Introducción a los Sistemas Distribuídos (75.43)

TP N°1: File Transfer

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

10 de septiembre 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

```
def send_file(self, file_path: str, address: (str, int)) -> bool:
    with open(file_path, 'rb') as file:
        while True:
            chunk = file.read(1024)
            if not chunk: break

            packet = create_data_packet(0, 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:
                        received_packet = self.receive_packet()
                        if received_packet and received_packet.has_flag(RDTFlags.ACK):
                            if received_packet.header.ack_number == self.current_seq -1:
                                ack_received = True
                    if ack_received: break
                    else:
                        if not self.handle_timeout(self.current_seq - 1):
                            return False
                else: return False
    return True
```

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) % 65536
        return ack_num, data_to_write, self.expected_seq
    prev_seq = self.expected_seq - 1 if self.expected_seq > 0
    else 0
    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:

```python
def _check_timeouts(self):
    current_time = time.time()
    to_remove: list[int] = []

    for seq_num, entry in list(self.send_window.items()):
        if current_time - entry['timestamp'] > self.timeout:
            if not self.handle_timeout(seq_num):
                to_remove.append(seq_num)

    for seq_num in to_remove:
        if seq_num in self.send_window:
            del self.send_window[seq_num]
        if seq_num in self.ack_received:
            del self.ack_received[seq_num]
```

- 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

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. Cliente envía SYN para establecer conexión
2. Servidor responde con SYN-ACK
3. Cliente envía paquete de operación UPLOAD con metadata
4. Servidor crea archivo temporal y envía ACK
5. Cliente transmite datos usando protocolo seleccionado
6. Servidor confirma recepción con ACKs
7. Cliente envía FIN al completar
8. Servidor renombra archivo temporal y envía FIN-ACK

Operación DOWNLOAD:

1. Cliente envía SYN para establecer conexión
2. Servidor responde con SYN-ACK
3. Cliente envía paquete de operación DOWNLOAD
4. Servidor verifica existencia y envía ACK
5. Servidor transmite archivo usando protocolo especificado
6. Cliente escribe datos y confirma con ACKs
7. Servidor envía FIN al completar transferencia
8. Cliente renombra archivo temporal y envía FIN-ACK

4. Pruebas

4.1. Configuración del Entorno

(Pendiente: describir la configuración de mininet y las condiciones de prueba)

4.2. Resultados de Performance

(Pendiente: incluir tablas y gráficos comparativos de Stop & Wait vs. 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

7. Conclusión