Introducción a los Sistemas Distribuídos (75.43)

TP N°1: File Transfer

Martin Palazón
Sebastian Martin Pagura
Facultad de Ingeniería, Universidad de Buenos Aires

09 de Octubre de 2025

1. Introducción

En busqueda de demostrar conocimientos sobre el protocolo RDT, se implementó un programa con arquitectura Cliente-Servidor, donde la comunicación entre las partes esta gobernada por un protocolo de aplicacion inspirado en mecanismos que usa TCP. Esta comunicacion podra ser de bajada (Download) o de carga (Upload) de archivos. Se debera garantizar una comuncacion confiable sobre un sistema poco confiable (protocolo UDP), esto se lograra simulando una conexion entre las partes que facilite el reenvio de los paquetes que no lleguen a destino.

Se implementaron dos protocolos de recuperación de errores: Stop & Wait y Selective Repeat, su performance fue comparada bajo distintas condiciones de red.

2. Hipótesis y suposiciones realizadas

La hipótesis central del desarrollo fue que tomando aspectos fundamentales del protocolo TCP se puede lograr transferencia confiable sobre UDP sin replicar toda la complejidad del estándar.

Suposiciones consideradas para el diseño del protocolo:

1. El checksum calculado como hash del payload es suficiente para detectar corrupción de datos durante la transmisión.
2. No se enfrentarán ataques maliciosos que adulteren tanto el paquete como su checksum de forma coordinada.
3. La pérdida de paquetes es aleatoria y no intencional.
4. El entorno de mininet simulará adecuadamente las condiciones de red especificadas.
5. Los timeouts configurados son apropiados para la topología de red simulada.
6. Los archivos a transferir no superan significativamente el tamaño de 5 MB en condiciones normales.

3. Implementación

3.1. Arquitectura de la Aplicación

El sistema sigue una arquitectura cliente-servidor tradicional donde:

- **Servidor**: Es un proceso centralizado que escucha en un puerto UDP, maneja múltiples clientes concurrentemente mediante un ThreadPool, y provee servicios de almacenamiento y recuperación de archivos.
- **Cliente**: Puede consumir dos aplicaciones (upload y download) que se conectan al servidor para realizar operaciones de transferencia de informacion.

Dependiendo de quien transfiere y quien recibe, el cliente y el servidor toman comportamientos similares e intercambiables. Por esto, la implementacion de los protocolos es compartida en ambos extremos.

3.2. Protocolo de Red

Se implementó un protocolo basado en paquetes donde primero el cliente manda un intento de conexion, el servidor, de poder aceptarlo, asignara un thread a ese cliente (distinguido por ip y puerto) y devolvera una aceptacion de la conexion.

El cliente enviará el primer paquete siempre, donde establecera el tipo de operacion con el protocolo a ejecutar. Se intercambiara informacion y reconocimientos segun lo disponga el protocolo hasta que el emisor envie un paquete con el flag FIN.

3.2.1. Estructura de Paquetes y Flags

Se implementó un protocolo personalizado sobre UDP con la siguiente estructura de paquetes:

- **Header**: Contiene números de secuencia (16 bits), números de ACK, y flags de control
- **Checksum**: Verificación de integridad mediante hash MD5 del payload
- **Payload**: Datos de aplicación o contenido del archivo

Los flags implementados incluyen:

- **SYN**: Sincronización para establecimiento de conexión
- **ACK**: Confirmación de recepción
- **DATA**: Transporte de datos del archivo
- **FIN**: Finalización de conexión
- **ERR**: Indicación de condición de error

3.2.2. Handshake de Conexión

El establecimiento de conexión sigue un handshake simple de 2 pasos:

```
Cliente: SYN (seq=0) → Servidor
Servidor: SYN-ACK → Cliente
```

3.2.3. Protocolos de Transferencia Confiable

Se implementaron dos protocolos de recuperación de errores sobre UDP:
Stop & Wait Protocol

```
ddef send_file(self, file_path: str, address: (str, int)) ->
    bool:
        if not os.path.exists(file_path):
            return False

        try:
            with open(file_path, 'rb') as file:
                while True:
                    chunk = file.read(1024)
                    if not chunk:
                        break
                    packet = create_data_packet(self.
    current_seq, chunk)
                    while True:
                        if self.send_packet(packet, address):
                            ack_received = False
                            start_time = time.time()
                            while not ack_received and (time.
    time() - start_time) < self.timeout:
                                ack_received = receive_ack()
                            if ack_received:
                                self.update_with_ack()
                                break
                            else:
                                if not self.handle_timeout(
    self.current_seq):
                                    return False
                        else:
                            return False
                return True
        except Exception as e:
            return False
```

Listing 1: Implementación Stop & Wait - Envío de archivos

Características de Stop & Wait:

- Envío de un paquete por vez
- Espera de ACK antes del siguiente envío
- Retransmisión tras timeout (5 segundos por defecto)
- Límite máximo de reintentos (3 por defecto)
- Ventana de tamaño 1

Selective Repeat Protocol

```
def send_file(self, file_path: str, address: tuple) -> bool:
    with open(file_path, 'rb') as file:
        eof_reached = False
        while not eof_reached or len(self.send_window) > 0:
            while not eof_reached and len(self.send_window) <
    self.window_size:
                chunk = file.read(1024)
                if not chunk:
                    eof_reached = True
                    break
                packet = create_data_packet(0, chunk)
                if not self.send_packet(packet, address):
                    break
            pkt = self.receive_packet()
            if pkt is None:
                self._check_timeouts()
                time.sleep(0.005)
    return True
```

Listing 2: Implementación Selective Repeat - Ventana deslizante

Características de Selective Repeat:

- Ventana deslizante de tamaño configurable (8 por defecto)
- Múltiples paquetes en tránsito simultáneamente
- ACKs selectivos para paquetes específicos
- Retransmisión solo de paquetes perdidos
- Fast Retransmit en caso de 3 ACKs duplicados
- Buffer de recepción para paquetes fuera de orden

3.2.4. Manejo de Paquetes de Datos

La recepción de datos implementa lógica diferente según el protocolo:
Stop & Wait - Recepción:

```
def on_data(self, packet: RDTPacket):
        if packet.header.sequence_number == self.expected_seq
    :
            ack_num = packet.header.sequence_number
            data_to_write = packet.payload
            self.expected_seq = (self.expected_seq + 1) % 2
            return ack_num, data_to_write, self.expected_seq
        else:
            prev_seq = (self.expected_seq - 1) % 2
            return prev_seq, b"", self.expected_seq
```

Selective Repeat - Recepción:

```
def on_data(self, packet: RDTPacket):
        seq = packet.header.sequence_number
        base = self.expected_seq
        if (seq - base) % 65536 < self.window_size:
            self.receive_window[seq] = packet
        ack_num = seq
        data_to_write = b""
        while self.expected_seq in self.receive_window:
            p = self.receive_window.pop(self.expected_seq)
            data_to_write += p.payload
            self.expected_seq = (self.expected_seq + 1) %
    65536
        return ack_num, data_to_write, self.expected_seq
```

3.2.5. Manejo de Timeouts y Retransmisiones

Ambos protocolos implementan mecanismos de timeout y retransmisión.
Stop & Wait:

- Timeout único para el paquete en tránsito
- Reintentos secuenciales hasta máximo configurado
- Abortar transferencia tras agotar reintentos

Selective Repeat:

- Timer individual por cada paquete en la ventana
- Fast Retransmit basado en ACKs duplicados
- Limpieza de paquetes no confirmados tras timeout

3.2.6. Manejo de Concurrencia en Servidor

El servidor implementa concurrencia mediante ThreadPool y bloqueos:

```
class RDTProtocol:
    def __init__(self, storage_dir="server_files", verbose:
    bool = False):
        self.states_lock = threading.Lock()
        self.client_states = {}
        self.file_lock_manager = FileLockManager()
        self.storage_dir = storage_dir
        self.verbose = verbose
        os.makedirs(storage_dir, exist_ok=True)

def worker_logic(data, address, server_socket, rdt_protocol):
    try:
        rdt_protocol.handle_packet(data, address,
    server_socket)
    except Exception as e:
        print(f"error: {e}")

# En el servidor principal:
pool = multiprocessing.pool.ThreadPool(processes=WORKERS)
while True:
    data, address = server_socket.recvfrom(2048)
    pool.apply_async(worker_logic, (data, address,
    server_socket, rdt_protocol))
```

Listing 3: Manejo concurrente de clientes

3.2.7. Manejo de Archivos Temporales

Para garantizar integridad durante transferencias:

- **UPLOAD**: Archivos temporales con prefijo `.filename.upload.`
- **DOWNLOAD**: Archivos temporales con prefijo `.filename.download.`
- Renombrado atómico al completar transferencia
- Limpieza de archivos temporales en caso de error

3.2.8. Códigos de Error

El protocolo define códigos de error específicos:

- **001**: Error creación de archivo
- **002**: Error en configuración de operación
- **003**: Archivo no encontrado
- **004**: Error acceso a archivo
- **005**: Error escritura de datos

3.2.9. Secuencia de Operaciones

Operación UPLOAD:

1. El cliente envía un paquete `SYN` para establecer la conexión.
2. El servidor responde con `SYN-ACK`.
3. El cliente envía un paquete de operación `UPLOAD` con la metadata del archivo (nombre, tamaño, protocolo).
4. El servidor crea un archivo temporal y envía un `ACK` de confirmación.
5. El cliente comienza a transmitir los datos del archivo usando el protocolo seleccionado (`Stop&Wait` o `SelectiveRepeat`).
6. El servidor confirma cada paquete recibido con `ACKs`, almacenando temporalmente los datos recibidos.
7. Una vez enviado todo el archivo, el cliente transmite un paquete `FIN`.
8. El servidor renombra el archivo temporal y responde con un `FIN-ACK`, finalizando la sesión.

Operación DOWNLOAD:

1. El cliente envía un paquete `SYN` para solicitar conexión.
2. El servidor responde con `SYN-ACK`.
3. El cliente envía un paquete de operación `DOWNLOAD` indicando el nombre del archivo.
4. El servidor verifica su existencia y responde con un `ACK`.
5. El servidor transmite el contenido del archivo usando el protocolo seleccionado (`Stop&Wait` o `SelectiveRepeat`).
6. El cliente escribe los datos recibidos y confirma con `ACKs`.
7. Al completar la transferencia, el servidor envía un paquete `FIN`.
8. El cliente renombra el archivo temporal descargado y responde con un `FIN-ACK`, cerrando la conexión.

4. Pruebas

Con el objetivo de evaluar el desempeño de los protocolos implementados, se diseñaron dos tipos de pruebas utilizando el entorno de emulación `Mininet`. En todos los casos, los enlaces se configuraron con un ancho de banda de 100 Mbps y una latencia de 1 ms por enlace. Los archivos transmitidos fueron pdfs de 6 MB de tamaño.

4.0.1. Prueba 1: Benchmark de desempeño con diferentes tasas de pérdida

La primera prueba consistió en medir el tiempo total de transferencia de un archivo entre un cliente y un servidor bajo diferentes tasas de pérdida de paquetes. Para ello se utilizó un script de benchmarking que inicializa la topología de cliente, servidor y switch, lanza los procesos del servidor, del cliente, y finalmente, registra el tiempo transcurrido desde el inicio hasta la finalización de la transferencia.

- **Topología:** 1 cliente y 1 servidor conectados a un mismo switch.
- **Protocolos evaluados:** Stop-and-Wait y Selective Repeat.
- **Tasas de pérdida simuladas:** 0 %, 5 %, 10 %, 15 %, 20 % y 25 %.
- **Archivo transmitido:** 6 MB.
- **Timeout de socket:** 0.08 s.
- **Tamaño del paquete:** 1400 bytes.

Los resultados obtenidos se resumen en la Tabla 1.

Cuadro 1: Tiempos de transferencia para ambos protocolos según porcentaje de pérdida

Pérdida (%)	**Stop-and-Wait (s)**	**Selective Repeat (s)**
0	31.7	4.4
5	122.1	57.9
10	230.9	116.4
15	345.8	148.4
20	461.5	214.2
25	629.6	380.3

Se puede concluir que el protocolo **Stop-and-Wait** presenta una degradación de desempeño significativa incluso ante pérdidas moderadas, debido a su naturaleza secuencial y a la necesidad de esperar un ACK por cada paquete. El protocolo **Selective Repeat** mantiene un rendimiento de casi la mitad del tiempo al aprovechar una ventana deslizante que para el envío concurrente de múltiples segmentos, reduciendo el impacto de los retransmisiones individuales.

4.0.2. Prueba 2: Concurrencia de Clientes

La segunda prueba tuvo como objetivo verificar el correcto funcionamiento del servidor ante múltiples clientes concurrentes. En este escenario, dos clientes se conectan simultáneamente al mismo servidor y realizan cargas de archivos independientes.

- **Topología:** 1 servidor y 2 clientes conectados al mismo switch.
- **Pérdida simulada:** 10 %.
- **Protocolos empleados:**
  - Cliente 1: Stop-and-Wait.
  - Cliente 2: Selective Repeat.
- **Archivos transmitidos:** 230 KB cada uno.

El siguiente pseudocódigo representa el flujo principal de esta prueba:

```
1. Se inicia el servidor en el host 10.0.0.1.
2. Se crean dos clientes (10.0.0.2 y 10.0.0.3).
3. Ambos suben un su archivo simultáneamente:
   - Cliente 1 usa Stop-and-Wait.
   - Cliente 2 usa Selective Repeat.
4. Se espera a que ambos procesos finalicen o se alcance un timeout.
5. Se verifica la integridad y disponibilidad de los archivos en el servidor.
```

Durante la ejecución se observó que ambos clientes completaron la transferencia correctamente, incluso bajo una pérdida del 10 %. Sin embargo, el cliente que utilizó **Stop-and-Wait** presentó un tiempo de finalización notablemente mayor debido a la retransmisión frecuente de paquetes, mientras que **Selective Repeat** mantuvo un flujo constante de datos.

4.0.3. Conclusión de las pruebas

Los resultados experimentales confirman las expectativas teóricas:

- **Stop-and-Wait** es simple pero ineficiente en entornos con pérdida, debido al bajo aprovechamiento del canal.
- **Selective Repeat** ofrece un desempeño robusto frente a pérdidas crecientes, manteniendo una tasa de transferencia alta gracias al uso de ventanas y ACKs selectivos.

El protocolo **Selective Repeat** resulta más adecuado siempre que se quiera implementar un Protocolo RDT y se pueda prescindir de la simplifidad que aporta **Stop-and-Wait**. Para entornos con condiciones adversas de red o con tráfico concurrente se destaca con creces el protocolo **Selective Repeat**.

5. Preguntas a responder

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

   La arquitectura implementada sigue el modelo cliente-servidor donde el servidor actúa como un repositorio central de archivos y los clientes solicitan operaciones de transferencia. El servidor es un proceso pasivo que espera conexiones, mientras que los clientes inician las comunicaciones. Esta arquitectura permite centralizar el almacenamiento y facilitar el acceso concurrente a múltiples clientes.

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

   El protocolo de capa de aplicación define el lenguaje de comunicación entre las aplicaciones cliente y servidor. Su función principal es especificar:

   - Los tipos de mensajes intercambiados (SYN, DATA, ACK, FIN, ERR).
   - El formato y estructura de los datos transmitidos.
   - Las secuencias válidas de intercambio de mensajes.
   - Los mecanismos de control de errores y recuperación.
   - La semántica de las operaciones disponibles (UPLOAD, DOWNLOAD).

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

   El protocolo desarrollado incluye:

   Estados de Conexión:

   - **SYN**: Establecimiento de conexión.
   - **DATA**: Transferencia en progreso.
   - **FIN**: Finalización ordenada.
   - **ERR**: Manejo de condiciones de error.

   Operaciones Soportadas:

   - UPLOAD: Cliente $\rightarrow$ Servidor.
   - DOWNLOAD: Servidor $\rightarrow$ Cliente.

   Características Clave:

   - Numeración de secuencia para control de flujo.

- Verificación de integridad mediante checksum.
- Mecanismos de ACK y retransmisión.
- Soporte para dos protocolos de recuperación de errores.
- Manejo de archivos binarios de cualquier tamaño.

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?**

   TCP (Transmission Control Protocol):

   - **Servicios**: Transferencia confiable, orientado a conexión, control de flujo, control de congestión.
   - **Características**: Establecimiento de conexión (3-way handshake), entrega en orden, retransmisión automática.
   - **Uso apropiado**: Aplicaciones que requieren integridad de datos (HTTP, FTP, SSH, email).

   UDP (User Datagram Protocol):

   - **Servicios**: Servicio de datagramas no confiable, sin conexión, mínimo overhead.
   - **Características**: Sin garantías de entrega, sin control de flujo, sin ordenamiento.
   - **Uso apropiado**: Aplicaciones en tiempo real (VoIP, video streaming, DNS), donde la velocidad es prioritaria sobre la confiabilidad.

   En este trabajo se eligió UDP para implementar mecanismos de confiabilidad personalizados en la capa de aplicación, permitiendo mayor control sobre las estrategias de recuperación de errores y facilitando el estudio comparativo entre Stop & Wait y Selective Repeat.

6. Dificultades encontradas

Durante el desarrollo del trabajo práctico se presentaron diversas dificultades tanto conceptuales como técnicas. Una de las principales consistió en alcanzar los tiempos de transferencia requeridos bajo distintas condiciones de pérdida de paquetes. Para lograrlo, fue necesario realizar un ajuste fino (*fine-tuning*) de múltiples parámetros, entre ellos el *timeout* de los sockets, el tamaño de los archivos utilizados en las pruebas y la configuración de los métodos concurrentes empleados por el servidor.

La implementación de concurrencia en el servidor presentó un desafío adicional. El manejo de estados para múltiples clientes dentro de un entorno concurrente requirió la definición de distintos niveles de sincronización mediante el uso de *locks*. Estos mecanismos fueron aplicados sobre el diccionario compartido de estados, la información particular de cada cliente y las operaciones de lectura y escritura de archivos, con el fin de garantizar la coherencia del sistema y evitar condiciones de carrera.

Asimismo, la depuración de los protocolos —tanto *Stop & Wait* como *Selective Repeat*— en presencia de pérdida de paquetes resultó especialmente compleja. El proceso de diagnóstico de errores y la validación del comportamiento bajo diferentes configuraciones demandaron un esfuerzo considerable, particularmente teniendo en cuenta que el desarrollo fue realizado por dos integrantes. Esto requirió una comprensión detallada del comportamiento de la capa de transporte, del manejo de los temporizadores y de la interacción entre las capas de red y aplicación.

7. Conclusión

El trabajo permitió comprender en profundidad los principios de la transferencia confiable de datos sobre el protocolo UDP, así como los mecanismos de control de flujo, numeración de secuencias y recuperación de errores propios de los protocolos *Stop & Wait* y *Selective Repeat*.

A través de la implementación y prueba de ambos mecanismos en entornos simulados con Mininet, se pudo observar el impacto directo que la pérdida de paquetes y los valores de *timeout* tienen sobre la eficiencia y el tiempo total de transferencia. En particular, se evidenció que, si bien *Stop & Wait* presenta una estructura más simple, su rendimiento se ve afectado significativamente ante pérdidas moderadas; mientras que *Selective Repeat*, al utilizar una ventana deslizante y manejo selectivo de confirmaciones, logra mantener una mayor estabilidad.

Además, durante el desarrollo fue necesario profundizar en el funcionamiento interno del protocolo TCP, con el objetivo de adaptar varios de sus principios y estrategias al entorno no confiable de UDP, logrando así implementar mecanismos de retransmisión y control de flujo confiables de manera manual.

En conjunto, el trabajo favoreció la consolidación de conocimientos en programación concurrente, uso de la interfaz de *sockets*, simulación de redes y diseño de protocolos confiables sobre canales no confiables, reforzando la comprensión práctica de los conceptos fundamentales de la capa de transporte y su relación con la arquitectura cliente-servidor.