Universidad de Buenos Aires

Facultad de Ingeniería

75.33 - Introducción a los Sistemas Distribuidos

Trabajo Practico

Nombre	Padrón	Mail
Gamberale, Luciano Martin	105892	lgamberale@fi.uba.ar
Martínez Quintero, Erick	103745	egmartinez@fi.uba.ar
Monpelat, Facundo	92716	fmonpelat@fi.uba.ar
Vasquez Jimenez, Miguel Angel	107378	mvasquezj@fi.uba.ar
Guglielmone, Lionel	96963	lguglielmone@fi.uba.ar

Índice

1. Introducción

En el siguiente informe se describe el diseño e implementación de:

- Una arquitectura de aplicación cliente-servidor para la transferencia de archivos
- Una implementación de protocolo RDT (Reliable Data Transfer) empleando UDP de base, extendiéndolo con técnicas de control de flujo y control de errores.

La solución se desarrolló con la finalidad de brindar una transferencia de datos confiable y eficiente entre dispositivos en una red de computadoras, simulada con **Mininet**.

La arquitectura implementada se basa en los principios de la arquitectura cliente-servidor, en la que el cliente envía solicitudes al servidor y el servidor responde con los recursos solicitados. Para lograr esto, se utilizaron *sockets* en **Python** para implementar la comunicación entre los distintos hosts.

Durante la implementación, se encontraron dificultades en el manejo de paquetes y segmentos, así como en la coordinación del *time out* para garantizar la correcta transferencia de los archivos. Para superar estos desafíos, se aplicaron técnicas de control de flujo y control de errores. Específicamente, se utilizaron las técnica de **Stop-and-wait** y **Selective Repeat** para controlar el flujo de datos y emplear manejo de los errores de transmisión.

Con el fin de evaluar el rendimiento de la arquitectura implementada, se utilizó Wireshark para observar diferentes métricas de red y analizar el tráfico de paquetes.

Este trabajo toma como base los conceptos presentados en el libro *Computer Networking: A Top-Down Approach*, de Kurose y Ross, séptima edición, en el que se describen los diferentes aspectos de las redes de computadoras y los protocolos utilizados en ellas. Asimismo, se utilizaron las notas de las clases. La implementación de esta solución no solo permitió poner en práctica los conceptos aprendidos, sino que también permitió aplicarlos a un problema real en el mundo de las redes de computadoras.

2. Hipótesis y suposiciones realizadas

Suposiciones

- Los clientes tienen acceso a una red con suficiente ancho de banda para la transferencia de archivos (el desarrollo está en un ámbito controlado)
- Los servidores tienen suficiente capacidad de almacenamiento para los archivos que se subirán (el desarrollo está en un ámbito controlado)
- Los clientes y servidores tienen un sistema operativo que soporta los *sockets* y el protocolo de transporte UDP
- Los clientes y servidores pueden resolver nombres de host utilizando servicios de DNS.
- Los tamaños de archivo no son tan grandes como para agotar el espacio en disco disponible en el servidor (se define un límite máximo para las transferencias)

Hipótesis

- Los clientes tienen los permisos necesarios para acceder a los archivos que quieren subir o descargar
- El cliente y el servidor utilizan los mismos algoritmos de control de flujo y errores (Stop-and-wait o Selective Repeat) y temporizadores (*timeout*)
- No hay restricciones de seguridad en la red que puedan afectar la transferencia de archivos, como *firewalls* o filtrado de paquetes

3. Implementación

Para el protocolo de transporte se utilizo el *header* llamado HeaderRDT, conformado de los siguientes campos:

1. Protocol: 1 byte - Selecciona el protocolo
   - SAW = 0
   - SR = 1
2. Data Size: 4 bytes - Denota el tamaño de los datos
3. Seq Number: 4 bytes - Numero de secuencia del paquete
4. Ack Number: 4 bytes - Acuso de recibo del paquete
5. Syn: 1 byte - Bit de sincronización
6. Fin: 1 byte - Bit de finalización
7. Header Checksum: 1 byte - Numero validador CRC de la integridad del *header*

Siguiendo con el protocolo de aplicación se utilizo el *header* llamado ApplicationHeaderRDT, donde los campos son:

1. Transfer Type: 1 byte - Selecciona la transferencia
   - Upload = 0
   - Download = 1
2. File Name: 40 bytes - Nombre de archivo
3. File Size: 4 bytes - Tamaño de archivo
4. Header Checksum: 1 byte - Numero validador CRC de la integridad del header

Se puede ver en la siguiente imagen como se conforma un segmento completo con ambos *headers* y datos de aplicación:

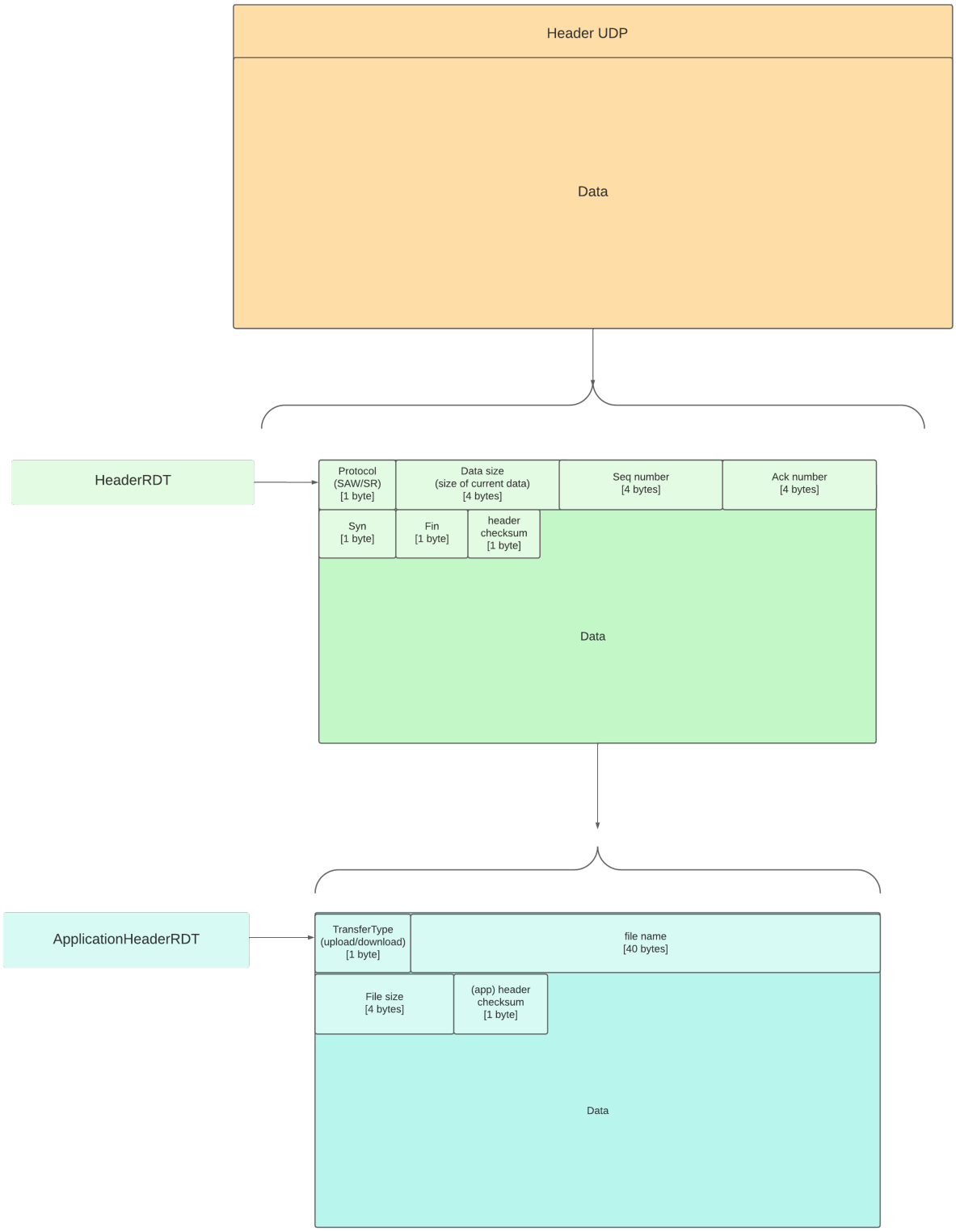

Figura 1: Arquitectura de los segmentos Headers mas Datos

En general, para la realización del *three-way-handshake* se implementó un protocolo de envío y recepción de forma tal que se realizan múltiples reintentos de conexión, esto dado a que definimos que ambos hosts reciban al menos un mensaje de tipo *handshake* como condición necesaria para que el cliente establezca una conexión. Si bien se envía un tercer mensaje de confirmación desde el cliente (como se muestra en la figura de la siguiente página), el servidor puede optar por recibir data de usuario directamente tras haber recibido el primer mensaje. Esto hace que se contemple el eventual caso en el que se pierda el ultimo paquete de confirmación del *handshake*, y funcione de manera eficiente.

Luego, para implementar los protocolos de aplicación se definieron las siguientes secuencias de mensajes a interpretar por cada uno de los hosts:

$->$ **Caso: Cliente solicita descarga**
Para este caso se definió el envió obligatorio del objeto "$ApplicationHeaderRDT$", particularmente requiriendo ida y vuelta (Cliente $->$ Server y Server $->$ Cliente), de tal forma que el cliente pueda obtener información sobre el estado del archivo que quiere descargar (dado que el mismo solo conoce el nombre del archivo a priori). En la respuesta recibida por el servidor puede obtener:

- Confirmación de que va a enviarle el archivo si se reemite el ApplicationHeader con el mismo filename y si se le brinda un filesize no nulo.
- Rechazo del pedido (porque el archivo no existe en el server), en donde se le envía el ApplicationHeader con NO SUCH FILE de filename y valor nulo de filesize.

En la siguiente página se observa un gráfico del inicio de la comunicación recién descrita: (Para simplificar la visualización se muestra el caso mencionado utilizando Stop And Wait)

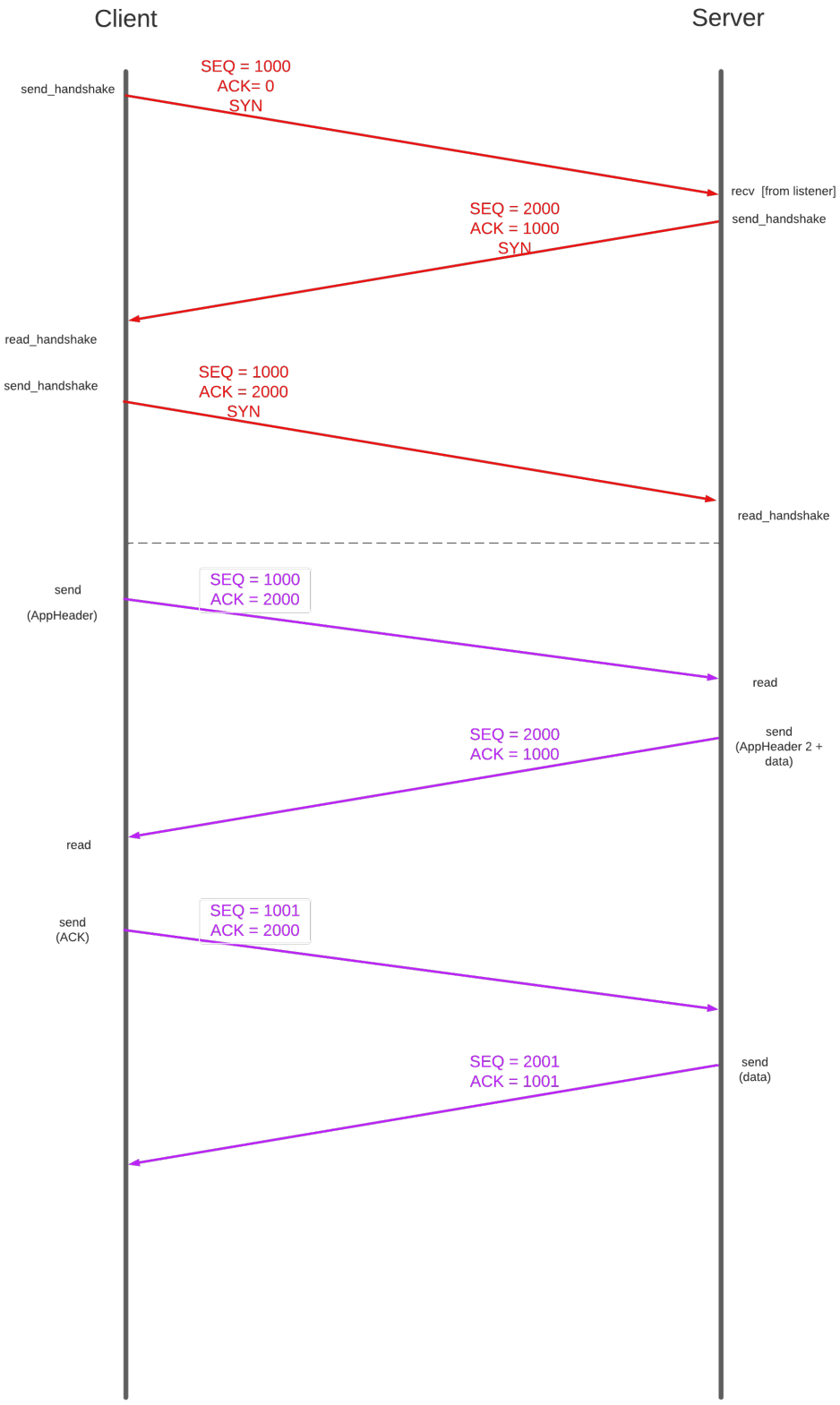

Figura 2: Comunicación Cliente-Servidor Stop and Wait Download

$->$ **Caso: Cliente solicita subida**
Para este caso se definió el envió del objeto "*ApplicationHeaderRDT*", requiriendo solo ida (Cliente $->$ Server), de tal forma que el cliente pueda avisar al server sobre la información del archivo que quiere subir. En la respuesta recibida por el servidor puede obtener:

- Confirmación de que puede subir el archivo (por medio de un mensaje sin data, solo con el HeaderRDT para representar un ACK).
- Rechazo del pedido de subida (en caso de que el archivo exceda el tamaño máximo de almacenamiento permitido por el server, definido arbitrariamente como 500 MiB).

En la siguiente página se muestra el comienzo de la comunicación del caso recién descrito:

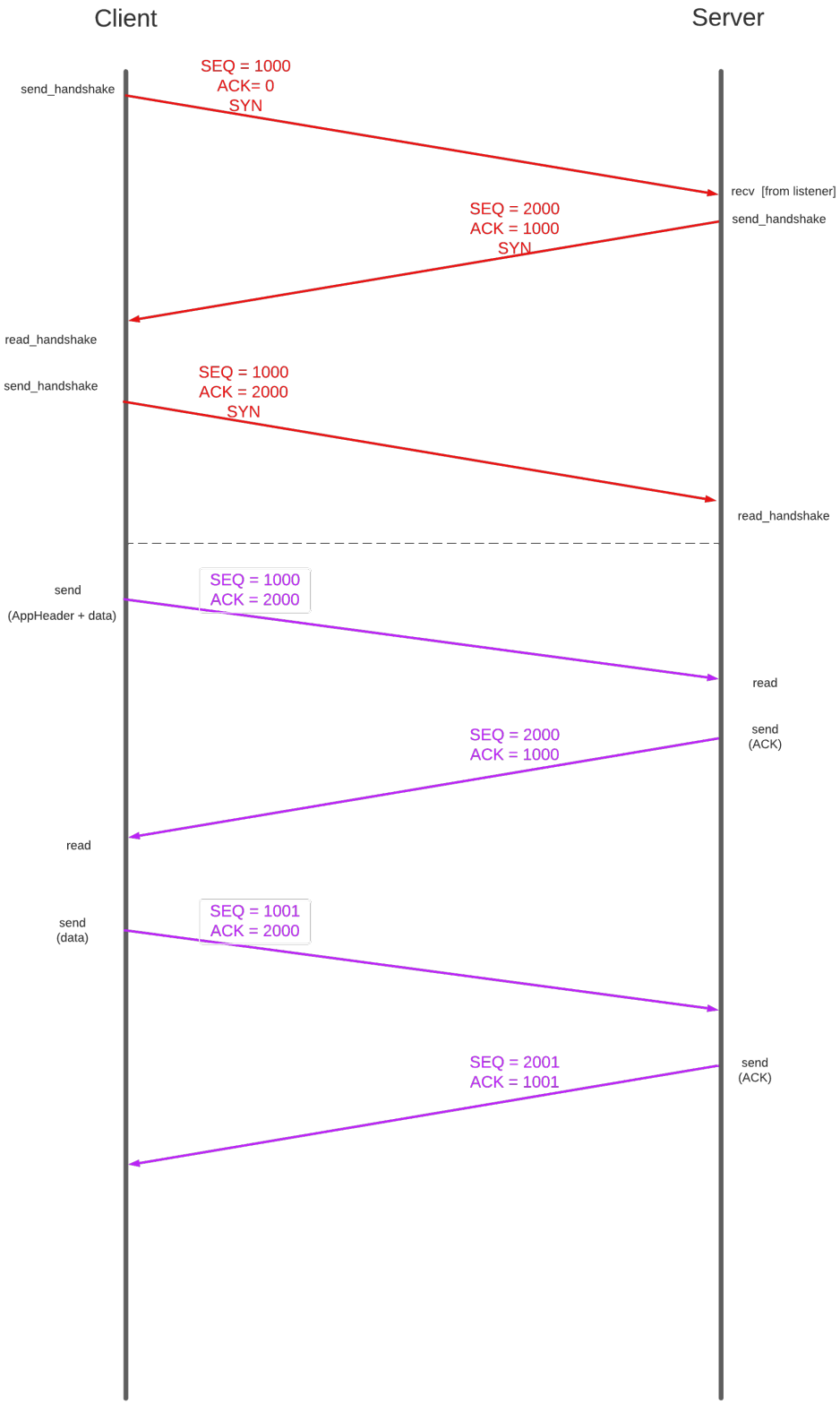

Figura 3: Comunicación Cliente-Servidor Stop and Wait Upload

A continuación se detalla la implementación de una de las secciones más importante del desarrollo del protocolo: El manejo de envío de paquetes por medio de Selective Repeat.

Dicho protocolo se implementó por medio del modelado de una SlindingWindow encargada de mantener información del estado de una cantidad fija de paquetes que puede ser enviada, por medio del almacenamiento de sus correspondientes sequence numbers locales, ack numbers, y una lista con registros para identificar si determinado paquete contenido en la window ya fue enviado o si ya se recibió su correspondiente mensaje ACK.

La secuencia de envío se hace de tal forma que se pueden enviar de forma eficiente los paquetes contenidos en la window actual, realizando una recepción (read) no bloqueante por cada uno de dichos envíos.

De esta forma se pueden continuar recibiendo ACKs dinámicamente, y la window se va a "desplazar.en el caso en el que su paquete contenido de menor sequence number ya se haya informado como recibido por el host contrario.

Finalmente, si se agotan los posibles envíos de la window (es decir, si para la window actual ya se enviaron todos los paquetes pero aún no se recibió el ACK necesario para desplazarla), se procede a realizar una recepción bloqueante (con un timeout y cierta cantidad de reintentos en caso del mismo) para permitir la recepción de posibles ACKs que aun no hubiesen llegado.

Si se da el caso de timeout se procede a reenviar selectivamente los paquetes que no se hayan confirmado como recibidos por el host contrario (ya sea porque se perdieron dichos paquetes o porque se hayan perdido los correspondientes ACKs. En este último caso si se llegan a obtener paquetes replicados simplemente son descartados por el host que corresponda).

De esta forma no solo se garantiza el envío confiable de paquetes en situaciones con packet loss, sino que tambien se tiene en cuenta la eficiencia al enviar selectivamente los paquetes necesarios a retransmitir en caso de perdidas.

En la siguiente página se muestra un gráfico ilustrativo de la comunicación anterior:

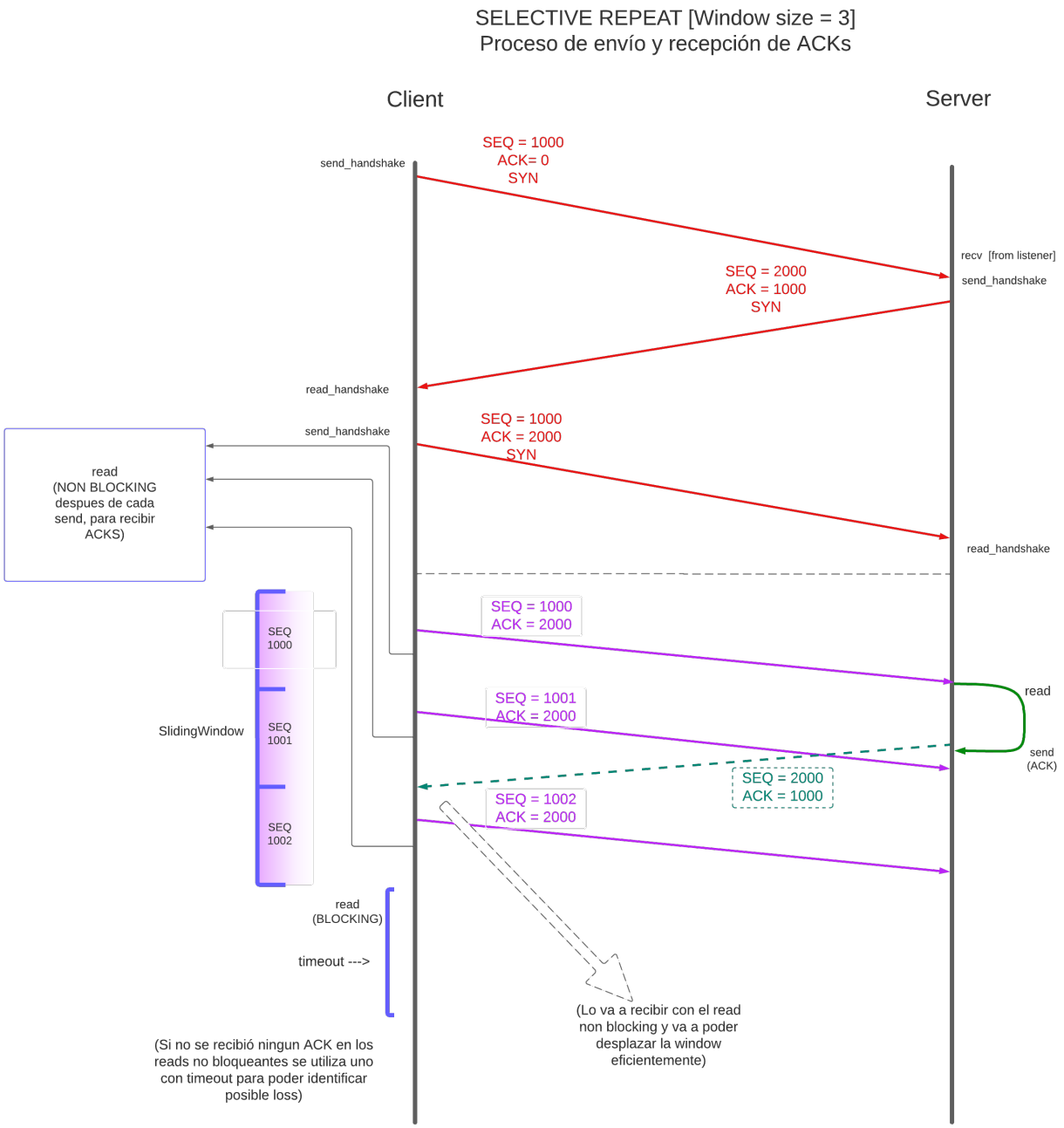

Figura 4: Comunicación Cliente-Servidor Stop and Wait Upload

Notas adicionales de la implementación

- **Implementación de los streams de conexión** Una parte importante del desarrollo del programa fue el modelado e implementación de las entidades ListenerRDT, AccepterRDT y StreamRDT. Como se esperaría (tomando como ejemplo a las conexiones TCP), el rol del Listener es poder escuchar intentos de comunicación entrantes. Posteriormente el Accepter se encarga de culminar el establecimiento de conexión por medio de la creación de una instancia de StreamRDT para tener un canal de comunicación directa con el cliente particular que solicitó la conexión. Luego, el Stream es el ente más importante ya que se debe encargar del manejo completo de las transferencias de paquetes (haciendo uso de los servicios que le brinde su protocolo, sea Stop And Wait o Selective Repeat), incluyendo recepción y envío seguro de los mismos.

- **Implementación concurrente del servidor**
  Para el desarrollo del manejo concurrente de clientes por parte del servidor, se implementó el mismo de tal forma que se crea un thread por cada recepción de intento de conexión distinto por cada cliente hasta un determinado máximo. De esta manera se pueden realizar las operaciones requeridas por cada conexión de forma completamente independiente. Esto último es facilitado por el hecho de que los clientes (para este programa particular) no requieren de comunicarse entre sí, por lo cual su comunicación es exclusiva con el servidor y su request específica.

- **Stop And Wait como caso particular**
  Dado que nuestra implementación del manejo de packet loss y errores Selective Repeat abarca el uso de la lógica de timeouts y reintentos para el envío de multiples paquetes (dependiendo del tamaño de la window), nos resultó lógico y viable realizar la implementación de Stop and Wait como caso particular de Selective Repeat con un tamaño de window = 1.

- **Utilización de librerías**
  Si bien en el entorno controlado de pruebas (virtualización en Mininet) no puede llegar a darse una corrupción de bits en los paquetes transferidos, se añadió adicionalmente el uso de la librería crc de Python para corroborar la integridad de los headers pertinentes de cada paquete enviado y recibido por cada host al momento de decodificar los bits.

- **Wireshark Dissector**
  Adicionalmente se implementó un disector de Wireshark para que el programa sea capaz de interpretar el HeaderRDT que se adhiere a cada paquete transmitido por medio de nuestro protocolo. En la interpretación de cada paquete se pueden observar todos los correspondientes campos del segmento que se mencionaron anteriormente en la arquitectura de segmentos implementada.

4. Pruebas

Se toman en cuenta los siguientes casos de prueba para validación de comportamiento general (cada caso se describe por medio de la topología probada y distintos valores de *packet loss*, así como secuencias de ejecución distintas):

Nota aclaratoria: En las pruebas mostradas a continuación se utiliza un archivo llamado "*upload test*", autogenerado con una utilidad de linea de comandos Linux. (Aclarado para no causar confusión dado que tambien se utiliza en las pruebas de downloads) El mismo tiene un tamaño al alrededor de 5 KiB.

Topología: Server $<->$ 1 Cliente

- **packet loss = 0**: (para observación del comportamiento ideal)

```
miguelv5 ~/fiuba/intro_distrib/tp1/repo  improve ± sudo mn --custom ./src/topologia.py --topo customTopo,num_clients=1,loss_percent=0 --mac -x
*** No default OpenFlow controller found for default switch!
*** Falling back to OVS Bridge
*** Creating network
*** Adding controller
*** Adding hosts:
h0 h1
*** Adding switches:
s1
*** Adding links:
(h0, s1) (s1, h1)
*** Configuring hosts
h0 h1
*** Running terms on :0
*** Starting controller

*** Starting 1 switches
s1 ...
*** Starting CLI:
mininet>
```

Figura 5: Mininet Topologia Packet loss 0 %

- UPLOAD del archivo (Client $->$ Server)

```
"host: h0"
# python3 src/start-server.py -sr -v -H 10.0.0.1
"host: h1"
# python3 src/upload.py -v -H 10.0.0.1 -n upload_test -s ./misc/upload_test -sr
```

Figura 6: Comandos de ejecucción de servidor y cliente

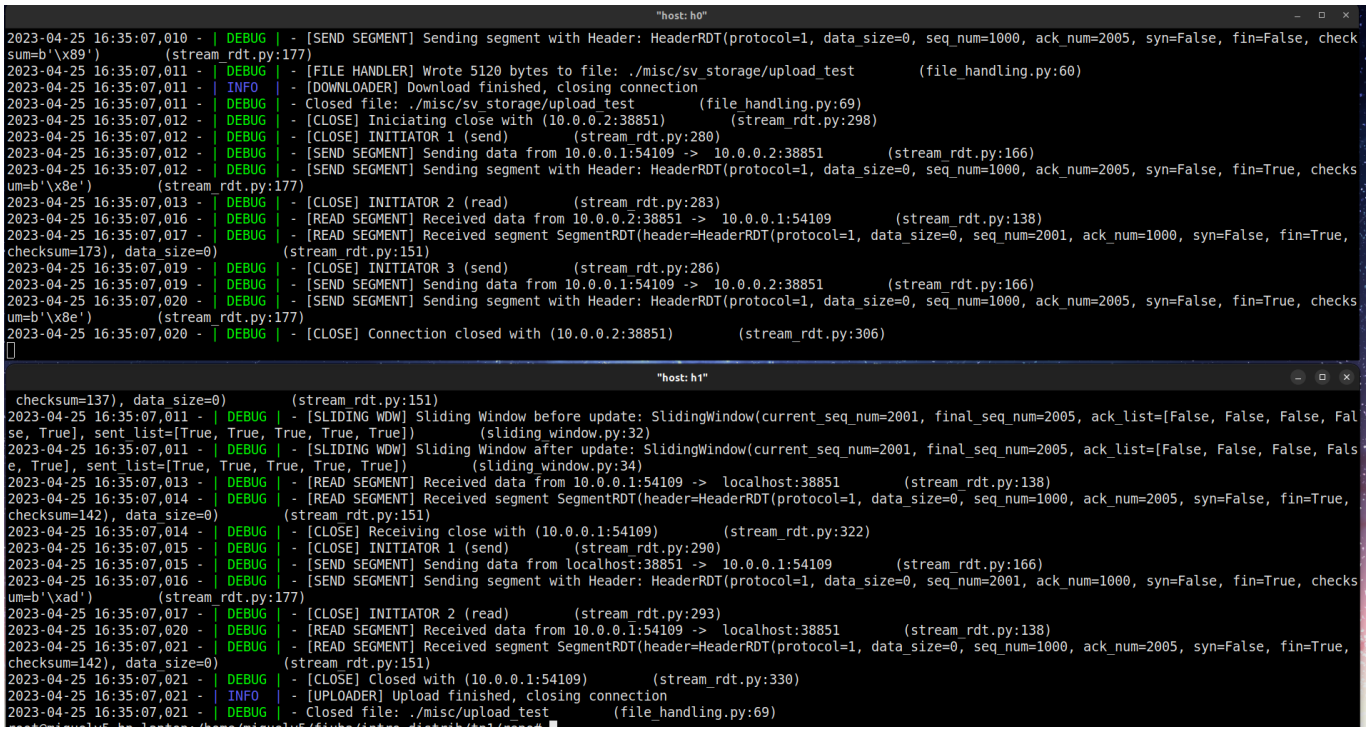

Figura 7: Captura de comunicación por linea de comandos

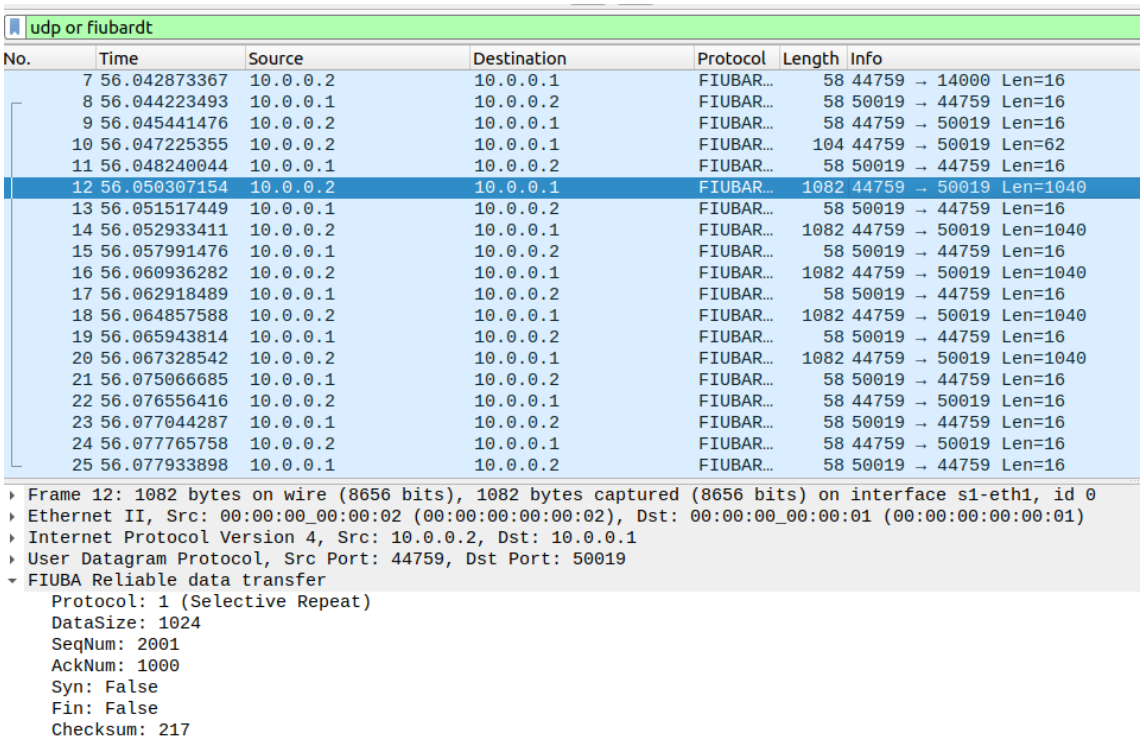

Figura 8: Captura de wireshark del protocolo

```
 miguelv5  ~/fiuba/intro_distrib/tp1/repo  improve ±  sha1sum ./misc/upload_test
ec8d8db07ace21ae014c4d7dbe42297dfe61976a  ./misc/upload_test
 miguelv5  ~/fiuba/intro_distrib/tp1/repo  improve ±  sha1sum ./misc/sv_storage/upload_test
ec8d8db07ace21ae014c4d7dbe42297dfe61976a  ./misc/sv_storage/upload_test
```

Figura 9: Verificación de SHA1 de los archivos

- DOWNLOAD del archivo (Client $<--$ Server)

```
"host: h0"
# python3 src/start-server.py -sr -v -H 10.0.0.1

"host: h1"
# python3 src/download.py -v -H 10.0.0.1 -n upload_test -d ./misc/downloads/upload_test -sr
```

Figura 10: Comandos de ejecucción de servidor y cliente

```
"host: h0"
2023-04-25 16:36:45,844 - | DEBUG | - [SLIDING WDW] Sliding Window before update: SlidingWindow(current_seq_num=1001, final_seq_num=1005, ack_list=[False, False, False, True, True], sent_list=[True, True, True, True, True])        (sliding_window.py:32)
2023-04-25 16:36:45,845 - | DEBUG | - [SLIDING WDW] Sliding Window after update: SlidingWindow(current_seq_num=1001, final_seq_num=1005, ack_list=[False, False, False, True, True], sent_list=[True, True, True, True, True])        (sliding_window.py:34)
2023-04-25 16:36:45,845 - | DEBUG | - [READ SEGMENT] Received data from 10.0.0.2:42374 ->  10.0.0.1:56893        (stream_rdt.py:138)
2023-04-25 16:36:45,846 - | DEBUG | - [READ SEGMENT] Received segment SegmentRDT(header=HeaderRDT(protocol=1, data_size=0, seq_num=2001, ack_num=1005, syn=False, fin=True, checksum=109), data_size=0)        (stream_rdt.py:151)
2023-04-25 16:36:45,846 - | DEBUG | - [CLOSE] Receiving close with (10.0.0.2:42374)        (stream_rdt.py:322)
2023-04-25 16:36:45,846 - | DEBUG | - [CLOSE] INITIATOR 1 (send)        (stream_rdt.py:290)
2023-04-25 16:36:45,846 - | DEBUG | - [SEND SEGMENT] Sending data from 10.0.0.1:56893 ->  10.0.0.2:42374        (stream_rdt.py:166)
2023-04-25 16:36:45,847 - | DEBUG | - [SEND SEGMENT] Sending segment with Header: HeaderRDT(protocol=1, data_size=0, seq_num=1001, ack_num=2000, syn=False, fin=True, checksum=b'\x91')        (stream_rdt.py:177)
2023-04-25 16:36:45,848 - | DEBUG | - [CLOSE] INITIATOR 2 (read)        (stream_rdt.py:293)
2023-04-25 16:36:45,850 - | DEBUG | - [READ SEGMENT] Received data from 10.0.0.2:42374 ->  10.0.0.1:56893        (stream_rdt.py:138)
2023-04-25 16:36:45,851 - | DEBUG | - [READ SEGMENT] Received segment SegmentRDT(header=HeaderRDT(protocol=1, data_size=0, seq_num=2001, ack_num=1005, syn=False, fin=True, checksum=109), data_size=0)        (stream_rdt.py:151)
2023-04-25 16:36:45,852 - | DEBUG | - [CLOSE] Closed with (10.0.0.2:42374)        (stream_rdt.py:330)
2023-04-25 16:36:45,853 - | INFO  | - [UPLOADER] Upload finished, closing connection
2023-04-25 16:36:45,854 - | DEBUG | - Closed file: ./misc/sv_storage/upload_test        (file_handling.py:69)

"host: h1"
2023-04-25 16:36:45,843 - | DEBUG | - [SEND SEGMENT] Sending data from localhost:42374 ->  10.0.0.1:56893        (stream_rdt.py:166)
2023-04-25 16:36:45,843 - | DEBUG | - [SEND SEGMENT] Sending segment with Header: HeaderRDT(protocol=1, data_size=0, seq_num=2001, ack_num=1005, syn=False, fin=False, checksum=b'j')        (stream_rdt.py:177)
2023-04-25 16:36:45,843 - | DEBUG | - [FILE HANDLER] Wrote 5120 bytes to file: ./misc/downloads/upload_test        (file_handling.py:60)
2023-04-25 16:36:45,844 - | INFO  | - [DOWNLOADER] Download finished, closing connection
2023-04-25 16:36:45,844 - | DEBUG | - Closed file: ./misc/downloads/upload_test        (file_handling.py:69)
2023-04-25 16:36:45,844 - | DEBUG | - [CLOSE] Iniciating close with (10.0.0.1:56893)        (stream_rdt.py:298)
2023-04-25 16:36:45,844 - | DEBUG | - [CLOSE] INITIATOR 1 (send)        (stream_rdt.py:280)
2023-04-25 16:36:45,844 - | DEBUG | - [SEND SEGMENT] Sending data from localhost:42374 ->  10.0.0.1:56893        (stream_rdt.py:166)
2023-04-25 16:36:45,845 - | DEBUG | - [SEND SEGMENT] Sending segment with Header: HeaderRDT(protocol=1, data_size=0, seq_num=2001, ack_num=1005, syn=False, fin=True, checksum=b'm')        (stream_rdt.py:177)
2023-04-25 16:36:45,845 - | DEBUG | - [CLOSE] INITIATOR 2 (read)        (stream_rdt.py:283)
2023-04-25 16:36:45,847 - | DEBUG | - [READ SEGMENT] Received data from 10.0.0.1:56893 ->  localhost:42374        (stream_rdt.py:138)
2023-04-25 16:36:45,848 - | DEBUG | - [READ SEGMENT] Received segment SegmentRDT(header=HeaderRDT(protocol=1, data_size=0, seq_num=1001, ack_num=2000, syn=False, fin=True, checksum=145), data_size=0)        (stream_rdt.py:151)
2023-04-25 16:36:45,849 - | DEBUG | - [CLOSE] INITIATOR 3 (send)        (stream_rdt.py:286)
2023-04-25 16:36:45,849 - | DEBUG | - [SEND SEGMENT] Sending data from localhost:42374 ->  10.0.0.1:56893        (stream_rdt.py:166)
2023-04-25 16:36:45,850 - | DEBUG | - [SEND SEGMENT] Sending segment with Header: HeaderRDT(protocol=1, data_size=0, seq_num=2001, ack_num=1005, syn=False, fin=True, checksum=b'm')        (stream_rdt.py:177)
2023-04-25 16:36:45,851 - | DEBUG | - [CLOSE] Connection closed with (10.0.0.1:56893)        (stream_rdt.py:306)
```

Figura 11: Captura de comunicación por linea de comandos

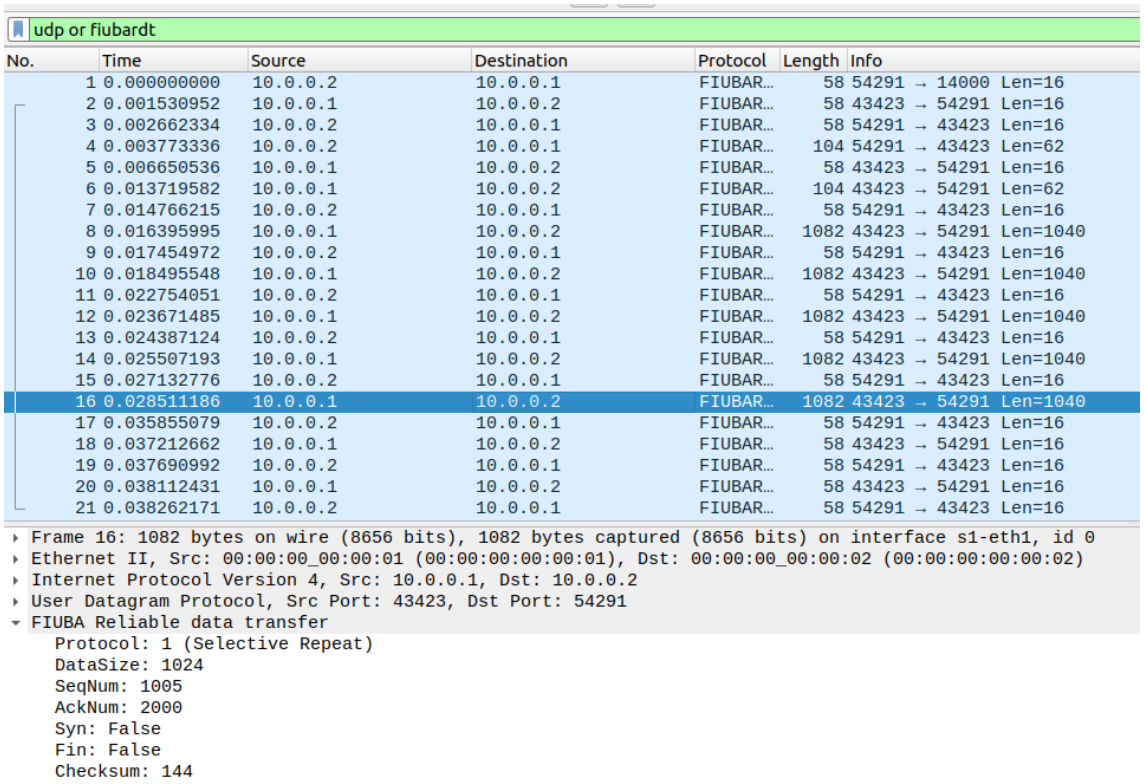

Figura 12: Captura de wireshark del protocolo

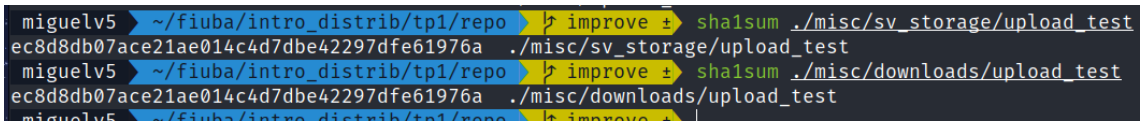

Figura 13: Verificación de SHA1 de los archivos

- **packet loss = 10**:

```
miguelv5 ~/fiuba/intro_distrib/tp1/repo improve sudo mn --custom ./src/topologia.py --topo customTopo,num_clients=1,loss_percent=10 --mac -x
[sudo] password for miguelv5:
*** No default OpenFlow controller found for default switch!
*** Falling back to OVS Bridge
*** Creating network
*** Adding controller
*** Adding hosts:
h0 h1
*** Adding switches:
s1
*** Adding links:
(10.00000% loss) (10.00000% loss) (h0, s1) (10.00000% loss) (10.00000% loss) (s1, h1)
*** Configuring hosts
h0 h1
*** Running terms on :0
*** Starting controller

*** Starting 1 switches
s1 ...(10.00000% loss) (10.00000% loss)
*** Starting CLI:
mininet>
```

Figura 14: Mininet Topologia Packet loss 10 %

- UPLOAD del archivo (Client $--> $ Server)

```
"host: h0"
# python3 src/start-server.py -sr -v -H 10.0.0.1

"host: h1"
# python3 src/upload.py -v -H 10.0.0.1 -n upload_test -s ./misc/upload_test -sr
```

Figura 15: Comandos de ejecución de servidor y cliente

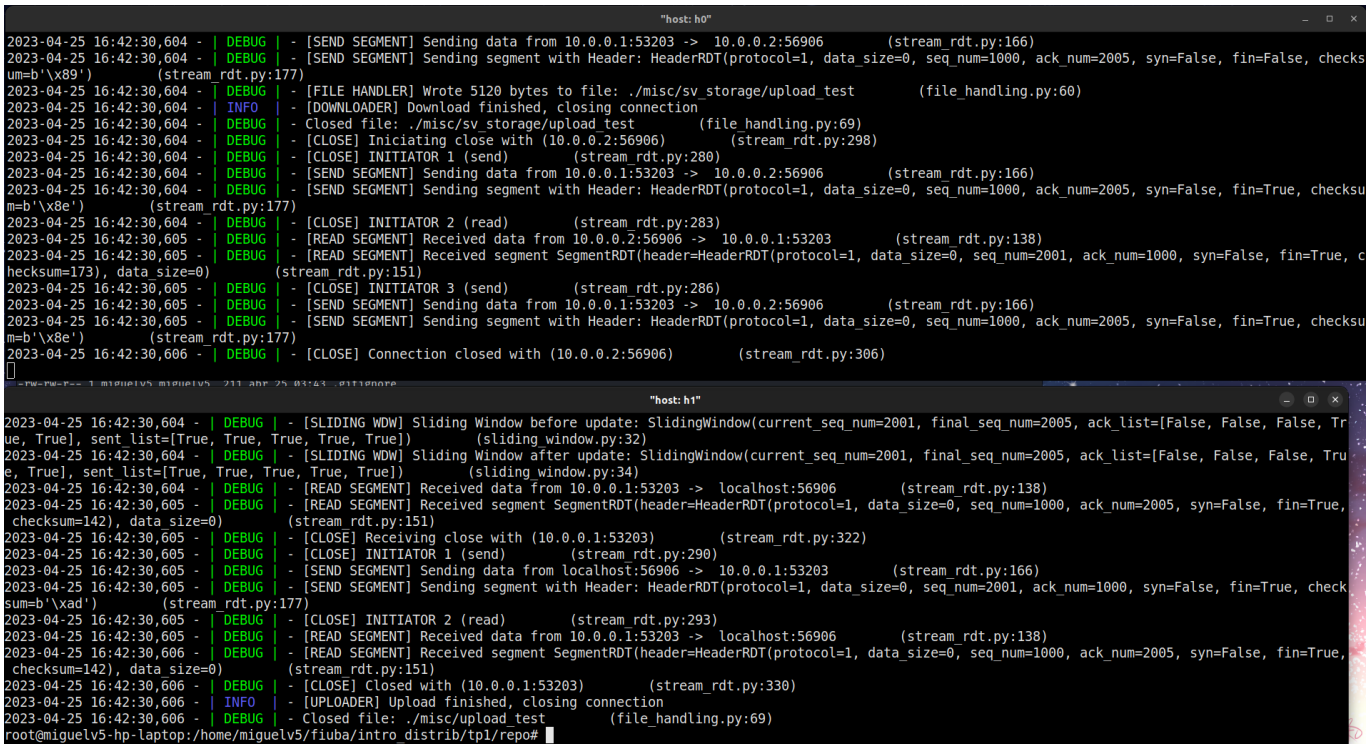

Figura 16: Captura de comunicación por linea de comandos

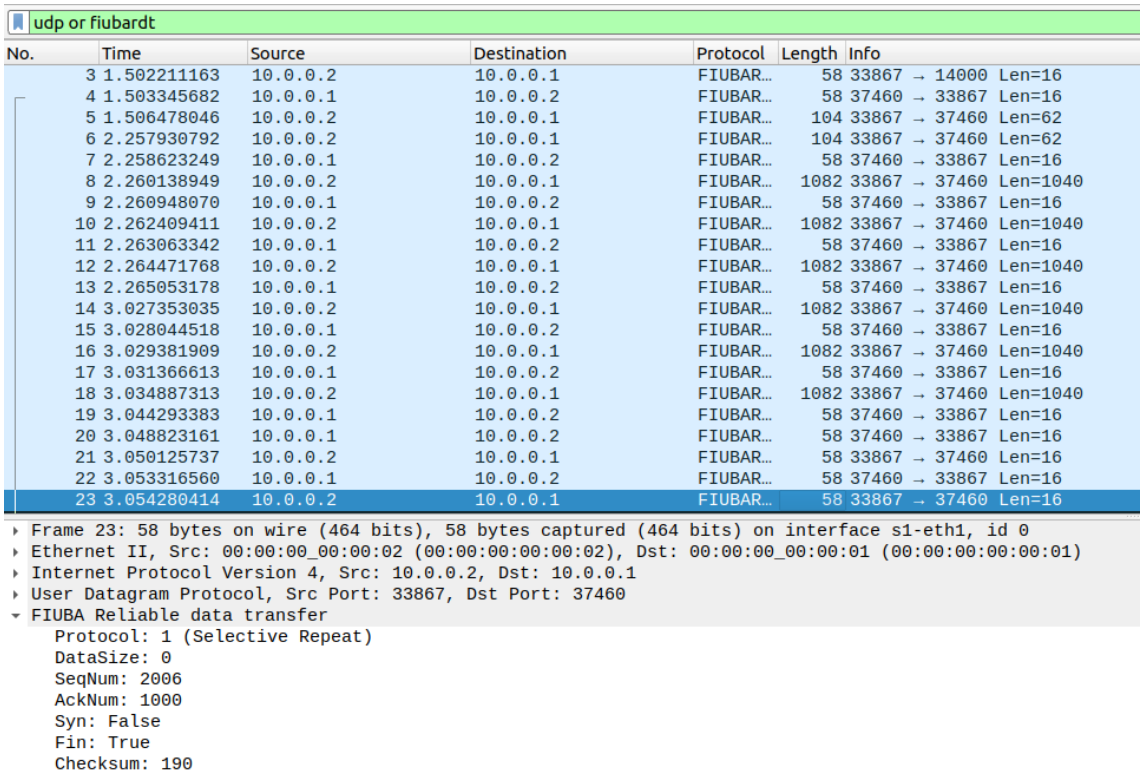

Figura 17: Captura de wireshark del protocolo

```
 miguelv5  ~/fiuba/intro_distrib/tp1/repo  improve ±  sha1sum ./misc/upload_test
ec8d8db07ace21ae014c4d7dbe42297dfe61976a  ./misc/upload_test
 miguelv5  ~/fiuba/intro_distrib/tp1/repo  improve ±  sha1sum ./misc/sv_storage/upload_test
ec8d8db07ace21ae014c4d7dbe42297dfe61976a  ./misc/sv_storage/upload_test
```

Figura 18: Verificación de SHA1 de los archivos

- DOWNLOAD del archivo (Client $<--$ Server)

```
"host: h0"
# python3 src/start-server.py -sr -v -H 10.0.0.1

"host: h1"
# python3 src/download.py -v -H 10.0.0.1 -n upload_test -d ./misc/downloads/upload_test -sr
```

Figura 19: Comandos de ejecucción de servidor y cliente

```
"host: h0"
checksum=106), data_size=0)        (stream_rdt.py:151)
2023-04-25 16:44:18,240 - | DEBUG | - [SLIDING WDW] Sliding Window before update: SlidingWindow(current_seq_num=1001, final_seq_num=1005, ack_list=[False, False, False, True
, True], sent_list=[True, True, True, True, True])        (sliding_window.py:32)
2023-04-25 16:44:18,240 - | DEBUG | - [SLIDING WDW] Sliding Window after update: SlidingWindow(current_seq_num=1001, final_seq_num=1005, ack_list=[False, False, False, True,
 True], sent_list=[True, True, True, True, True])        (sliding_window.py:34)
2023-04-25 16:44:18,240 - | DEBUG | - [READ SEGMENT] Received data from 10.0.0.2:59496 ->  10.0.0.1:40989        (stream_rdt.py:138)
2023-04-25 16:44:18,241 - | DEBUG | - [READ SEGMENT] Received segment SegmentRDT(header=HeaderRDT(protocol=1, data_size=0, seq_num=2001, ack_num=1005, syn=False, fin=True, c
hecksum=109), data_size=0)        (stream_rdt.py:151)
2023-04-25 16:44:18,241 - | DEBUG | - [CLOSE] Receiving close with (10.0.0.2:59496)        (stream_rdt.py:322)
2023-04-25 16:44:18,241 - | DEBUG | - [CLOSE] INITIATOR 1 (send)        (stream_rdt.py:290)
2023-04-25 16:44:18,241 - | DEBUG | - [SEND SEGMENT] Sending data from 10.0.0.1:40989 ->  10.0.0.2:59496        (stream_rdt.py:166)
2023-04-25 16:44:18,241 - | DEBUG | - [SEND SEGMENT] Sending segment with Header: HeaderRDT(protocol=1, data_size=0, seq_num=1001, ack_num=2000, syn=False, fin=True, checksu
m=b'\x91')        (stream_rdt.py:177)
2023-04-25 16:44:18,241 - | DEBUG | - [CLOSE] INITIATOR 2 (read)        (stream_rdt.py:293)
2023-04-25 16:44:18,242 - | DEBUG | - [READ SEGMENT] Received data from 10.0.0.2:59496 ->  10.0.0.1:40989        (stream_rdt.py:138)
2023-04-25 16:44:18,242 - | DEBUG | - [READ SEGMENT] Received segment SegmentRDT(header=HeaderRDT(protocol=1, data_size=0, seq_num=2001, ack_num=1005, syn=False, fin=True, c
hecksum=109), data_size=0)        (stream_rdt.py:151)
2023-04-25 16:44:18,242 - | DEBUG | - [CLOSE] Closed with (10.0.0.2:59496)        (stream_rdt.py:330)
2023-04-25 16:44:18,243 - | INFO  | - [UPLOADER] Upload finished, closing connection
2023-04-25 16:44:18,243 - | DEBUG | - Closed file: ./misc/sv_storage/upload_test        (file_handling.py:69)

"host: h1"
2023-04-25 16:44:18,239 - | DEBUG | - [SEND SEGMENT] Sending segment with Header: HeaderRDT(protocol=1, data_size=0, seq_num=2001, ack_num=1005, syn=False, fin=False, chec
ksum=b'j')        (stream_rdt.py:177)
2023-04-25 16:44:18,240 - | DEBUG | - [FILE HANDLER] Wrote 5120 bytes to file: ./misc/downloads/upload_test        (file_handling.py:60)
2023-04-25 16:44:18,240 - | INFO  | - [DOWNLOADER] Download finished, closing connection
2023-04-25 16:44:18,240 - | DEBUG | - Closed file: ./misc/downloads/upload_test        (file_handling.py:69)
2023-04-25 16:44:18,240 - | DEBUG | - [CLOSE] Iniciating close with (10.0.0.1:40989)        (stream_rdt.py:298)
2023-04-25 16:44:18,240 - | DEBUG | - [CLOSE] INITIATOR 1 (send)        (stream_rdt.py:280)
2023-04-25 16:44:18,240 - | DEBUG | - [SEND SEGMENT] Sending data from localhost:59496 ->  10.0.0.1:40989        (stream_rdt.py:166)
2023-04-25 16:44:18,241 - | DEBUG | - [SEND SEGMENT] Sending segment with Header: HeaderRDT(protocol=1, data_size=0, seq_num=2001, ack_num=1005, syn=False, fin=True, check
sum=b'm')        (stream_rdt.py:177)
2023-04-25 16:44:18,241 - | DEBUG | - [CLOSE] INITIATOR 2 (read)        (stream_rdt.py:283)
2023-04-25 16:44:18,241 - | DEBUG | - [READ SEGMENT] Received data from 10.0.0.1:40989 ->  localhost:59496        (stream_rdt.py:138)
2023-04-25 16:44:18,242 - | DEBUG | - [READ SEGMENT] Received segment SegmentRDT(header=HeaderRDT(protocol=1, data_size=0, seq_num=1001, ack_num=2000, syn=False, fin=True,
 checksum=145), data_size=0)        (stream_rdt.py:151)
2023-04-25 16:44:18,242 - | DEBUG | - [CLOSE] INITIATOR 3 (send)        (stream_rdt.py:286)
2023-04-25 16:44:18,242 - | DEBUG | - [SEND SEGMENT] Sending data from localhost:59496 ->  10.0.0.1:40989        (stream_rdt.py:166)
2023-04-25 16:44:18,242 - | DEBUG | - [SEND SEGMENT] Sending segment with Header: HeaderRDT(protocol=1, data_size=0, seq_num=2001, ack_num=1005, syn=False, fin=True, check
sum=b'm')        (stream_rdt.py:177)
2023-04-25 16:44:18,242 - | DEBUG | - [CLOSE] Connection closed with (10.0.0.1:40989)        (stream_rdt.py:306)
root@miguelv5-hp-laptop:/home/miguelv5/fiuba/intro_distrib/tp1/repo#
```

Figura 20: Captura de comunicación por linea de comandos

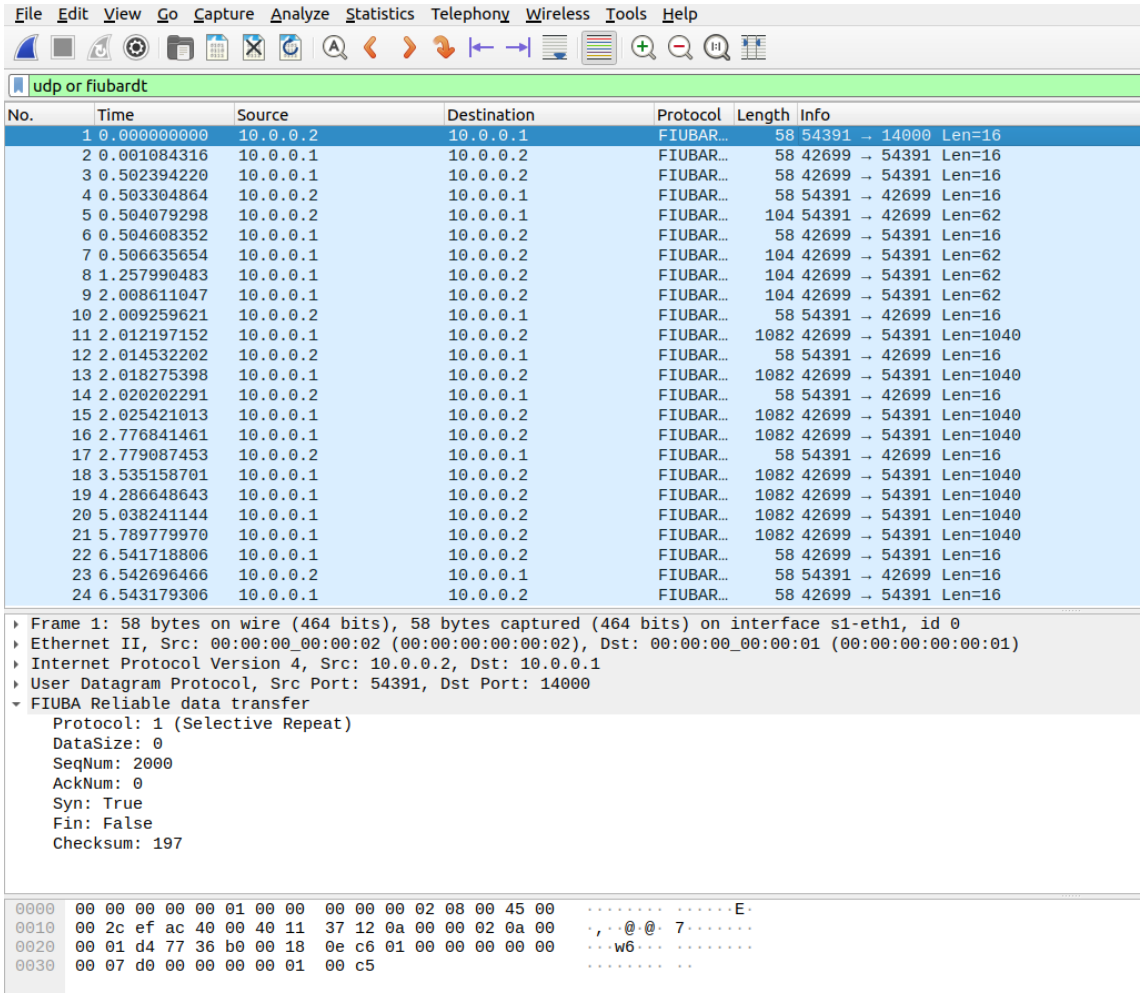

Figura 21: Captura de wireshark del protocolo

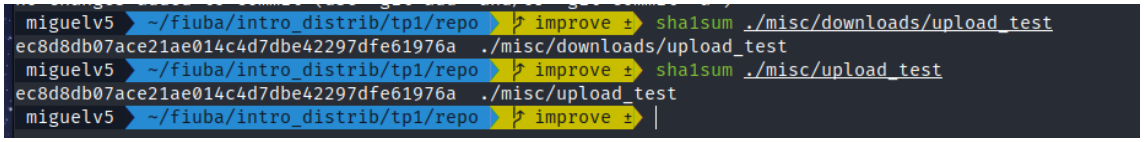

Figura 22: Verificación de SHA1 de los archivos

Para la correcta validación de las transferencias se utilizó el comando **sha1sum**. El comando sha1sum se encuentra en los sistemas Unix y permite identificar la integridad de un fichero mediante la comprobación del hash encodificado en **SHA-1**.

5. Preguntas a responder

Preguntas

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

Respuestas

1. La arquitectura cliente-servidor es un modelo de red en el que los procesos de la aplicación se dividen en dos categorías: el cliente y el servidor. El servidor es un programa que espera las solicitudes de los clientes y responde a ellas. Por otro lado, el cliente es un programa que envía solicitudes al servidor y espera a que éste le devuelva una respuesta.

   En esta arquitectura, los servidores se ejecutan en máquinas dedicadas y están diseñados para atender solicitudes de múltiples clientes al mismo tiempo. Los clientes, por otro lado, pueden ser programas que se ejecutan en la misma máquina que el servidor o en máquinas remotas.

   Una de las características clave de la arquitectura cliente-servidor es que los clientes no interactúan directamente entre sí. En lugar de eso, cualquier comunicación entre clientes se realiza a través del servidor. Esto significa que el servidor actúa como un intermediario entre los clientes, lo que puede simplificar el diseño y la implementación de la aplicación.

   En la arquitectura cliente-servidor, la comunicación entre el cliente y el servidor se lleva a cabo mediante el intercambio de mensajes. Los clientes envían solicitudes al servidor en forma de mensajes, y el servidor responde a esas solicitudes enviando mensajes de respuesta. Estos mensajes pueden ser de diferentes tipos, como mensajes de solicitud, mensajes de respuesta, mensajes de confirmación, etc.

   Para que la comunicación cliente-servidor se realice correctamente, se utilizan varios protocolos. Los protocolos definen las reglas y el formato de los mensajes que se intercambian entre el cliente y el servidor, y también definen el comportamiento de los clientes y los servidores en diferentes situaciones.

   Entonces, la arquitectura cliente-servidor es un modelo de red en el que los procesos de la aplicación se dividen en dos categorías: el cliente y el servidor. Los clientes envían solicitudes al servidor y esperan a que éste les devuelva una respuesta. El servidor, por su parte, está diseñado para atender solicitudes de múltiples clientes al mismo tiempo y actúa como intermediario entre los clientes. La comunicación entre el cliente y el servidor se lleva a cabo mediante el intercambio de mensajes, y se utilizan varios protocolos para garantizar una comunicación efectiva.

2. La función principal de un protocolo de capa de aplicación es proporcionar servicios de comunicación a las aplicaciones que se ejecutan en diferentes hosts en la red y definir las reglas y los formatos para el intercambio de datos entre las aplicaciones.

   El objetivo principal de un protocolo de capa de aplicación es proporcionar servicios de comunicación a las aplicaciones que se ejecutan en diferentes hosts en la red.

   En otras palabras, la capa de aplicación es la capa superior del modelo de referencia OSI y TCP/IP, y está diseñada para proporcionar una interfaz entre las aplicaciones y la red subyacente.

Los protocolos de capa de aplicación se encargan de los detalles de la comunicación de extremo a extremo, incluyendo la codificación, el formato y el procesamiento de los mensajes enviados y recibidos por las aplicaciones.

Además de proporcionar servicios de comunicación a las aplicaciones, los protocolos de capa de aplicación también definen las reglas y los formatos para el intercambio de datos entre las aplicaciones. Por ejemplo, algunos protocolos de aplicación comunes incluyen el protocolo HTTP para la transferencia de documentos web, el protocolo FTP para la transferencia de archivos y el protocolo SMTP para el correo electrónico.

En nuestro caso particular, como se detalla anteriormente en la sección de Implementación, se define nuestro protocolo de aplicación cuando se hace referencia exclusivamente al envío de "$ApplicationHeaderRDT$" con datos de archivos subsecuentes.

3. El protocolo de aplicación del trabajo hace uso de un encabezado especifico que denota los siguientes campos:

   - El nombre del archivo
   - El tamaño del archivo
   - El tipo de interacción, *upload* o *download*

   Estos campos se utilizan en la capa de aplicación para reunir los datos de los segmentos y corroborar su tamaño final. A la capa de aplicación le llegan datos ordenados de la capa de transporte de a partes, para que luego vaya escribiendo el archivo abierto. El protocolo que define la secuencia de envío de mensajes de nuestra aplicación se encuentra definido en la sección Implementación de este informe.

4. TCP (Transmission Control Protocol) y UDP (User Datagram Protocol) son los dos protocolos ofrecidos por la capa de transporte del stack TCP/IP. Ambos protocolos tienen diferentes características y se utilizan en diferentes situaciones.

   TCP es un protocolo orientado a la conexión que proporciona una comunicación confiable y ordenada de extremo a extremo. TCP asegura que todos los datos se envíen y se reciban correctamente, y en el orden correcto. Para lograr esto, TCP utiliza un sistema de confirmación y retransmisión de paquetes que garantiza que los datos lleguen a su destino sin errores. TCP es ampliamente utilizado por aplicaciones que requieren una transmisión de datos confiable y en orden, como la transferencia de archivos y el correo electrónico.

   Por otro lado, UDP es un protocolo sin conexión que proporciona una comunicación no confiable y no ordenada de extremo a extremo. A diferencia de TCP, UDP no utiliza confirmaciones ni retransmisiones de paquetes, por lo que no se garantiza que los datos lleguen a su destino sin errores o en el orden correcto. UDP se utiliza en situaciones en las que la velocidad y la eficiencia son más importantes que la confiabilidad, como en la transmisión de video y audio en tiempo real.

6. Dificultades encontradas

Durante el desarrollo e implementación del protocolo y las distintas capas, nos encontramos con diferentes dificultades.

Una de estas dificultades estuvo asociada la las pruebas de transferencia de archivos. Cuando se hacen este tipo de pruebas con un porcentaje de pérdida de paquetes, puede ser muy difícil reproducir errores específicos. Esto se debe a que el porcentaje de pérdida ejerce un peso probabilístico sobre la pérdida de cada paquete individual. Además, es difícil hacer estas pruebas con múltiples clientes conectados al servidor al mismo tiempo, ya que esto agrega un factor de incertidumbre inherente a la concurrencia, lo que limita la validez de la prueba.

Para abordar este problema, se decidió implementar un comportamiento similar al del protocolo TCP mediante la creación de un *three-way-handshake* para establecer una conexión directa con un *socket* específico del servidor.

Adicionalmente, durante el proceso de implementación, surgieron varios desafíos relacionados con la configuración de los parámetros del sistema, como los *timeouts* para cada lectura y la cantidad de intentos permitidos después de dichos *timeouts*. Factores como la pérdida del primer paquete del *handshake* o la pérdida del último paquete del *handshake* también complicaron las cosas. Además, los retrasos no previstos, como los causados por la lectura y escritura de archivos en disco durante las transferencias, obligaron a ajustar los parámetros para evitar que los *timeouts* se excedieran en gran medida.

Para todo lo anterior fue de mucha utilidad la utilización de Wireshark para ver de forma sencilla el orden y todos los factores asociados a cada transmisión particular, y sumandole el disector personalizado facilitó aun más este proceso de análisis.

7. Conclusiones

Este trabajo logró desarrollar una solución eficiente y confiable para la transferencia de datos entre dispositivos en una red simulada con Mininet. La arquitectura implementada se basó en los principios de la arquitectura cliente-servidor, utilizando sockets en Python para la comunicación entre hosts.

Para corroborar el correcto envio y recepcion se agrego packet loss a la red simulada y aunque se encontraron dificultades en el manejo de paquetes y segmentos, se aplicaron técnicas de control de flujo y control de errores, como Stop-and-wait y Selective Repeat, para garantizar la correcta transferencia de los archivos y verificacion de headers mediante Cyclic Redundancy Checks.

En general, este trabajo permitió aplicar los conceptos de la materia y las notas de las clases a un problema real en el mundo de redes. La implementación de esta solución fue una oportunidad valiosa para poner en práctica los conocimientos adquiridos y enfrentar los desafíos que impone una implementación de una solución eficiente y confiable de transferencia de datos en una red de computadoras, y la complejidad que poseen los protocolos de capa de transporte que garantizan RDT (Reliable Data Transfer) por sobre el protocolo UDP que no ofrece ninguna ganrantía del estilo.

Adicionalmente permitió el aprendizaje de herramientas como Wireshark para realizar un análisis de disección de cada paquete que se registra como transferido por la red.

8. Enunciado

Introducción a los Sistemas Distribuidos (75.43)
TP N°1: File Transfer

Esteban Carisimo y Juan Ignacio Lopez Pecora

Facultad de Ingeniería, Universidad de Buenos Aires

12 de abril de 2023

Resumen

El presente trabajo práctico tiene como objetivo la creación de una aplicación de red. Para tal finalidad, será necesario comprender cómo se comunican los procesos a través de la red, y cuál es el modelo de servicio que la capa de transporte le ofrece a la capa de aplicación. Además, para poder lograr el objetivo planteado, se aprenderá el uso de la interfaz de sockets y los principios básicos de la transferencia de datos confiable (del inglés Reliable Data Transfer, RDT).

Palabras clave— Socket, protocolo, tcp, udp

1. Propuesta de trabajo

Este trabajo práctico se plantea como objetivo la comprensión y la puesta en práctica de los conceptos y herramientas necesarias para la implementación de un protocolo RDT. Para lograr este objetivo, se deberá desarrollar una aplicación de arquitectura cliente-servidor que implemente la funcionalidad de transferencia de archivos mediante las siguientes operaciones:

- **UPLOAD**: Transferencia de un archivo del cliente hacia el servidor
- **DOWNLOAD**: Transferencia de un archivo del servidor hacia el cliente

Dada las diferentes operaciones que pueden realizarse entre el cliente y el servidor, se requiere del diseño e implementación de un protocolo de aplicación básico que especifique los mensajes intercambiados entre los distintos procesos.

2. Herramientas a utilizar y procedimientos

La implementación de las aplicaciones solicitadas deben cumplir los siguientes requisitos:

- La aplicaciones deben ser desarrolladas en lenguaje Python [1] utilizando la librería estandard de sockets [2].
- La comunicación entre los procesos se debe implementar utilizando UDP como protocolo de capa de transporte.
- Para lograr una transferencia confiable al utilizar el protocolo UDP, se pide implementar una versión utilizando el protocolo *Stop & Wait* y otra versión utilizando el protocolo *Selective Repeat*.
- El servidor debe ser capaz de procesar de manera concurrente la transferencia de archivos con múltiples clientes.

2.1. Interfaz del cliente

La funcionalidad del cliente se divide en dos aplicaciónes de línea de comandos: `upload` y `download`. El comando `upload` envía un archivo al servidor para ser guardado con el nombre asignado. El listado 1 especifica la interfaz de linea de comandos para la operación de upload:

```
> python upload -h
usage: upload [-h] [-v | -q] [-H ADDR] [-p PORT] [-s FILEPATH] [-n FILENAME]

<command description>

optional arguments:
  -h, --help     show this help message and exit
  -v, --verbose  increase output verbosity
  -q, --quiet    decrease output verbosity
  -H, --host     server IP address
  -p, --port     server port
  -s, --src      source file path
  -n, --name     file name
```

Listing 1: Interfaz de linea de comandos de upload

El comando `download` descarga un archivo especificado desde el servidor. El listado 2 especifica la interfaz de linea de comandos para la operación de download:

```
> python download -h
usage: download [-h] [-v | -q] [-H ADDR] [-p PORT] [-d FILEPATH] [-n FILENAME]

<command description>

optional arguments:
  -h, --help     show this help message and exit
  -v, --verbose  increase output verbosity
  -q, --quiet    decrease output verbosity
  -H, --host     server IP address
  -p, --port     server port
  -d, --dst      destination file path
  -n, --name     file name
```

Listing 2: Interfaz de linea de comandos de download

2.2. Interfaz del servidor

El servidor provee el servicio de almacenamiento y descarga de archivos. El listado 3 especifica la interfaz de linea de comandos para el inicio del servidor:

```
> python start-server -h
usage: start-server [-h] [-v | -q] [-H ADDR] [-p PORT] [-s DIRPATH]

<command description>

optional arguments:
  -h, --help     show this help message and exit
  -v, --verbose  increase output verbosity
  -q, --quiet    decrease output verbosity
  -H, --host     service IP address
  -p, --port     service port
  -s, --storage  storage dir path
```

Listing 3: Interfaz de linea de comandos del servidor

2.3. Mininet

Se debe tener en cuenta que para ejecutar dicha arquitectura Cliente-Servidor se utilizará una herramienta conocida como Mininet.

Mininet es un conocido simulador de redes, cuya aparición se debe a la necesidad de contar con un simulador capaz de operar con distintos switches, entre ellos OpenFlow. El ingreso revolucionario de las SDN y la posibilidad de correr simulaciones, llevo a un gran crecimiento de mininet, por lo cual hoy es muy sencillo encontrar gran cantidad de información disponible. Para poder familiarizarnos nos mininet y su uso, hay un tutorial en el GitHub[1] oficial del proyecto. **A su vez, la cátedra presenta un breve tutorial de mininet en SDN, inspirado en el tutorial oficial.**

Mininet se utiliza de una forma radicalmente diferente al resto de los simuladores de red, ya que los dispositivos incluidos y su interconexión se definen a través de un script en Python.

- Abrir una nueva terminal e iniciar mininet con una topología simple, con 3 hosts conectados a un switch.

```
$ sudo mn --topo single,3 --mac
```

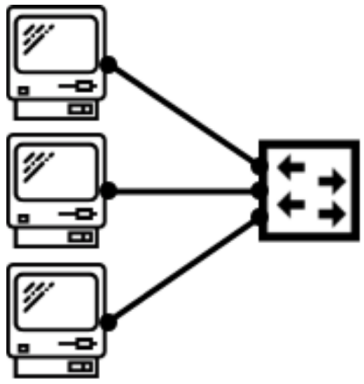

Figura 1: Topología Simple

- También se pueden crear topologías mas complejas, por ejemplo, se puede crear una topología propia denominada *FatTree* con el siguiente comando.

[1]Tutorial mininet con OpenFlow: `https://github.com/mininet/openflow-tutorial`

```
$ sudo mn --custom ~/mininet/custom/fattree.py --topo fattree --mac --arp --
    switch ovsk --controller remote
```

Para visualizar las topologías se puede usar la siguiente herramienta online

- `http://demo.spear.narmox.com/app/?apiurl=demo#!/mininet`

Para poder validar que el protocolo desarrollado provee garantía de entrega es necesario forzar la pérdida de paquetes. Cuando se crea la topología es necesario indicar a los enlaces el porcentaje de pérdida de paquetes que queremos. Para esto utilizaremos un parámetro a la hora de ejecutar mininet.

3. Ejercicios

Los grupos deberán elaborar un código capaz de generar la topología indicadas sobre la cual se realizaran diferentes ensayos. El día de la entrega se deberá ejecutar la topología y la ejecucion del protocolo confiable en clase, mostrando su funcionamiento y dando evidencias de que se conoce el uso de las herramientas dadas.

Además, la ejecución del programa debe ser por consola y se especifique por parámetro la cantidad de hosts especificados y el porcentaje de pérdida de paquetes de cada enlace en la topología. Para ejecutar Mininet se utilizará el siguiente comando:

```
$ sudo mn --custom ~/mininet/custom/custom.py --topo custom --mac -x
```

Se pedirá entregar el código y se exige no distribuirlo. Facilitar la distribución de código, como así también el uso de código desarrollado por otros, van en contra de la conducta que la Facultad de Ingeniería pretende de los alumnos. Más aún, nuestro objetivo en la cátedra es fomentar el aprendizaje y el interés por temas actuales e innovadores, por lo cual pretendemos que el desarrollo del trabajo sea motivador y enriquecedor para los alumnos.

3.1. Topología

Se propone desarrollar una topología parametrizable. Se tendrá una cantidad de hosts variable, formando una arquitectura en donde haya un único servidor que se encuentra conectado a la cantidad de clientes que se recibió como parámetro.

Asimismo, cada enlace que conecta a un cliente con el servidor, deberá tener la pérdida de paquetes que se introdujo como parametro en el comando de ejecución.

A continuación se muestra una clase con una topología que cuenta con un servidor conectado a tres hosts con una perdida de paquetes del 10 %.

```
from mininet.topo import Topo
from mininet.link import TCLink

class Topo( Topo ):
    def __init__( self ):
        # Initialize topology
        Topo.__init__( self )

        # Create hosts
        h1 = self.addHost('host_1'))
        h2 = self.addHost('host_2'))
        h3 = self.addHost('host_3'))
        h4 = self.addHost('host_4'))

        # Add links between server and hosts self.addLink(s1, s2)
        self.addLink(h1, h2, cls=TCLink, loss=10)
        self.addLink(h1, h3, cls=TCLink, loss=10)
        self.addLink(h1, h4, cls=TCLink, loss=10)

topos = { 'customTopo': Topo }
```

3.2. Protocolo de Entrega Confiable

Una vez verificado el correcto armado de la red, se utilizará dicha arquitectura para ejecutar distintas pruebas entre el o los clientes y el servidor. Para ello, se correará el código con el protocolo confiabale en dichos hosts y se transferirán distintos archivos de tamaño variable.

Una vez concluida la transferencia (subida y/o descarga) de un archivo, se verificará que el archivo no haya sido modificado.

4. Análisis

Comparar la performance de la versión Selective Repeat del protocolo y la versión Stop&Wait utilizando archivos de distintos tamaños y bajo distintas configuraciones de pérdida de paquetes.

5. Preguntas a responder

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

6. Entrega

La entrega consta de un informe, el código fuente de la aplicación desarrollada y un archivo README en formato MarkDown[3] con los detalles necesarios para ejecutar la aplicación. La codificación debe cumplir el standard PEP8 [4], para ello se sugiere utilizar el linter flake8 [5]. La figure 2 muestra la estructura y el contenido del archivo ZIP entregable.

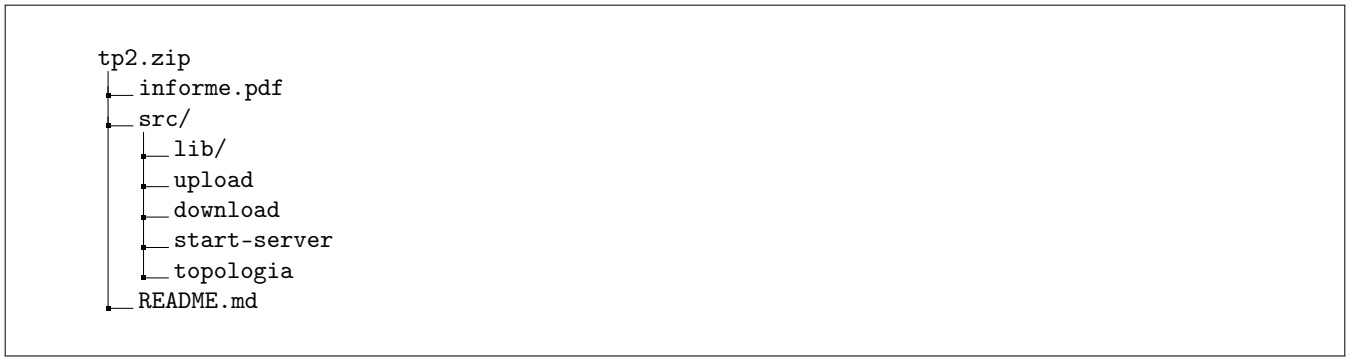

Figura 2: Estructura del archivo zip entregable

6.1. Fecha de entrega

La entrega se hará a través del campus. La fecha de entrega está pautada para el día martes 25 de Abril de 2023 a las 19.00hs. **Cualquier entrega fuera de término no será considerada.**

6.2. Informe

La entrega debe contar con un informe donde se demuestre conocimiento de la interfaz de sockets, así como también los resultados de las ejecuciones de prueba (capturas de ejecución de cliente y logs del servidor). El informe debe describir la arquitectura de la aplicación. En particular, se pide detallar el protocolo de red implementado para cada una de las operaciones requeridas.

El informe debe tener la siguientes secciones:

- Introducción
- Hipótesis y suposiciones realizadas
- Implementación
- Pruebas
- Preguntas a responder
- Dificultades encontradas
- Conclusión

Referencias

[1] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

[2] Python Software Foundation. *socket — Low-level networking interface*, 2020. `https://docs.python.org/3/library/socket.html` [Accessed: 15/10/2020].

[3] John Gruber. *Markdown*, 2020. `https://daringfireball.net/projects/markdown/` [Accessed: 15/10/2020].

[4] Guido van Rossum, Barry Warsaw, and Nick Coghlan. Style guide for Python code. PEP 8, Python Software Foundation, 2001.

[5] Ian Stapleton Cordasco. *Flake8: Your Tool For Style Guide Enforcement*, 2020. `https://flake8.pycqa.org/en/latest/` [Accessed: 15/10/2020].