secuencia del próximo paquete esperado.

El flujo básico es:

1. Emisor envía paquete con `seq_num = n`
2. Emisor inicia timer y espera ACK
3. Receptor recibe paquete, verifica checksum y número de secuencia
4. Si el paquete es correcto y tiene el `seq_num` esperado:
   - Receptor acepta los datos
   - Receptor envía ACK con `ack_num = n`
   - Receptor incrementa `expected_seq`
5. Si el paquete es duplicado o corrupto:
   - Receptor descarta los datos
   - Receptor envía ACK del paquete anterior (ACK duplicado)
6. Emisor recibe ACK y avanza al siguiente paquete
7. Si timer expira antes de recibir ACK, el emisor retransmite

3.7.2. Implementación del emisor

La clase `RDTSender` en `stop_wait.py` implementa el emisor del protocolo Stop & Wait. Sus componentes principales son:

Variables de estado:

- `seq_num`: Número de secuencia actual (0 o 1)
- `session_id`: Identificador de sesión asignado por el servidor
- `socket`: Socket UDP para transmisión
- `dest_addr`: Dirección y puerto del receptor

Preparación de paquetes:
El método `_prepare_packets()` lee el archivo fuente y lo fragmenta en paquetes DATA.

Envío confiable de paquetes:
El método `_send_packet_reliable()` implementa el núcleo del protocolo. Para cada paquete:

1. Se envía el paquete al receptor mediante `socket.sendto()`
2. Se inicia un timer implícito configurando `socket.settimeout(SW_TIMEOUT)` con un timeout de 50ms
3. Se bloquea esperando el ACK con `socket.recvfrom()`
4. Si se recibe un ACK válido (tipo ACK, `ack_num` correcto, checksum válido), se retorna éxito
5. Si el timeout expira sin recibir ACK, se reintenta hasta un máximo de 20 intentos
6. Si se recibe un ACK inválido o duplicado, se reintenta inmediatamente

La retransmisión automática por timeout es el mecanismo principal de recuperación ante pérdida de paquetes o ACKs. El timeout de 50ms fue ajustado empíricamente para balancear entre latencia y robustez ante pérdidas.

Cierre de sesión:
El método `_send_fin()` maneja el cierre graceful de la conexión enviando un paquete FIN y esperando el FIN_ACK correspondiente. Utiliza un timeout mayor (1 segundo) y reintenta hasta 20 veces antes de abortar. Este handshake de cierre garantiza que el receptor haya procesado todos los datos antes de liberar recursos.

3.7.3. Implementación del receptor

La clase `RDTReceiver` en `stop_wait.py` implementa el receptor del protocolo. Su responsabilidad es recibir paquetes en orden, filtrar duplicados y confirmar recepciones.
Variables de estado:

- `expected_seq`: Número de secuencia esperado (0 o 1)
- `socket`: Socket UDP para recepción

Procesamiento de paquetes:
El método `receive_file_with_first_packet()` maneja el primer paquete DATA (ya recibido por la capa de sesión) y delega al método `_continue_receiving()` para el resto de la transferencia.
Para cada paquete recibido:

1. Se verifica el checksum usando `packet.verify_checksum()`
2. Si el checksum es inválido, se descarta el paquete
3. Se compara `packet.seq_num` con `expected_seq`
4. Si coinciden:
    - Se aceptan los datos y se agregan a un queue para escritura asíncrona a disco

- Se envía un ACK con `ack_num = packet.seq_num`
- Se alterna `expected_seq` ($0 \rightarrow 1$ o $1 \rightarrow 0$)

5. Si no coinciden (paquete duplicado):
   - Se descartan los datos (para evitar duplicación)
   - Se reenvía el ACK del paquete (ACK duplicado)

Detección de fin de transferencia:
El receptor permanece en un loop esperando paquetes hasta recibir un paquete de tipo FIN. Al detectar FIN:

- Se señaliza al writer thread que la transferencia terminó (enviando `None` al queue)
- Se invoca `_handle_fin()` para enviar FIN_ACK y manejar posibles retransmisiones del FIN
- Se retorna el resultado de la transferencia (éxito o fallo)

El método `_handle_fin()` es crítico para evitar cierres prematuros. Después de enviar el FIN_ACK inicial, el receptor permanece escuchando durante 5 segundos por posibles retransmisiones del FIN (que ocurren si el emisor no recibió el FIN_ACK). Si detecta un FIN duplicado, reenvía el FIN_ACK.

3.7.4. Manejo de errores y casos especiales

Corrupción de paquetes: Los paquetes con checksum inválido se descartan. El emisor detectará la ausencia de ACK por timeout y retransmitirá.

Pérdida de paquetes DATA: Si un paquete DATA se pierde en la red, el timer del emisor expirará y retransmitirá el paquete. El receptor, al recibir el paquete retransmitido, lo procesará normalmente.

Pérdida de ACKs: Si un ACK se pierde, el emisor retransmitirá el paquete por timeout. El receptor recibirá un paquete duplicado (con `seq_num` anterior), lo detectará mediante `expected_seq`, descartará los datos y reenviará el ACK.

Paquetes fuera de orden: Aunque UDP puede entregar paquetes fuera de orden, Stop & Wait garantiza orden mediante su naturaleza secuencial. Solo un paquete está en tránsito a la vez, por lo que no puede haber overtaking.

Señalización de shutdown graceful: El protocolo verifica periódicamente la variable global `_shutdown_requested` (configurada por SIGINT/SIGTERM). Si se detecta, tanto el emisor como el receptor abortan la transferencia ordenadamente.

3.8. Selective Repeat

Selective Repeat es un protocolo de ventana deslizante que permite tener múltiples paquetes en tránsito simultáneamente, logrando un throughput significativamente mayor que Stop & Wait. Selective Repeat solo retransmite los paquetes que efectivamente se perdieron, optimizando el uso del ancho de banda.

3.8.1. Principio de funcionamiento

El protocolo se basa en el concepto de ventana deslizante: el emisor puede transmitir hasta `WINDOW_SIZE` paquetes sin esperar confirmación. Cada paquete tiene su propio timer independiente, y los ACKs son selectivos (confirman paquetes individuales, no acumulativos).

Variables fundamentales:

- `send_base`: Índice del paquete más antiguo sin confirmar en el emisor
- `nextseqnum`: Próximo número de secuencia disponible para envío
- `rcv_base`: Índice del paquete más antiguo esperado en el receptor
- `WINDOW_SIZE`: Tamaño máximo de la ventana (20 paquetes en esta implementación)

El emisor mantiene una ventana de envío `[send_base, send_base + WINDOW_SIZE)` donde puede enviar paquetes. Cuando un paquete es confirmado, si coincide con `send_base`, la ventana se desliza para incluir nuevos paquetes. El receptor mantiene una ventana de recepción `[rcv_base, rcv_base + WINDOW_SIZE)` donde puede bufferar paquetes fuera de orden.

3.8.2. Implementación del emisor

La clase `SelectiveRepeatSender` en `selective_repeat.py` implementa el emisor con las siguientes estructuras:

Variables de estado:

- `send_base`: Primer paquete sin ACK
- `nextseqnum`: Siguiente paquete a enviar
- `send_window`: Diccionario `{seq_num: (packet, timer)}` con paquetes en tránsito y sus timers
- `window_size`: Tamaño de ventana (20 por defecto)
- `session_id`: Identificador de sesión

Algoritmo principal de envío:

El método `_send_packets()` implementa el bucle principal del protocolo, es decir, la lógica de envío en el protocolo con ventana deslizante (Selective Repeat).

El bucle principal se ejecuta mientras existan paquetes por transmitir o aún haya paquetes pendientes de confirmación. Para cada iteración, se envían nuevos paquetes dentro de los límites de la ventana, asignándoles un número de secuencia calculado con *wrap-around* sobre un espacio finito de 256 valores, lo que permite reutilizar números sin perder consistencia.

Cada paquete enviado se asocia con un temporizador individual para detectar pérdidas y posibilitar retransmisiones en caso de que el ACK correspondiente no llegue a tiempo. En paralelo, se procesan los ACKs recibidos y se gestionan los posibles *timeouts*, asegurando que la transferencia avance de manera confiable y eficiente.

Este diseño permite pipelining: múltiples paquetes se envían sin esperar confirmación, maximizando el uso del canal.

Manejo de ACKs y timeouts:

El método `_handle_acks_and_timeouts()` ejecuta dos tareas en cada iteración:

1. **Recepción de ACKs** (no bloqueante con timeout de 100ms):
   - Si se recibe un ACK válido para un paquete en `send_window`, se elimina ese paquete
   - Si el ACK confirma `send_base`, se desliza la ventana: `send_base` avanza hasta el primer paquete sin confirmar
   - Los ACKs fuera de ventana o duplicados se ignoran
2. **Verificación de timers expirados**:
   - Se itera sobre todos los paquetes en `send_window`
   - Para cada paquete cuyo timer haya expirado (>200ms), se retransmite selectivamente
   - Se reinicia el timer del paquete retransmitido

Esta lógica garantiza que solo se retransmiten los paquetes perdidos, y no toda la ventana.

Timers independientes por paquete:

Cada paquete enviado mantiene asociado un objeto `PacketTimer`, un temporizador propio que permite medir de forma precisa su tiempo de vida en tránsito. Gracias a este mecanismo, el emisor puede detectar si un paquete específico no recibió su confirmación dentro del intervalo configurado.

El uso de timers individuales aporta granularidad a la retransmisión: si algunos paquetes de la ventana (por ejemplo, los números 5 y 8) se pierden, únicamente esos se reenvían al expirar sus temporizadores, mientras que los demás continúan su curso normal. De esta forma se evita la retransmisión innecesaria de bloques completos, optimizando tanto el rendimiento como la eficiencia del protocolo.

Manejo del wrap-around de números de secuencia:

El protocolo utiliza números de secuencia de 8 bits (0-255), lo que implica que deben reiniciarse al llegar al valor máximo. Para resolver esto, se aplica un mecanismo de *wrap-around*, asegurando que la numeración vuelva a comenzar desde cero de manera controlada.

Mediante este enfoque, es posible identificar de forma correcta los paquetes incluso en transferencias largas que superan los 256 segmentos. Al recibir un `ACK`, el protocolo realiza la comparación considerando este reinicio, lo que permite mantener la consistencia de la ventana de envío y asegurar que cada confirmación se asocie al paquete correcto.

3.8.3. Implementación del receptor

La clase `SelectiveRepeatReceiver` implementa el receptor con buffering para paquetes fuera de orden:

Variables de estado:

- `rcv_base`: Número de secuencia del próximo paquete esperado en orden
- `rcv_window`: Diccionario `{seq_num: packet}` que bufferea paquetes recibidos dentro de la ventana
- `received_data`: Acumulador de datos entregados en orden a la aplicación
- `window_size`: Tamaño de ventana de recepción (20)

Procesamiento de paquetes:
El método `_process_packet()` maneja cada paquete recibido según su número de secuencia:

1. **Paquete dentro de la ventana de recepción** `[rcv_base, rcv_base + window_size)`:
   - Si es un paquete nuevo (no está en `rcv_window`), se bufferea
   - Se envía ACK selectivo con `ack_num = packet.seq_num`
   - Se intenta entregar paquetes consecutivos: mientras `rcv_base` esté en `rcv_window`:
     - Se extrae el paquete del buffer
     - Se agregan sus datos al queue de escritura
     - Se incrementa `rcv_base`
2. **Paquete antes de la ventana** (`seq_num <rcv_base`):
   - Es un paquete duplicado (ya entregado)
   - Se reenvía ACK (ACK duplicado)
   - Se descartan los datos
3. **Paquete después de la ventana** (`seq_num >= rcv_base + window_size`):
   - Paquete demasiado adelantado, fuera de capacidad de buffering
   - Se envía ACK pero se descarta el paquete

Entrega en orden a la aplicación:
El mecanismo de buffering permite recibir paquetes fuera de orden pero entregarlos secuencialmente:

```
# Ejemplo: recibimos paquetes 0, 2, 1, 3 (fuera de orden)
Paquete 0 llega: rcv_base=0
    -> bufferea en rcv_window[0]
    -> entrega paquete 0, rcv_base avanza a 1

Paquete 2 llega: rcv_base=1
    -> bufferea en rcv_window[2]
    -> no puede entregar (falta paquete 1), rcv_base=1
```

```
Paquete 1 llega: rcv_base=1
    -> bufferea en rcv_window[1]
    -> entrega paquete 1, rcv_base avanza a 2
    -> encuentra paquete 2 ya buffereado
    -> entrega paquete 2, rcv_base avanza a 3

Paquete 3 llega: rcv_base=3
    -> bufferea en rcv_window[3]
    -> entrega paquete 3, rcv_base avanza a 4
```

De esta manera, los paquetes se reciben en cualquier orden, pero se entregan consecutivamente a la capa de aplicación.

Manejo de wrap-around en la ventana:

El método `_is_in_window()` determina si un número de secuencia está dentro de la ventana considerando el wrap-around.

La verificación de si un número de secuencia pertenece a la ventana activa debe contemplar el caso en que esta cruce el límite de 255 a 0. Para ello, el protocolo implementa una lógica de *wrap-around* que distingue entre el caso normal (sin cruce) y el caso donde la ventana se extiende de los valores altos hacia los bajos del rango. De esta forma, la ventana puede abarcar intervalos como $[240, 255] \cup [0, 20]$, garantizando que la transmisión y la recepción funcionen correctamente incluso en transferencias extensas. Esto asegura que el control de flujo sea continuo y robusto frente al límite de numeración de 8 bits.

4. Pruebas

4.1. 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.

4.2. 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.

4.3. 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* que garantizan una perdida de paquetes del 5 y 10 %.

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.

4.4. Resultados obtenidos

Los resultados obtenidos para los experimentos mencionados se pueden analizar según las siguientes métricas:

Tiempo total de transferencia: Para la transferencia de archivos pequeños, Stop & Wait resultó significativamente más rápido que Selective Repeat. Si bien para tamaños más grandes ambos protocolos se equipararon considerablemente, incluso con archivos de 10 MB la diferencia se mantuvo en un rango de 5 segundos.

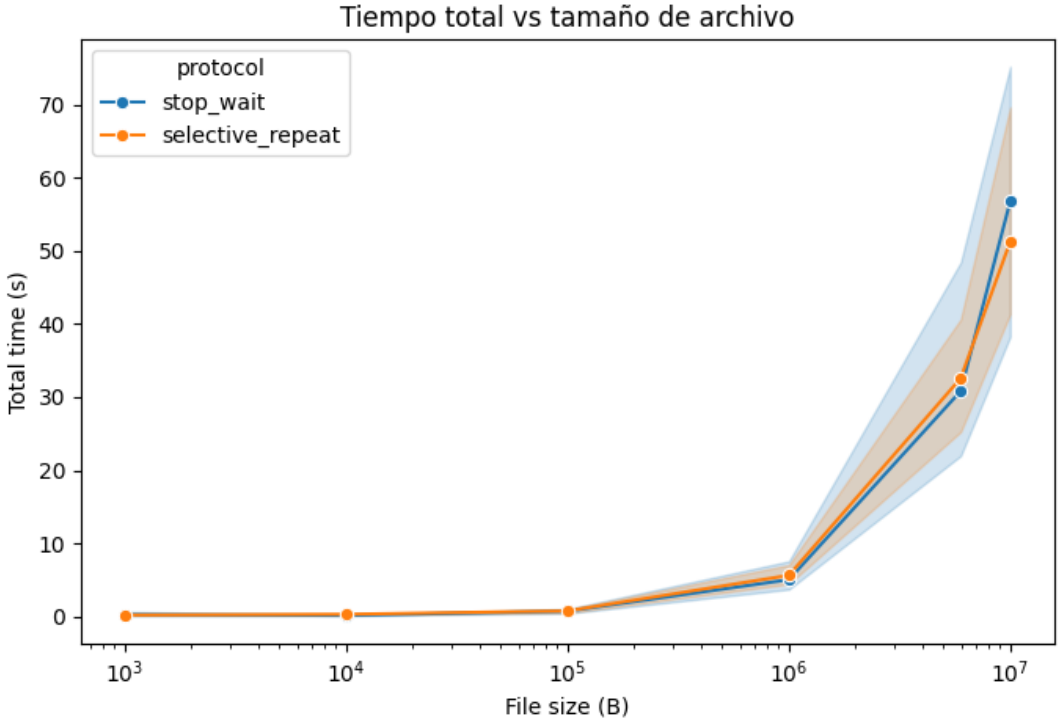

Figura 9: Comparación del tiempo total de transferencia entre Stop & Wait y Selective Repeat

Throughput efectivo: Consistentemente con la métrica anterior, y probablemente responsable de la misma, el throughput promedio para Selective Repeat fue inferior en la mayoría de los casos. Nuevamente, para archivos más grandes Selective Repeat se acerca e incluso puede llegar a superar a Stop & Wait, aunque no de manera significativa.

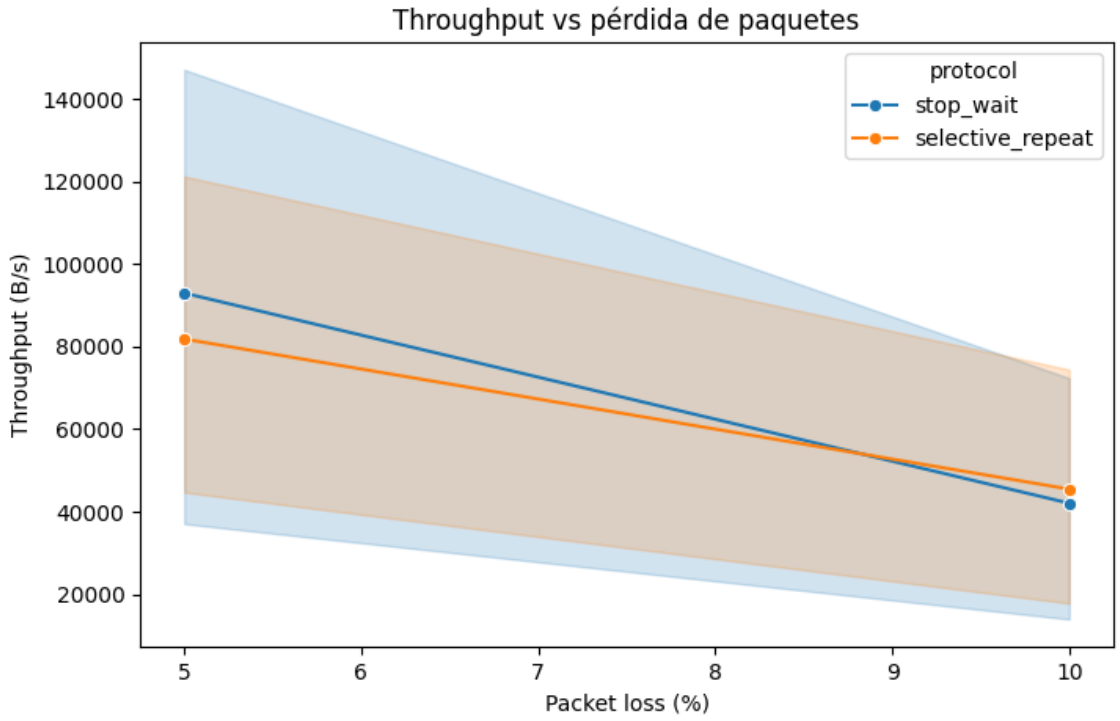

Figura 10: Comparación del throughput efectivo entre Stop & Wait y Selective Repeat

RTT y Jitter: Esta fue la métrica más favorable para Stop & Wait, ya que todos los experimentos realizados con este protocolo obtuvieron menor latencia y mayor estabilidad que Selective Repeat, el cual mostró amplias variaciones entre un RTT y otro.

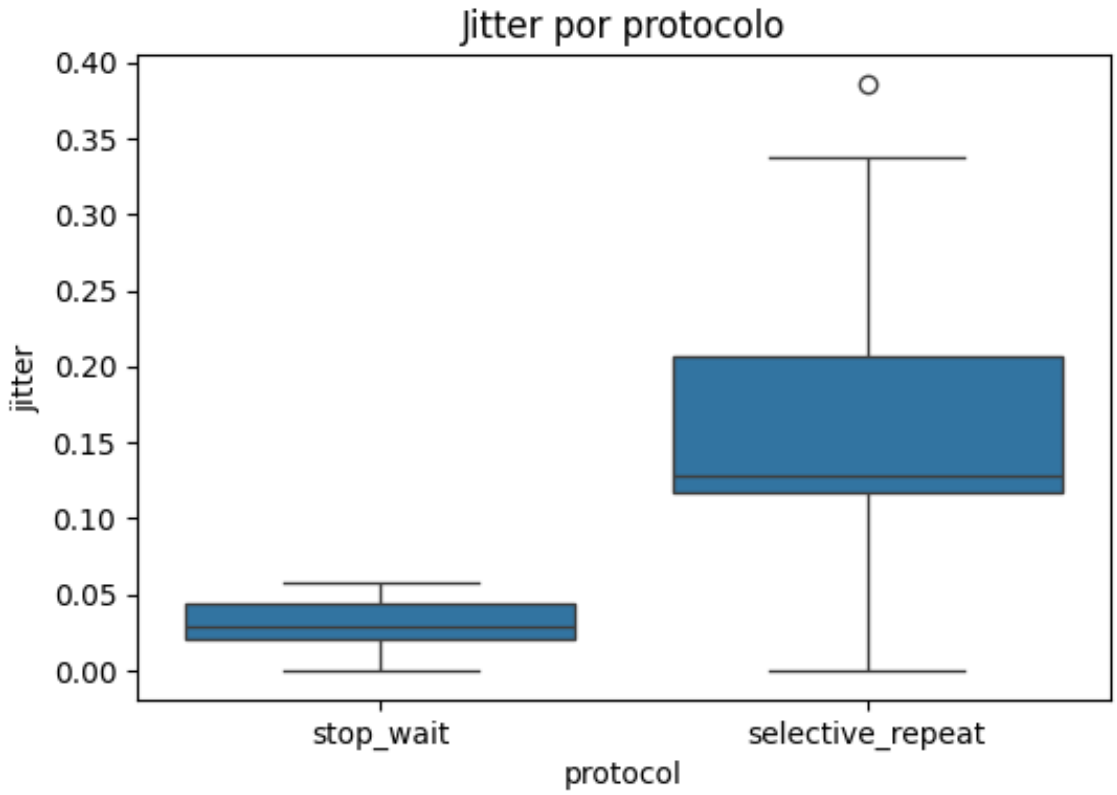

Figura 11: Comparación del Jitter entre Stop & Wait y Selective Repeat

Utilización de la ventana en Selective Repeat: Es interesante resaltar este parámetro ya que probablemente sea el responsable del rendimiento inferior de Selective Repeat. Para tamaños de archivos pequeños, el uso no representa ni el 10 % del tamaño total de la ventana, e incluso para archivos de 10 MB el uso alcanza apenas un 50 % de su capacidad total.

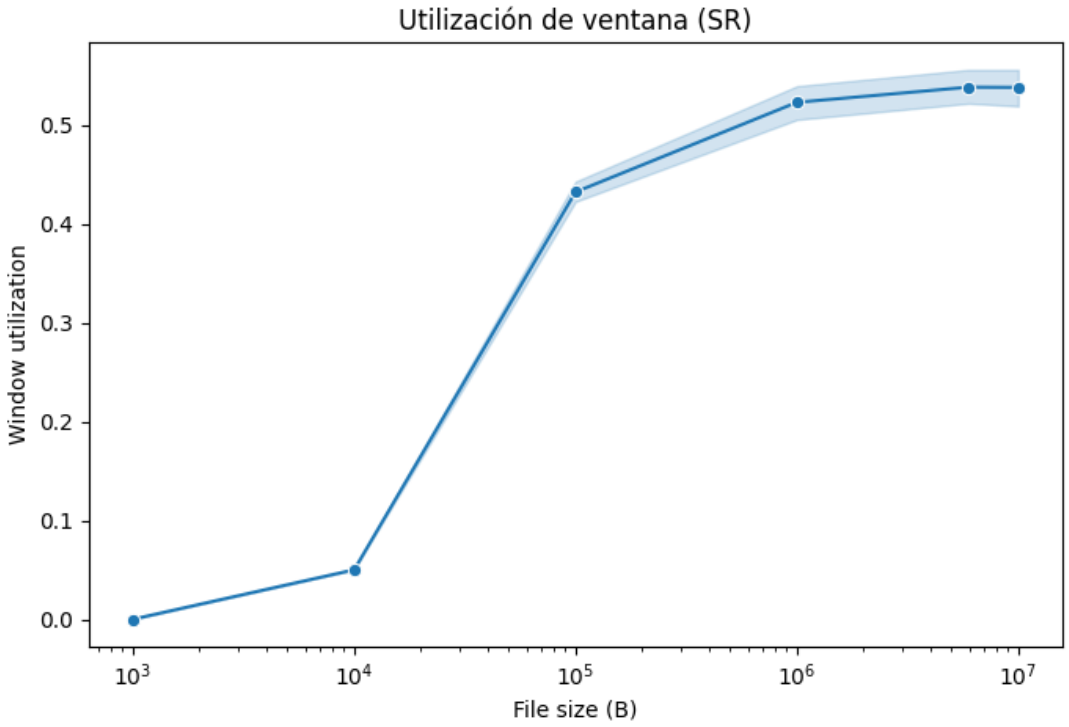

Figura 12: Uso promedio de la ventana de envío en Selective Repeat

El objetivo de estas pruebas era comparar ambos protocolos en condiciones similares. Sin embargo, los resultados obtenidos muestran que Stop & Wait supera a Selective Repeat en casi todas las métricas evaluadas, incluso en escenarios donde se esperaba que Selective Repeat tuviera un mejor desempeño debido a su capacidad para manejar múltiples paquetes en tránsito. Esto puede atribuirse a varios factores, entre ellos la implementación específica de los protocolos, la configuración de la red simulada y las características inherentes de cada protocolo. Es probable que sea necesario ajustar ciertos parámetros de Selective Repeat, como el tamaño de la ventana o los tiempos de espera, para optimizar su rendimiento en este entorno particular. Además, la sobrecarga adicional que implica la gestión de múltiples paquetes y acuses de recibo en Selective Repeat puede estar afectando negativamente su eficiencia en comparación con la simplicidad de Stop & Wait.

5. Preguntas a responder

1. **Describa la arquitectura Cliente-Servidor**

   La arquitectura Cliente-Servidor es un paradigma predominante en las aplicaciones de red modernas. En esta arquitectura, existe un host siempre activo, denominado servidor, que atiende las solicitudes de muchos otros hosts, que son los clientes.

   Las características clave de la arquitectura Cliente-Servidor incluyen:

   - Servidor siempre activo: El servidor está constantemente disponible para atender solicitudes.
   - Dirección conocida: El servidor posee una dirección fija y bien conocida, denominada dirección IP, lo que permite a los clientes contactarlo enviando un paquete a esa dirección.
   - Comunicación indirecta entre clientes: Los clientes generalmente no se comunican directamente entre sí.
   - Servidor Virtual: En aplicaciones populares, cuando un único host servidor es incapaz de responder a todas las solicitudes, se utiliza un centro de datos que alberga un gran número de hosts para crear un servidor virtual de gran capacidad (como en los motores de búsqueda o redes sociales).

   En el contexto de una sesión de comunicación entre dos procesos, el proceso cliente es el que inicia la comunicación (el que se pone en contacto inicialmente con el otro proceso), mientras que el proceso servidor es el que espera a ser contactado para comenzar la sesión.

2. **¿Cuál es la función de un protocolo de capa de aplicación?**

   La función principal de un protocolo de la capa de aplicación es definir cómo los procesos de una aplicación, que se ejecutan en distintos sistemas terminales, intercambian mensajes entre sí.

   Un protocolo de red es un conjunto definido de reglas y procedimientos que determinan cómo se transmiten los datos en las redes de computadoras.

   Específicamente, un protocolo de la capa de aplicación define los siguientes elementos:

   - El formato y el orden de los mensajes intercambiados entre dos o más entidades que se comunican. Esto incluye los mensajes de petición y respuesta.
   - La sintaxis de los diversos tipos de mensajes (los campos que componen el mensaje y cómo se delimitan).
   - La semántica de los campos (el significado de la información contenida en los campos).
   - Las acciones tomadas al producirse la transmisión y/o recepción de un mensaje u otro evento.
   - Las reglas para determinar cuándo y cómo un proceso envía mensajes y responde a los mismos.

Los protocolos de la capa de aplicación (como HTTP y SMTP) se implementan casi siempre por software en los sistemas terminales.

3. **Detalle el protocolo de aplicación desarrollado en este trabajo.**

   El protocolo de aplicación desarrollado es un protocolo de transferencia de archivos bidireccional (upload y download) que opera sobre UDP e implementa confiabilidad mediante mecanismos de RDT (Reliable Data Transfer). El protocolo está diseñado para soportar transferencias concurrentes y ofrece dos estrategias de recuperación de errores: Stop & Wait y Selective Repeat.

Características principales del protocolo

Modelo de comunicación:

- Arquitectura cliente-servidor con roles intercambiables (el servidor puede actuar como emisor en operaciones de descarga)
- Protocolo basado en sesiones con identificadores únicos
- Cada transferencia se maneja en un puerto UDP dedicado para permitir concurrencia
- Handshake inicial para negociar parámetros de la transferencia
- Cierre graceful con confirmación bidireccional

Formato de mensajes:

Todos los mensajes del protocolo siguen una estructura uniforme de header fijo de 14 bytes seguido de un payload de longitud variable:

- `seq_num` (1 byte): Número de secuencia para ordenamiento de paquetes DATA y detección de duplicados
- `checksum` (1 byte): Suma de verificación para detección de errores de transmisión
- `ack_num` (1 byte): Número de reconocimiento en paquetes ACK
- `payload_length` (4 bytes): Longitud del payload en bytes
- `file_size` (4 bytes): Tamaño total del archivo (utilizado en INIT)
- `packet_type` (1 byte): Tipo de paquete (DATA, ACK, INIT, ACCEPT, FIN, ERROR)
- `protocol` (1 byte): Protocolo de recuperación de errores seleccionado (1=Stop&Wait, 2=Selective Repeat)
- `session_id` (1 byte): Identificador único de sesión asignado por el servidor (1-255)

Flujo del protocolo

1. Establecimiento de conexión (Handshake):

El cliente inicia la transferencia enviando un paquete INIT al puerto principal del servidor (49153) que contiene:

- Nombre del archivo en el payload
- Tamaño del archivo (`file_size >0` para upload, `file_size = 0` para download)
- Protocolo de recuperación de errores deseado (`protocol`)
- `session_id = 0` (aún no asignado)

El servidor, al recibir el INIT:

- Valida la solicitud (existencia del archivo en caso de download, capacidad disponible)
- Genera un `session_id` único (1-255)
- Crea un socket UDP dedicado en un puerto dinámico
- Responde con un paquete ACCEPT que contiene el `session_id` asignado y el puerto dedicado en el payload
- Crea un thread dedicado para manejar la transferencia

El cliente, al recibir el ACCEPT:

- Extrae el `session_id` y el puerto dedicado
- Reconecta su socket al puerto dedicado del servidor
- Procede a la fase de transferencia de datos

2. Transferencia de datos:

La transferencia de datos utiliza paquetes DATA con payload de hasta 4096 bytes (8192 bytes en Stop & Wait para optimizar throughput). El emisor fragmenta el archivo y transmite los fragmentos según el protocolo seleccionado:

- **Stop & Wait**: Envía un paquete y espera su ACK antes de enviar el siguiente. Usa números de secuencia alternantes (0, 1) para detección de duplicados.
- **Selective Repeat**: Mantiene una ventana deslizante de hasta 20 paquetes en tránsito simultáneamente. Los ACKs son selectivos (confirman paquetes individuales) y solo se retransmiten los paquetes cuyo timer expire.

El receptor:

- Verifica el checksum de cada paquete recibido
- En Stop & Wait: Acepta solo el paquete con el número de secuencia esperado, descarta duplicados

- En Selective Repeat: Bufferea paquetes fuera de orden y los entrega en secuencia cuando es posible
- Envía ACKs por cada paquete recibido correctamente
- Escribe los datos recibidos a disco mediante un thread dedicado (writer thread)

3. Cierre de conexión:

Una vez transmitidos todos los paquetes DATA, el emisor:

- Envía un paquete FIN con el `session_id` de la sesión
- Espera un paquete FIN_ACK del receptor con timeout y retransmisión
- Mantiene el estado para reenviar FIN_ACK ante duplicados del FIN (mecanismo para evitar cierre prematuro)

El receptor:

- Al recibir FIN, confirma que todos los datos fueron escritos
- Envía FIN_ACK y permanece escuchando por posibles retransmisiones del FIN durante un período de timeout
- Cierra el socket y libera recursos al expirar el timeout

Mecanismos de confiabilidad

El protocolo implementa varios mecanismos para garantizar transferencia confiable sobre UDP:

Detección de errores:

- Checksum en cada paquete (suma de todos los bytes del header y payload)
- Validación de checksums en recepción, descartando paquetes corruptos

Recuperación de pérdidas:

- Timeouts configurables por paquete (200ms para Selective Repeat, 50ms para Stop & Wait)
- Retransmisión automática al expirar el timeout
- Límite máximo de reintentos (20 intentos) antes de abortar la transferencia

Control de duplicados y reordenamiento:

- Números de secuencia con wrap-around en 256 (1 byte)
- Stop & Wait: Descarta paquetes duplicados basándose en el número de secuencia esperado
- Selective Repeat: Buffers para reordenar paquetes y entrega en secuencia

Manejo de errores a nivel de aplicación:

- Paquetes ERROR para comunicar condiciones excepcionales (archivo no encontrado, servidor saturado)
- Validación de sesiones (session_id y dirección del cliente)
- Control de capacidad del servidor (máximo 10 transferencias concurrentes)
- Timeouts diferenciados según la fase (handshake: 1s, datos: 50-200ms, FIN: 1s)

Soporte para concurrencia

El protocolo fue diseñado para soportar múltiples transferencias simultáneas:

- **Identificación de sesiones**: Cada transferencia recibe un `session_id` único que identifica todos sus paquetes
- **Aislamiento por puerto**: Cada sesión opera en un puerto UDP dedicado, eliminando la necesidad de demultiplexar en el servidor
- **Estado independiente**: Cada thread de transferencia mantiene su propio estado (ventanas, buffers, timers) sin compartir recursos
- **Sincronización mínima**: Solo el diccionario de sesiones activas requiere sincronización mediante locks

Este diseño permite que el servidor procese transferencias de distintos clientes con diferentes protocolos (Stop & Wait y Selective Repeat) simultáneamente sin interferencias.

4. **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?**

Protocolo UDP (User Datagram Protocol)

Servicios y Características:

- Sin Conexión: No requiere una fase de establecimiento de conexión entre las entidades de la capa de transporte emisora y receptora antes del envío del segmento.
- Servicio no confiable: No ofrece ninguna garantía de que el mensaje llegue al proceso receptor. Los mensajes que llegan pueden hacerlo de manera desordenada.
- Control mínimo de integridad: Proporciona servicios de comprobación de la integridad de los datos mediante un checksum, pero no garantiza la recuperación del error.
- Sin control de flujo: UDP no ofrece control de flujo, lo que implica que los segmentos pueden perderse en el receptor debido al desbordamiento del buffer.

- Sin control de congestión: UDP no incluye un mecanismo de control de congestión. El emisor puede introducir datos en la capa inferior a cualquier velocidad.
- Poca sobrecarga: La cabecera de los paquetes UDP requiere únicamente 8 bytes.

Cuándo es apropiado utilizar UDP:

- Aplicaciones sensibles al tiempo/tolerantes a pérdidas: Cuando las aplicaciones requieren una velocidad mínima de transmisión (rendimiento) y toleran algunas pérdidas de datos, como en telefonía por Internet, videoconferencias en tiempo real y transmisión de vídeo/audio.
- Evitar retardos de establecimiento de conexión: Aplicaciones donde la velocidad es crítica y se desea evitar el establecimiento de conexión de TCP (ej. DNS).
- Mayor control a nivel de aplicación: Para aplicaciones que requieren un control más directo sobre qué datos se envían y cuándo, evitando las restricciones de control de congestión de TCP.

Protocolo TCP (Transmission Control Protocol)

Servicios y Características:

- Orientado a la conexión: Requiere un proceso de establecimiento de conexión (three-way handshake) entre el cliente y el servidor antes de que los mensajes de aplicación comiencen a transmitirse.
- Transferencia de datos confiable (RDT): Garantiza que los datos transmitidos sean entregados al proceso receptor correctamente y en orden, convirtiendo el servicio no confiable de IP en un servicio confiable. Esto se logra mediante mecanismos de control de flujo, números de secuencia, reconocimientos (ACKs) y temporizadores.
- Control de flujo: Proporciona un servicio de adaptación de velocidades para evitar que el emisor desborde el buffer del receptor, utilizando una ventana de recepción (`rwnd`) configurable.
- Control de congestión: Regula la velocidad de transmisión del emisor cuando la red está congestionada, para evitar saturar los enlaces y routers.
- Full-Duplex: La conexión permite que los procesos envíen mensajes entre sí simultáneamente.
- Seguridad (SSL/TLS): Puede mejorarse con SSL (Secure Sockets Layer) para proporcionar servicios de seguridad (confidencialidad, integridad y autenticación).

Cuándo es apropiado utilizar TCP:

- Confiabilidad es crítica: Aplicaciones que requieren que todos los datos lleguen a su destino sin errores ni pérdidas. Por ejemplo: Correo electrónico (SMTP), acceso remoto a terminales (Telnet), Web (HTTP) y transferencia de archivos (FTP).

- Transferencias de datos grandes: Cuando se manejan grandes volúmenes de datos, TCP es preferible debido a su capacidad para gestionar la transmisión confiable y ordenada.
- Aplicaciones que no toleran pérdidas: Aplicaciones donde la pérdida de datos es inaceptable, como en la transferencia de archivos y ciertas aplicaciones web.

6. Dificultades encontradas

Durante el desarrollo del trabajo se presentaron distintos desafíos técnicos que requirieron ajustes y soluciones iterativas. A continuación describimos algunos de ellos:

- **Manejo de timeouts:**
  El manejo de muy agresivos que generaban retransmisiones excesivas y sobrecarga innecesaria.
- **Problemas con la concurrencia del servidor:**
  La gestión de multiples usuarios en el mismo socket rapidamente obligo a crear sockets dedicados para cada sesión y gestionar identificadores únicos para cada cliente para facilitar la identificación y correción de errores.
- **Detección de duplicados y paquetes fuera de orden:**
  en Selective Repeat, se requirió una implementación más robusta de la ventana deslizante y el buffer.
- **Integración con Mininet:**
  La simulación de pérdida de paquetes introducía comportamientos difíciles de depurar.

7. Conclusiones

El desarrollo de este trabajo práctico permitió implementar y evaluar un sistema de transferencia de archivos confiable sobre UDP, explorando las diferencias entre los protocolos Stop & Wait y Selective Repeat. Mientras que el primero ofreció simplicidad a costa de eficiencia y gran desempeño para paquetes pequeños, el segundo demostró un mejor desempeño en la medida que se aumentaba el porcentage de pérdidas de paquetes gracias a su ventana deslizante y retransmisiones selectivas. El diseño modular de la solución y la definición de un protocolo de aplicación propio facilitaron la gestión de errores, el soporte de concurrencia y el cierre seguro de sesiones. En conjunto, la experiencia brindó una comprensión más profunda de los mecanismos de confiabilidad en redes distribuidas y su relevancia práctica para sistemas de comunicación robustos. Los resultados obtenidos subrayan la importancia de elegir el protocolo adecuado según las características de la red y los requisitos de la aplicación, destacando que no existe una solución única para todos los escenarios. En futuras iteraciones, se podrían explorar optimizaciones adicionales, como la adaptación dinámica del tamaño de la ventana en Selective Repeat o la implementación de técnicas avanzadas de control de congestión para mejorar aún más el rendimiento en entornos con alta latencia o pérdida de paquetes.