Universidad de Buenos Aires

Facultad de Ingeniería

Redes

Curso Hamelin

Trabajo Práctico 1

Implementación de Protocolos de Transferencia Confiable

`STOP_AND_WAIT` y `GO_BACK_N`

Integrantes:

Estudiante	**Padrón**
Colman Salinas, Agustín Reinaldo	108804
Fernández Márquez, Ramiro Andrés	105595
Lofano, Tomás Arián	101721
Rulli, Federico	92347
Telmo, Lautaro	111644

Profesores: Esteban Carisimo y Juan Ignacio Lopez Pecora

Fecha de Entrega: 07 de Octubre de 2025

Índice

1. Introducción

El objetivo de este trabajo práctico fue desarrollar una aplicación de red con **arquitectura cliente-servidor** para la **transferencia de archivos** (UPLOAD y DOWNLOAD). La implementación requirió la comprensión de la **interfaz de sockets** y el modelo de servicio de la capa de transporte.

El principal desafío fue la implementación de un protocolo de **Transferencia de Datos Confiable (RDT)** sobre el protocolo de capa de transporte **UDP**, que no es confiable. Para ello, se implementaron dos mecanismos de recuperación de errores: **Stop & Wait** y **Go-Back-N**. Las pruebas de validación se realizaron en una topología simulada con `mininet` forzando la pérdida de paquetes.

2. Hipótesis y Suposiciones Realizadas

Para llevar a cabo la implementación, se realizaron las siguientes hipótesis y suposiciones:

1. **Fiabilidad sobre Pérdida:** Se asume que el protocolo RDT implementado (Stop & Wait y Go-Back-N) garantizará la entrega de paquetes incluso bajo la condición crítica de hasta un **10 % de pérdida de paquetes** en los enlaces.
2. **Performance:** Se asume que la versión **Go-Back-N** ofrecerá una **mejor performance** (menor tiempo de transferencia) que la versión Stop & Wait, especialmente con archivos grandes y bajo condiciones de pérdida de paquetes.
3. **Ambiente de Prueba:** Se asume que el entorno `mininet` simula de manera efectiva las condiciones de red necesarias para forzar la pérdida de paquetes y validar los protocolos.

3. Implementación

3.1. Arquitectura General

La aplicación se estructura en tres componentes principales que implementan un protocolo de transferencia confiable sobre UDP:

3.1.1. Capa de Transporte Confiable (`SocketTP`)

El núcleo de la implementación reside en la clase `SocketTP` (`socket_tp.py`), que proporciona una abstracción de comunicación confiable sobre UDP, simulando características de TCP como:

- **Control de flujo mediante ventana deslizante**: Implementado en la clase `Window`, permite regular la cantidad de datos en tránsito.
- **Numeración de secuencia**: Manejada por la clase `Sequence`, garantiza el orden correcto de los paquetes.
- **Temporizadores adaptativos**: La clase `Timer` implementa el cálculo dinámico del RTO (Retransmission Timeout) usando las fórmulas de Jacobson/Karels para estimar el RTT y su desviación.
- **Recuperación de errores configurable**: Soporta dos modos mediante `ErrorRecoveryMode`:
  - **Go-Back-N**: Ventana de 7000 bytes (5 paquetes de 1400 bytes)
  - **Stop-and-Wait**: Ventana de 1400 bytes (1 paquete)

3.1.2. Protocolo de Comunicación

Estructura de Paquetes Los paquetes utilizan un formato fijo de 7 bytes de encabezado más datos:

```
| Seq Number (4B) | ACK (1B) | SYN (1B) | FIN (1B) | Data (variable) |
```

- **Seq Number**: Número de secuencia acumulativo (total de bytes enviados)
- **ACK**: Indica paquete de reconocimiento
- **SYN**: Usado en el establecimiento de conexión
- **FIN**: Reservado para cierre de conexión (no implementado completamente)
- **Data**: Hasta 1400 bytes de payload

Establecimiento de Conexión (Three-Way Handshake) El protocolo implementa un handshake de tres vías similar a TCP:

1. **Cliente → Servidor**: `SYN` con payload indicando el modo de recuperación de errores

   ```
   Packet(syn=True, data=build_syn_payload(mode))
   ```

2. **Servidor → Cliente**: `SYN-ACK` indicando aceptación

   ```
   Packet(syn=True, ack=True)
   ```

3. **Cliente → Servidor**: `ACK` final confirmando la conexión

`Packet(ack=True)`

El servidor crea un nuevo socket UDP en un puerto efímero para cada conexión aceptada, permitiendo múltiples conexiones concurrentes.

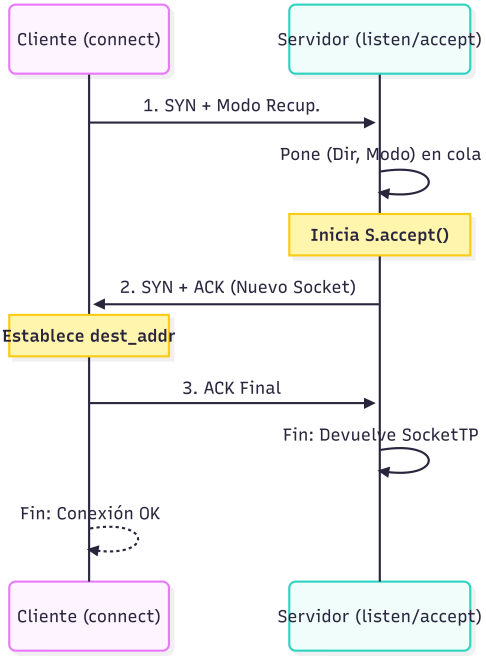


Figura 1: Diagrama de secuencia - Three-Way Handshake

Transferencia de Datos

Envío (sendall) El método `sendall()` implementa el protocolo de ventana deslizante:

```
while not fin:
    # Calcular ventana disponible
    end = start + min(PACKET_DATA_SIZE, window.size, len(data[start:]))

    # Enviar paquete con número de secuencia
    packet = Packet(data=data[start:end], seq_number=sequence)
    socket.sendto(packet.to_bytes(), dest_addr)

    # Iniciar timer si no está activo
    if not timer.is_set():
        timer.set()

    # Reducir ventana disponible
    window.decrease(len(data[start:end]))
```

Características clave:

- Los números de secuencia son acumulativos (indican bytes totales enviados)
- La ventana se reduce con cada envío y aumenta con cada ACK
- El timer se actualiza dinámicamente usando estimaciones de RTT

- En caso de timeout, se reinicia la transmisión desde el último ACK recibido (Go-Back-N)

Recepción (recv) El receptor implementa ACKs acumulativos:

```
if packet.seq_number == received_ack:
    # Paquete esperado, aceptar
    received_ack += len(packet.data)
    packet_queue.put(packet)
else:
    # Paquete fuera de orden, descartar
    logger.debug(f'IGNORED expected: {received_ack}')

# Siempre enviar ACK con el próximo byte esperado
socket.sendto(Packet(ack=True, seq_number=received_ack).to_bytes(), addr)
```

Comportamiento:

- Solo acepta paquetes en orden (no bufferiza paquetes fuera de secuencia)
- Envía ACK por cada paquete recibido, incluso duplicados
- Los ACKs indican el próximo byte esperado

Manejo de Timeouts y Retransmisiones El sistema de timeouts utiliza un algoritmo adaptativo:

```
# Cálculo del RTT estimado (EWMA)
estimated_RTT = (1 - \alpha) * estimated_RTT + \alpha * sample_RTT

# Desviación del RTT
dev_RTT = (1 - \beta) * dev_RTT + \beta * |sample_RTT - estimated_RTT|

# Timeout calculado
RTO = max(0.02, estimated_RTT + 4 * dev_RTT)
```

Con $\alpha = 0{,}125$ y $\beta = 0{,}25$ (valores estándar de TCP).

Cuando expira el timer:

- Se reinicia la ventana al tamaño configurado
- Se resetea el número de secuencia de envío al último ACK
- Se retransmiten todos los paquetes desde ese punto (Go-Back-N)

3.1.3. Protocolo de Aplicación (Transferencia de Archivos)

La capa de aplicación implementa dos operaciones mediante un protocolo simple:

Upload (Cliente → Servidor)

1. Cliente envía: modo operación (4 bytes) = `ClientMode.UPLOAD`
2. Cliente envía: longitud del nombre (4 bytes)
3. Cliente envía: nombre del archivo (variable)

4. Cliente envía: tamaño del archivo (4 bytes)
5. Cliente envía: datos del archivo (variable)

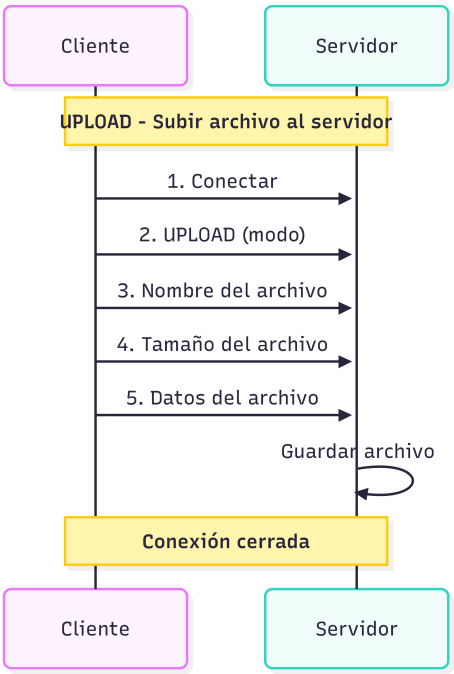


Figura 2: Diagrama de secuencia - UPLOAD (Subir archivo al servidor)

Download (Cliente ← Servidor)

1. Cliente envía: modo operación (4 bytes) = `ClientMode.DOWNLOAD`
2. Cliente envía: longitud del nombre (4 bytes)
3. Cliente envía: nombre del archivo solicitado (variable)
4. Servidor responde:
   - Si existe: tamaño (4 bytes) + datos (variable)
   - Si no existe: `FILE_NOT_FOUND_ERROR_CODE` (-1 en 4 bytes con signo)

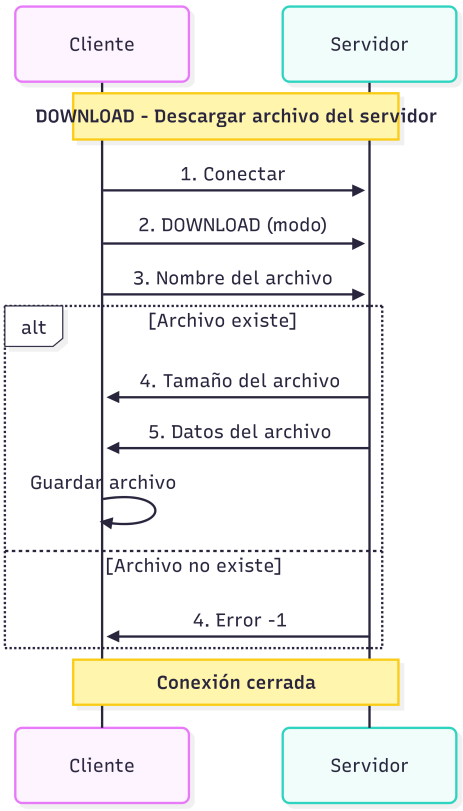


Figura 3: Diagrama de secuencia - DOWNLOAD (Descargar archivo desde servidor)

3.1.4. Concurrencia y Threading

El servidor maneja múltiples aspectos concurrentemente:

- **Thread principal**: Acepta nuevas conexiones
- **Thread de lectura de teclado**: Espera la tecla 'q' para cierre ordenado
- **Por cada conexión**:
  - Thread del cliente: Maneja la lógica de upload/download
  - Thread de procesamiento de paquetes entrantes: Lee del socket UDP
  - Thread del timer: Monitorea expiración de timeouts

La sincronización se maneja mediante:

- `Lock` para acceso a variables compartidas (secuencia, ventana)
- `Condition` para notificación cuando la ventana tiene espacio disponible
- `Queue` thread-safe para paquetes entrantes y conexiones pendientes

3.1.5. Validaciones y Manejo de Errores

El módulo `validations.py` implementa verificaciones previas a la ejecución:

- **Host y puerto**: Validez de dirección IP/hostname y rango de puerto
- **Archivos**: Existencia, permisos de lectura/escritura
- **Nombres de archivo**: Caracteres válidos según el sistema operativo
- **Directorios**: Existencia y permisos de escritura

Durante la ejecución, se manejan:

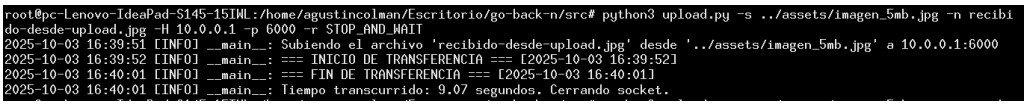
- Timeouts de conexión (30 segundos)
- Timeouts de operación (30 segundos sin recibir ACKs)
- Pérdida de paquetes (mediante retransmisión)
- Paquetes duplicados o fuera de orden (descarte)

4. Pruebas

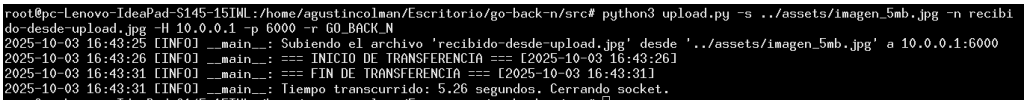
Inicialmente, se realizaron pruebas de **UPLOAD Y DOWNLOAD** utilizando archivos de prueba de 5 MB, en condiciones ideales de red (sin pérdida de paquetes) para establecer una línea base de rendimiento. Se probaron ambos protocolos de recuperación de errores: `STOP_AND_WAIT` y `GO_BACK_N`.

4.1. Subida de archivo sin pérdida de paquetes

Utilizando `STOP_AND_WAIT`:



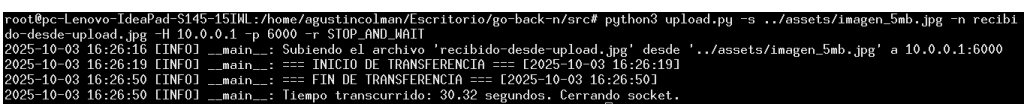
Utilizando `GO_BACK_N`:



`GO_BACK_N` demostró ser más eficiente en condiciones ideales, completando la transferencia en aproximadamente la mitad del tiempo que `STOP_AND_WAIT` (5.26 vs 9.07 segundos). Esto demuestra que `GO_BACK_N` aprovecha mejor la banda disponible al permitir múltiples paquetes en tránsito simultáneamente, mientras que `STOP_AND_WAIT` mantiene un enfoque secuencial que resulta menos eficiente.

4.2. Subida de archivo con pérdida de paquetes de 10 %

Utilizando `STOP_AND_WAIT`:



Utilizando `GO_BACK_N`:

```
root@pc-Lenovo-IdeaPad-S145-15IWL:/home/agustincolman/Escritorio/go-back-n/src# python3 upload.py -s ../assets/imagen_5mb.jpg -n recibi
do-desde-upload.jpg -H 10.0.0.1 -p 6000 -r GO_BACK_N
2025-10-03 16:25:45 [INFO] __main__: Subiendo el archivo 'recibido-desde-upload.jpg' desde '../assets/imagen_5mb.jpg' a 10.0.0.1:6000
2025-10-03 16:25:47 [INFO] __main__: === INICIO DE TRANSFERENCIA === [2025-10-03 16:25:47]
2025-10-03 16:26:07 [INFO] __main__: === FIN DE TRANSFERENCIA === [2025-10-03 16:26:07]
2025-10-03 16:26:07 [INFO] __main__: Tiempo transcurrido: 19.96 segundos. Cerrando socket.
```

Ambos protocolos de recuperación de errores demostraron ser efectivos, permitiendo que el archivo llegara completo al servidor incluso en condiciones adversas de red. Se observa una diferencia significativa en los tiempos de transferencia entre ambos protocolos: `GO_BACK_N` resultó más eficiente, completando la transferencia en aproximadamente 20 segundos, mientras que `STOP_AND_WAIT` requirió cerca de 30 segundos. Esta diferencia se debe a que `GO_BACK_N` permite tener múltiples paquetes en tránsito simultáneamente, mientras que `STOP_AND_WAIT` espera la confirmación de cada paquete antes de enviar el siguiente.

4.3. Descarga de archivo sin pérdida de paquetes

Utilizando `STOP_AND_WAIT`:

```
root@pc-Lenovo-IdeaPad-S145-15IWL:/home/agustincolman/Escritorio/FIUBA/2c-2025/redes/go-back-n/src# python3 download.py -d ../assets/do
wnload/perrito.png -n perro.png -H 10.0.0.1 -p 6000 -r STOP_AND_WAIT
2025-10-05 22:10:52 [INFO] __main__: Descargando el archivo 'perro.png' de 10.0.0.1:6000 a '../assets/download/perrito.png'
2025-10-05 22:10:53 [INFO] __main__: === INICIO DE DESCARGA === [2025-10-05 22:10:53]
2025-10-05 22:11:04 [INFO] __main__: === FIN DE DESCARGA === [2025-10-05 22:11:04]
2025-10-05 22:11:04 [INFO] __main__: Tiempo transcurrido: 10.80 segundos. Cerrando socket.
```

Utilizando `GO_BACK_N`:

```
root@pc-Lenovo-IdeaPad-S145-15IWL:/home/agustincolman/Escritorio/FIUBA/2c-2025/redes/go-back-n/src# python3 download.py -d ../assets/do
wnload/perrito.png -n perro.png -H 10.0.0.1 -p 6000 -r GO_BACK_N
2025-10-05 22:13:31 [INFO] __main__: Descargando el archivo 'perro.png' de 10.0.0.1:6000 a '../assets/download/perrito.png'
2025-10-05 22:13:32 [INFO] __main__: === INICIO DE DESCARGA === [2025-10-05 22:13:32]
2025-10-05 22:13:41 [INFO] __main__: === FIN DE DESCARGA === [2025-10-05 22:13:41]
2025-10-05 22:13:41 [INFO] __main__: Tiempo transcurrido: 9.11 segundos. Cerrando socket.
```

`GO_BACK_N` completó la transferencia en aproximadamente 9.11 segundos, mientras que `STOP_AND_WAIT` requirió 10.00 segundos, mostrando una mejora de alrededor del 9 %. Aunque `GO_BACK_N` es teóricamente más eficiente al permitir múltiples paquetes en tránsito, la diferencia observada es moderada debido a las condiciones favorables de la red local, donde la baja latencia reduce el impacto del enfoque secuencial de `STOP_AND_WAIT`. En escenarios con archivos de mayor tamaño, la ventaja de sería más pronunciada.

4.4. Descarga de archivo con pérdida de paquetes de 10 %

Utilizando `STOP_AND_WAIT`:

```
root@pc-Lenovo-IdeaPad-S145-15IWL:/home/agustincolman/Escritorio/FIUBA/2c-2025/redes/go-back-n/src# python3 download.py -d ../assets/re
cibido_desde_servidor.jpg -n imagen_5mb.jpg -H 10.0.0.1 -p 6000 -r STOP_AND_WAIT
2025-10-04 18:19:02 [INFO] __main__: Descargando el archivo 'imagen_5mb.jpg' de 10.0.0.1:6000 a '../assets/recibido_desde_servidor.jpg'
2025-10-04 18:19:03 [INFO] __main__: === INICIO DE DESCARGA === [2025-10-04 18:19:03]
2025-10-04 18:19:37 [INFO] __main__: === FIN DE DESCARGA === [2025-10-04 18:19:37]
2025-10-04 18:19:37 [INFO] __main__: Tiempo transcurrido: 34.34 segundos. Cerrando socket.
```

Utilizando `GO_BACK_N`:

```
root@pc-Lenovo-IdeaPad-S145-15IWL:/home/agustincolman/Escritorio/FIUBA/2c-2025/redes/go-back-n/src# python3 download.py -d ../assets/re
cibido_desde_servidor.jpg -n imagen_5mb.jpg -H 10.0.0.1 -p 6000 -r GO_BACK_N
2025-10-04 18:19:42 [INFO] __main__: Descargando el archivo 'imagen_5mb.jpg' de 10.0.0.1:6000 a '../assets/recibido_desde_servidor.jpg'
2025-10-04 18:19:44 [INFO] __main__: === INICIO DE DESCARGA === [2025-10-04 18:19:44]
2025-10-04 18:20:10 [INFO] __main__: === FIN DE DESCARGA === [2025-10-04 18:20:10]
2025-10-04 18:20:10 [INFO] __main__: Tiempo transcurrido: 26.59 segundos. Cerrando socket.
```

Con un 10 % de pérdida de paquetes, `GO_BACK_N` completó la transferencia en aproximadamente 26.59 segundos, mientras que `STOP_AND_WAIT` requirió 34.04 segundos, representando una mejora del 22 %. En condiciones de red adversas con pérdidas, `GO_BACK_N` demuestra una ventaja más notable que en el escenario sin pérdidas, ya que su capacidad de mantener múltiples paquetes en tránsito le permite recuperarse más eficientemente de las retransmisiones necesarias. Aunque ambos protocolos experimentan un aumento significativo en el tiempo de transferencia comparado con el escenario ideal, `GO_BACK_N` maneja las pérdidas de manera más efectiva.

5. Preguntas a responder

5.1. Describa la arquitectura Cliente-Servidor

La arquitectura cliente-servidor es un modelo de comunicación en el que existe un host que permanece siempre encendido, denominado servidor, cuya función es atender las solicitudes provenientes de múltiples hosts clientes. Un ejemplo clásico es la Web, donde el navegador, que se encuentra corriendo en el cliente, solicita una petición al servidor, y este responde enviando dicha respuesta. En este esquema, los clientes no se comunican directamente entre sí; por ejemplo, dos navegadores no establecen conexión entre ellos, sino que toda la interacción pasa por el servidor. Una característica fundamental de esta arquitectura es que el servidor posee una dirección fija y conocida, lo que permite que cualquier cliente pueda localizarlo y enviarle paquetes para establecer comunicación. Entre las aplicaciones más conocidas que emplean este modelo se encuentran la Web, FTP, Telnet y el correo electrónico. Sin embargo, en muchos casos un único servidor no es suficiente para procesar la gran cantidad de solicitudes de los clientes. Servicios de gran popularidad, como redes sociales o plataformas de comercio electrónico, pueden saturar rápidamente a un solo servidor. Por esta razón, es habitual que se utilicen centros de datos (data centers), que agrupan un gran número de hosts y permiten conformar un servidor virtual de gran capacidad.

Entre las aplicaciones más conocidas que emplean este modelo se encuentran la Web, FTP, Telnet y el correo electrónico. Sin embargo, en muchos casos un único servidor no es suficiente para procesar la gran cantidad de solicitudes de los clientes. Servicios de gran popularidad, como redes sociales o plataformas de comercio electrónico, pueden saturar rápidamente a un solo servidor. Por esta razón, es habitual que se utilicen centros de datos (data centers), que agrupan un gran número de hosts y permiten conformar un servidor virtual de gran capacidad.

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

La función de un protocolo de capa de aplicación es definir cómo los procesos de una aplicación, que se ejecutan en sistemas finales distintos, intercambian mensajes entre sí. En particular, un protocolo de aplicación establece los tipos de mensajes que se pueden enviar (por ejemplo, solicitudes y respuestas), la sintaxis de esos mensajes, es decir, qué campos incluyen y cómo se estructuran, la semántica de los campos, es decir, el significado de la información contenida en cada uno y las reglas de comunicación, que determinan cuándo y cómo un proceso debe enviar mensajes y cómo debe responder a los que recibe.

Los protocolos de capa de aplicación pueden ser públicos, como HTTP (definido en un RFC y utilizado en la Web), o de propietario, como el de Skype. Es importante aclarar que un protocolo de aplicación constituye sólo una parte de una aplicación de red más amplia: por ejemplo, en la Web, HTTP es el protocolo que define el intercambio de mensajes entre navegadores y servidores, pero la aplicación también incluye otros elementos como los formatos de documento (HTML), los navegadores o los servidores web.

Formalmente, la función principal de un protocolo de capa de aplicación es regular el formato, el significado y el intercambio de mensajes entre procesos de una aplicación distribuidos en la red, permitiendo así que las aplicaciones funcionen de manera interconectada entre diferentes sistemas.

5.3. Detalle el protocolo de aplicación desarrollado en este trabajo

La explicación de la implementación a bajo nivel del protocolo desarrollado en este trabajo puede encontrarse con lujo de detalles en la sección 3.3 del presente documento.

El protocolo de aplicación desarrollado en este trabajo corresponde a un protocolo propio diseñado e implementado sobre sockets, cuya finalidad es modelar el funcionamiento del algoritmo de control de flujo Go-Back-N y Stop and Wait. El objetivo del protocolo desarrollado es brindar un servicio de comunicación fiable sobre una red potencialmente no confiable.

Desde el punto de vista de su estructura, los mensajes que se intercambian entre cliente y

servidor se clasifican principalmente en dos tipos: paquetes de datos y paquetes de confirmación (denominados ACKs en nuestro código). Cada paquete de datos contiene un número de secuencia que permite identificarlo unívocamente y controlar el estado de la transmisión. A su vez, los paquetes de confirmación incluyen el número de secuencia del último paquete recibido en orden, lo que nos permite llevar control de los paquetes perdidos, o desordenados, desde el lado de quien está emitiendo información. El protocolo tiene en cuenta en su funcionamiento situaciones anómalas cómo pérdidas, duplicados o desorden en la llegada de paquetes.

Las reglas de comunicación establecidas en nuestro protocolo determinan que el emisor pueda enviar de manera continua un conjunto de paquetes, limitado por el tamaño de la ventana de transmisión y del modo de operación elegido (Go back N o Stop and Wait). Una vez enviados, debe esperar la llegada de los ACKs correspondientes. Si dentro de un tiempo de espera predeterminado no se recibe la confirmación, o si se detecta la pérdida de un paquete intermedio, el emisor retransmite todos los paquetes de la ventana desde el punto en el que se produjo la pérdida. En este punto, podemos pensar Stop and Wait, cómo un Go Back N de ventana 1. El receptor, por su parte, acepta únicamente los paquetes en orden, descartando aquellos que llegan fuera de secuencia y confirmando únicamente el último paquete recibido correctamente.

En conclusión, el protocolo de aplicación implementado sobre una red no confiable nos permite asegurar, por medio de nuestra implementación, garantías sobre la transmisión de los paquetes y la información por medio del mismo.

5.4. TCP y UDP: Servicios, Características y Casos de Uso

Protocolo	**Servicio que Provee**	**Características**	**Uso Apropiado**
TCP	Orientado a Conexión, Garantía de Entrega, Control de Flujo/Congestión, Entrega en Orden.	Overhead Alto, Lento en establecer conexión, Confiable.	Transferencia de Archivos (FTP, HTTP), Correo Electrónico (SMTP), Terminal Remota (SSH).
UDP	No Confiable (Best-effort), Sin Conexión, Sin Control de Flujo/-Congestión, Sin Entrega en Orden.	Overhead Mínimo, Rápido, No garantiza orden o entrega.	Streaming de Video/Voz (VoIP), DNS, SNMP, y aplicaciones que requieren su propia RDT (como este TP).

Cuadro 1: Servicios, características y casos de uso de TCP y UDP.

6. Dificultades encontradas

Durante el desarrollo del trabajo práctico se presentaron diversas dificultades técnicas y conceptuales que debieron ser abordadas de manera progresiva para lograr una implementación funcional del protocolo Go-Back-N sobre sockets.

En primer lugar, resultó desafiante el diseño de la lógica del protocolo sobre una red no confiable. Fue necesario implementar correctamente el temporizador del emisor y establecer los mecanismos de retransmisión de la ventana completa ante la pérdida de un paquete o la ausencia de confirmaciones (ACKs) dentro del tiempo de espera. La coordinación entre los eventos de envío, recepción y retransmisión demandó un control cuidadoso de concurrencia y sincronización.

Otra dificultad significativa fue la gestión de la numeración de secuencia y los ACKs acumulativos. Al tratarse de un protocolo que descarta paquetes fuera de orden, se presentaron errores iniciales en la interpretación de los ACKs y en el avance de la ventana deslizante del emisor. Resolver estas inconsistencias implicó depurar el código, trazar paso a paso los intercambios y verificar

que el comportamiento se ajustara al algoritmo Go-Back-N.

Finalmente, se presentaron problemas asociados a la observabilidad y verificación de la comunicación. La necesidad de capturar y estudiar los paquetes transmitidos con herramientas como Wireshark, y en muchos casos el output de la línea de comandos, evidenció la importancia de comprender en detalle la estructura de los mensajes, su trazabilidad y el análisis de la concurrencia en los distintos threads que maneja cada cliente. Esto permitió detectar fallas como duplicaciones o pérdidas simuladas, y ajustar la implementación hasta obtener un comportamiento confiable.

7. Conclusiones

Este trabajo práctico permitió aplicar los fundamentos teóricos de la capa de transporte para construir un servicio de capa de aplicación confiable sobre una base no confiable (UDP). Se logró validar que ambos protocolos RDT (Stop & Wait y Go-Back-N) cumplen con la garantía de entrega incluso bajo una pérdida del 10 %.

La simplicidad asociada al protocolo Stop-and-Wait conlleva un uso ineficiente del canal, dado que el emisor permanece inactivo durante buena parte del tiempo, sobre todo en enlaces con retardos de propagación significativos. La consecuencia directa es un bajo aprovechamiento del ancho de banda y una marcada limitación en la tasa de transferencia efectiva, de este protocolo podemos rescatar que en ámbitos educativos, su sencillez de operación posibilita una rápida comprensión de su lógica y el entorno en el que se desarrolla.

Con respecto al protocolo Go-Back-N notamos un incremento sustancial en el rendimiento, ya que permitió mantener ocupado el canal con múltiples tramas en vuelo de manera simultánea, reduciendo así los tiempos de inactividad del emisor. Si bien la retransmisión de varias tramas puede implicar cierto desperdicio de recursos en situaciones de error frecuente, en condiciones normales el protocolo resultó mucho más eficiente que Stop-and-Wait.

En términos evolutivos, puede afirmarse que Stop-and-Wait representa una solución válida únicamente en entornos de baja latencia y baja probabilidad de error, mientras que Go-Back-N evidencia cómo la introducción de mecanismos más sofisticados de control de flujo y retransmisión permiten aprovechar mejor los recursos de red y mejorar la confiabilidad.