Presentación File Transfer over UDP

Introducción a los Sistemas Distribuidos
(1c-2025)

Integrantes

- Jesabel Pugliese
- Josué Martel
- Leticia Figueroa
- Andrea Figueroa
- Kevin Vallejo

Contenidos

Capa de aplicación

- *Arquitectura de software.*
- *Protocolo de aplicación.*

Arquitectura de Software

Arquitectura de software

Server: Espera ante peticiones de clientes

<<pseudocódigo>>

```
class Server:
    def accept_clients(self):
        while True:
            self._reap_dead()
            client_data, client_addr = self.skt_listener.recvfrom(
                TAM_BUFFER)
            if client_addr not in self.clients:
                client_thread = threading.Thread(
                    target=self._handle_client, args=(
                        client_data, client_addr))
                client_thread.start()
                self.clients[client_addr] = client_thread
```

Liberación de recursos

Reserva de recursos para atender al cliente

Protocolo de aplicación

Protocolo de aplicación

<<Sender>>

```python
def _send_file(self, rdt, file):
    while True:
        data = file.read(FILE_CHUNK_SIZE)
        if not data:
            break
        rdt.send(data)

    # mensaje de fin de transferencia
    rdt.send(bytes())
```

Lee chunks del archivo y los manda usando el RDT.

Protocolo de aplicación

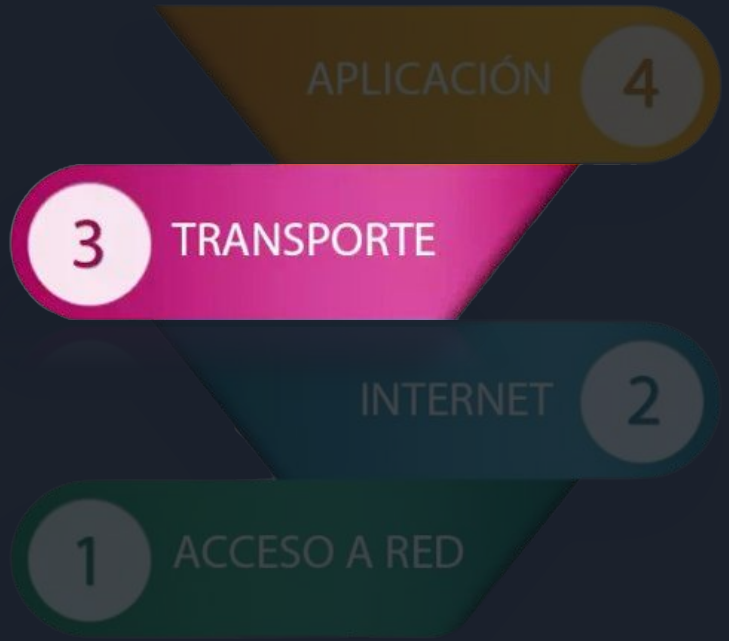
<<Receiver>>

```python
def _recv_file(self, rdt, file):
    while True:
        data = rdt.receive()
        if not data:
            # recibe mensaje de fin de transferencia
            break
        file.write(data)
```

Recibe data a través del RDT y los escribe en archivo

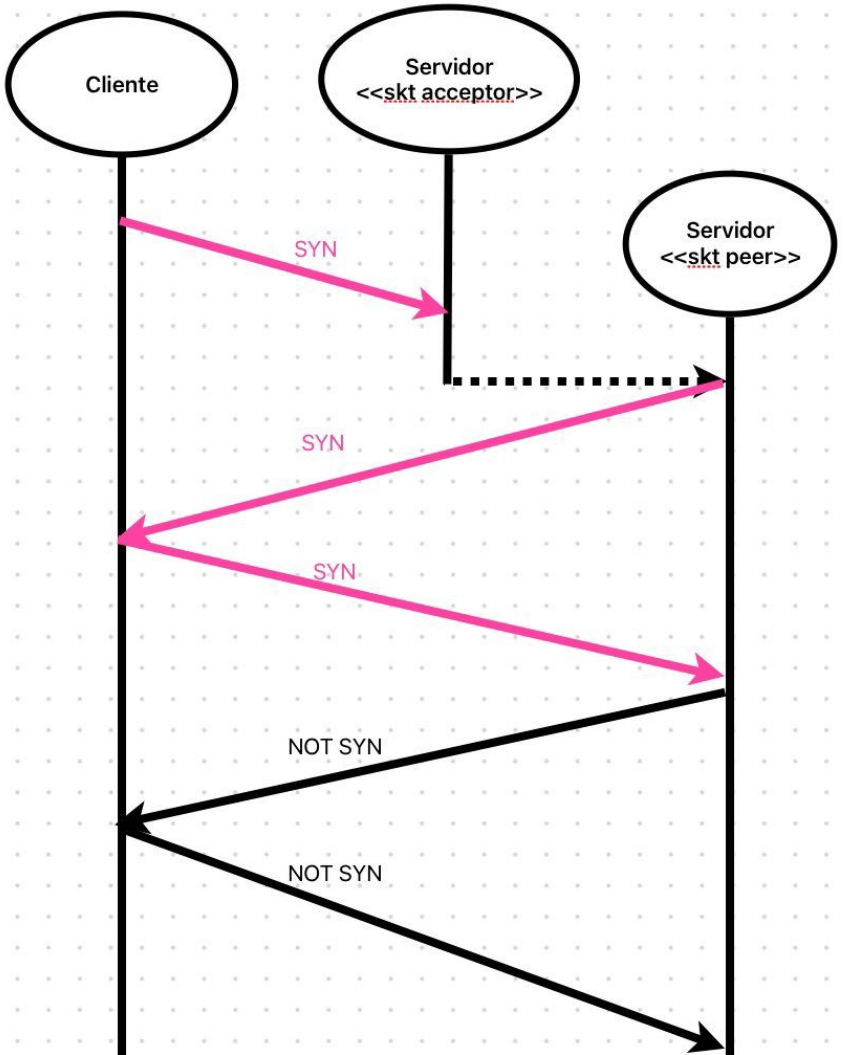
Capa de Transporte

- *Handshake.*
- *Protocolos recuperación de errores.*

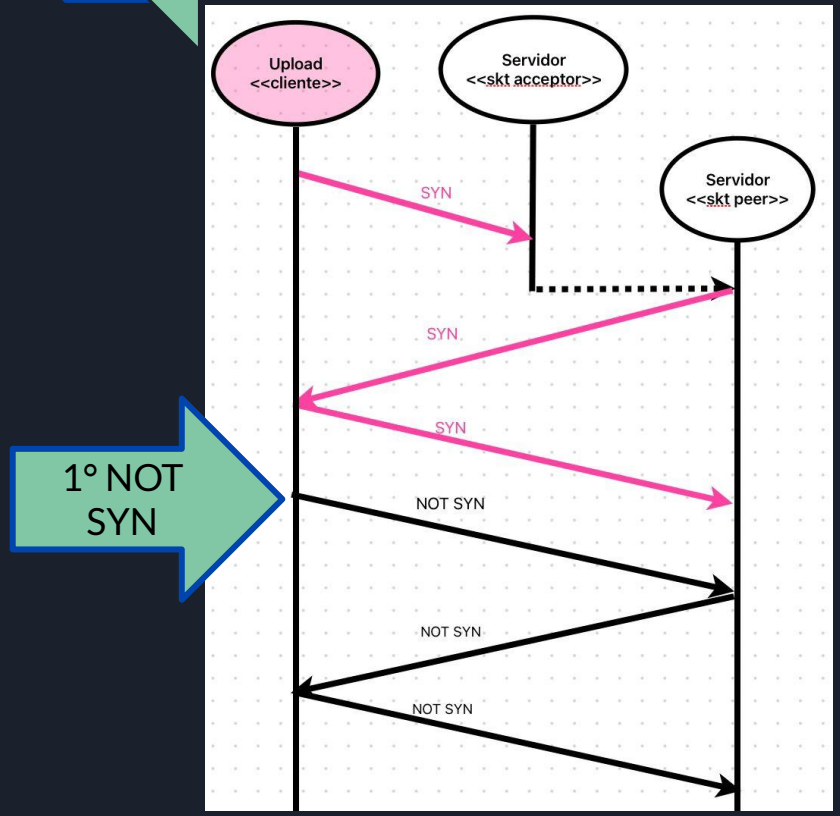


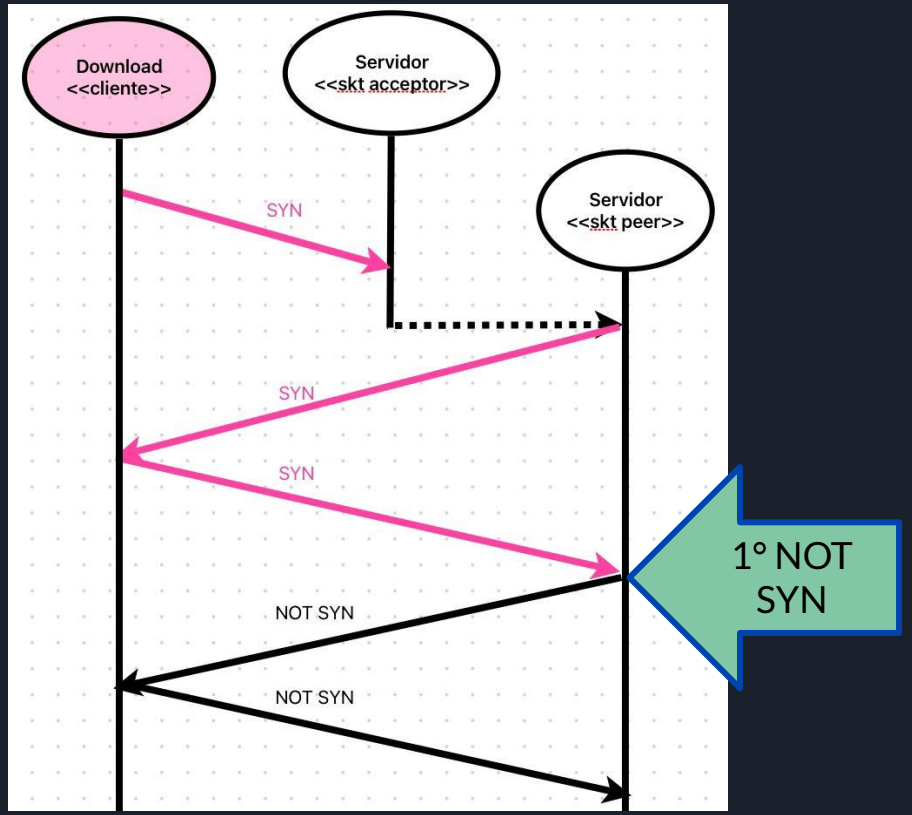
Handshake

<<Motivación>>



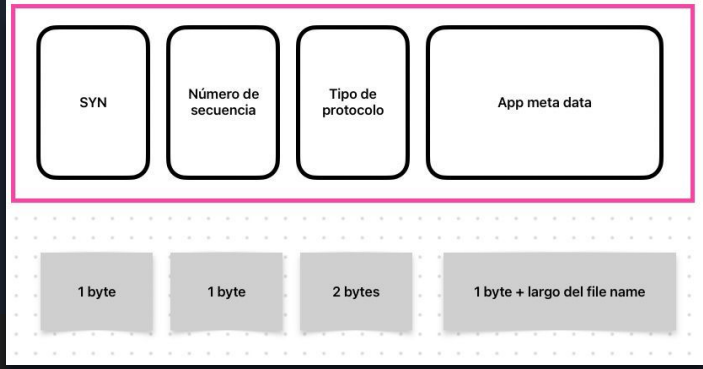
Distintos flujos de comunicación según el cliente





Primer mensaje SYN: Petición del cliente

```python
def _send_msg_1(self, skt, client_prot_type, app_metadata) -> int:
    packet = MessageSerializer.first_msg_to_bytes(
        self.num_seq, client_prot_type, app_metadata)

    for _ in range(NUM_ATTEMPS_HANDSHAKE):
        try:
            skt.sendto(packet, self.srv_addr)
            data_handshake, self.srv_addr = skt.recvfrom(TAM_BUFFER)
        except socket.timeout:
            reconsider_timeout(skt)
            continue

        srv_num_seq, ack = MessageSerializer.second_msg_from_bytes(
            data_handshake)
        return srv_num_seq

    raise ConnectionError(
        "tried to reach the server several times without response")
```



<<Cliente envía primer mensaje>>

Intenta solicitar el servicio hasta que recibe el 2do mensaje SYN

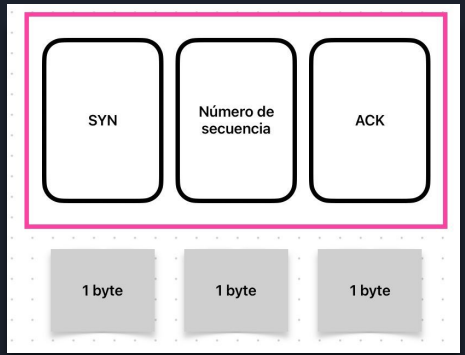
Se rinde porque excede el límite de intentos

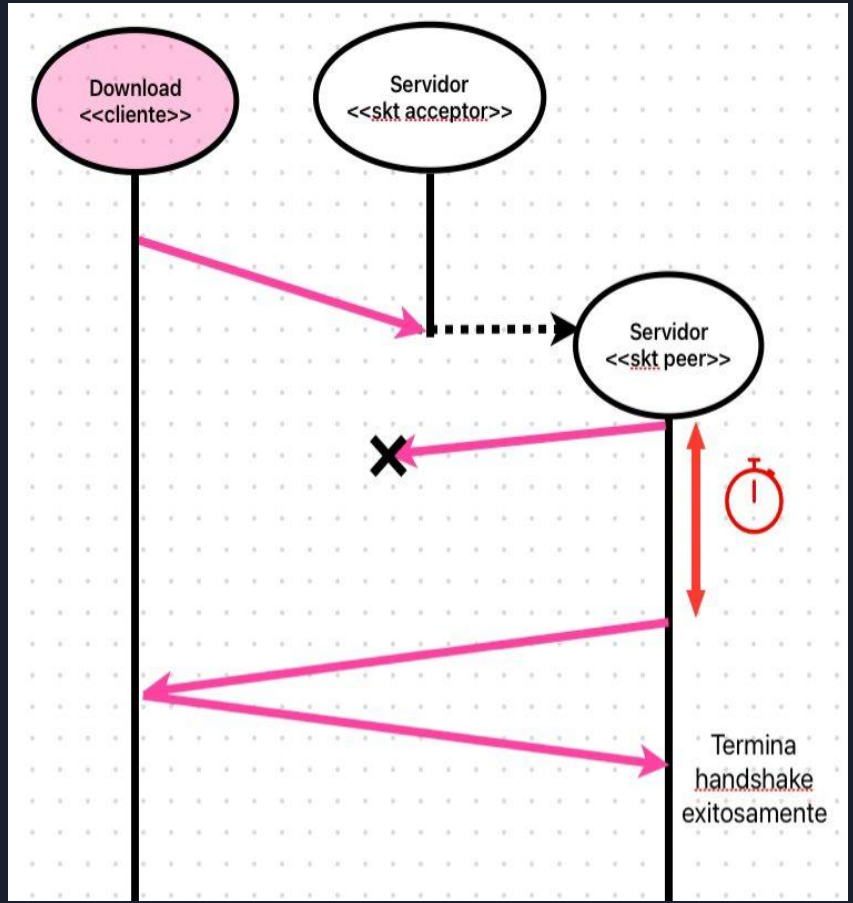
Segundo mensaje SYN:

- Envía el segundo mensaje hasta obtener una respuesta del cliente.

- después de ciertos intentos, se rinde y asume que la conexión no se pudo establecer

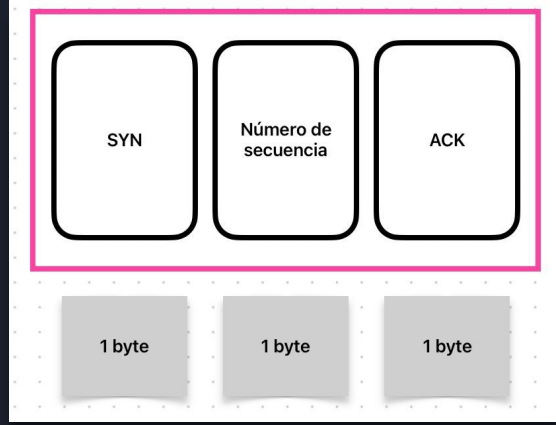
Resulta idéntico para un cliente upload





Segundo mensaje SYN: código

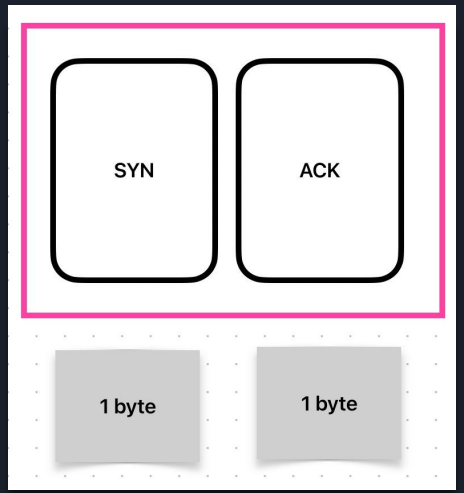
```python
def _send_msg_2(self, skt_peer, client_num_seq) -> int:
    packet = MessageSerializer.second_msg_to_bytes(
        self.num_seq, client_num_seq)
    for _ in range(NUM_ATTEMPS_HANDHAKE):
        try:
            skt_peer.sendto(packet, self.client_addr)
            client_data, self.client_addr = skt_peer.recvfrom(TAM_BUFFER)
        except socket.timeout:
            reconsider_timeout(skt_peer)
            continue

        # Si el mensaje está corrupto, espera de nuevo por el ACK
        _, ack = MessageSerializer.third_msg_from_bytes(client_data)
        if self.num_seq == ack:
            return

    raise ConnectionError(
        "tried to reach the client several times without response")
```



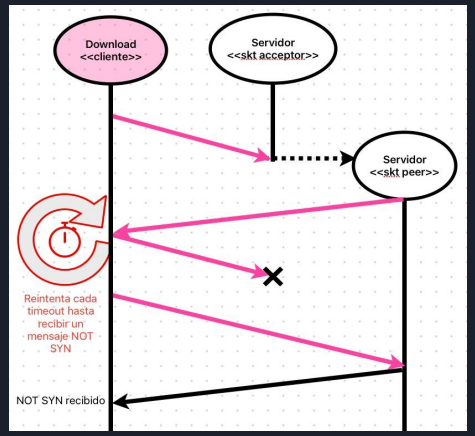
Intenta contactar al cliente para otorgar el servicio hasta recibir una respuesta (cualquiera)

Se rinde si excede el límite de intentos

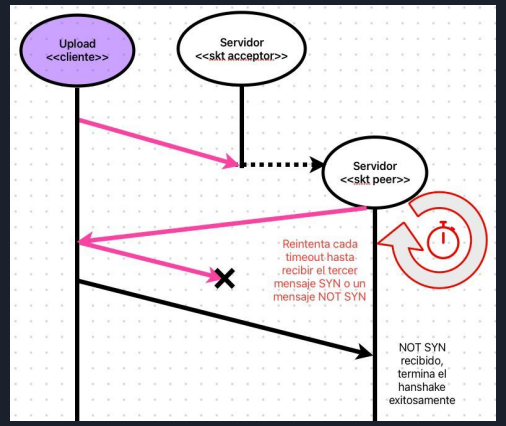
Tercer mensaje SYN



Mensaje



DOWNLOAD



UPLOAD

```python
def handshake(self, client_type, client_prot_type, srv_file_name, skt):
    app_metadata = client_type + srv_file_name
    srv_num_seq = self._send_msg_1(skt, client_prot_type, app_metadata)

    if client_type == UPLOAD:
        self._send_innocent_mgs_3(skt, srv_num_seq)
    else:
        self._send_msg_3(skt, srv_num_seq)
    return self.srv_addr
```

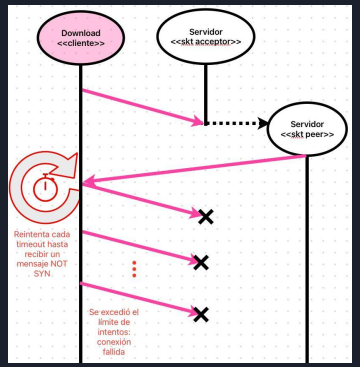
Flujo de comunicación según tipo de cliente

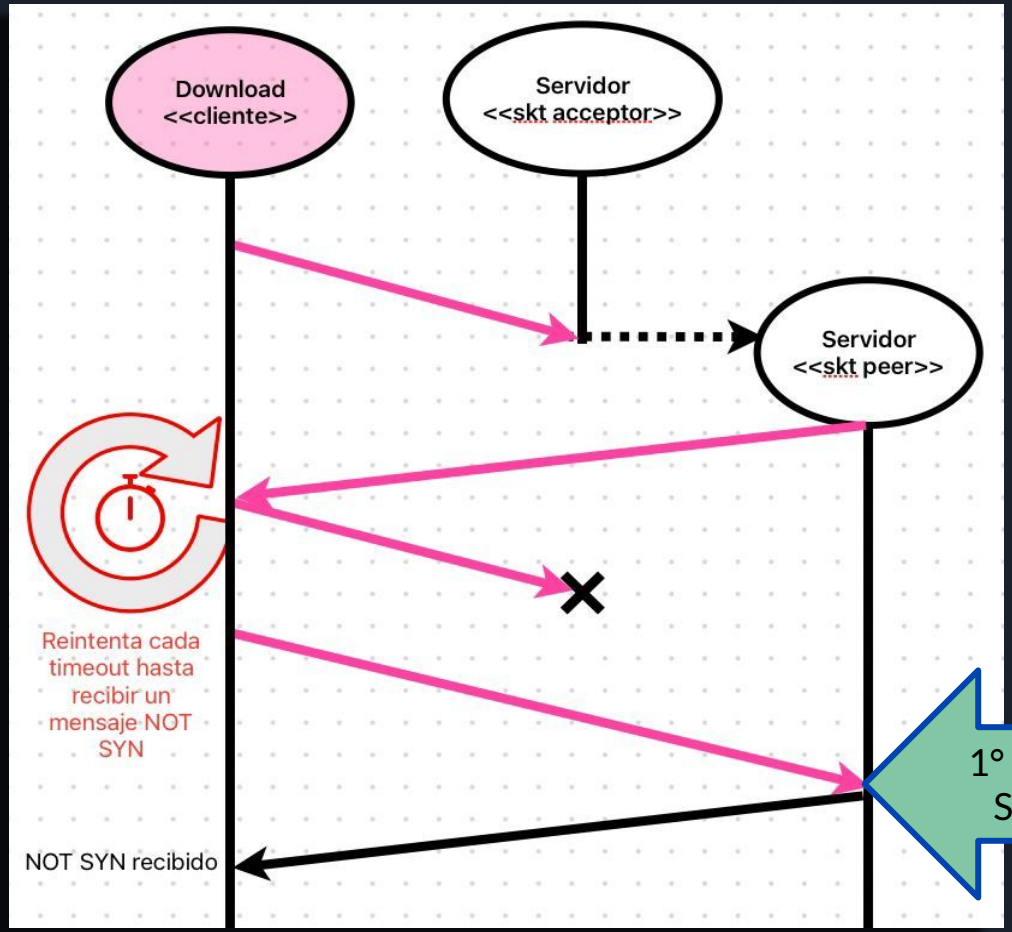
Tercer mensaje SYN

Caso feliz: Download

- *El servidor sigue enviando el segundo SYN hasta recibir el tercer SYN*

- *El cliente reenvía el tercer SYN hasta confirmar que le llegó al servidor (esto es que le llegue el primer NOT SYN)*





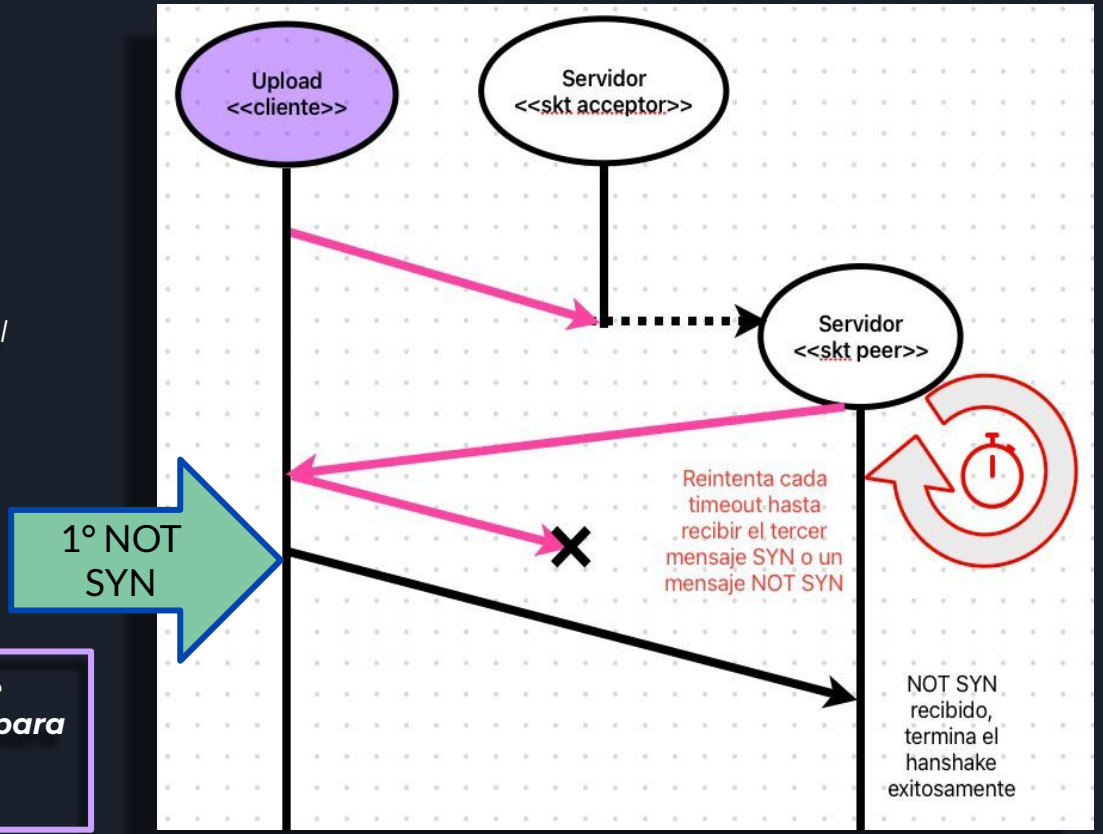
Tercer mensaje SYN

Upload

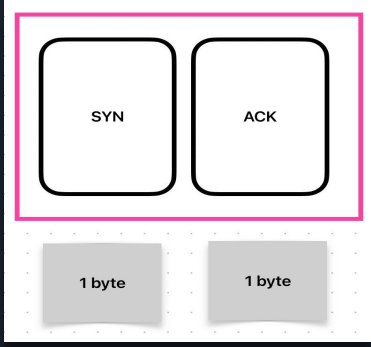
- *El servidor sigue enviando el segundo SYN hasta recibir el tercer SYN o en su defecto el mensaje el primer mensaje NOT SYN*

- *El cliente envía el tercer SYN y procede a enviar el primer mensaje NOT.*

Si incluso falla en enviarse el tercer mensaje SYN, el RDT se encargará de hacer lo mejor para enviar el primer mensaje SYN



Tercer mensaje SYN: código




```python
def _send_innocent_mgs_3(self, skt, srv_num_seq):
    packet = MessageSerializer.third_msg_to_bytes(srv_num_seq)
    skt.sendto(packet, self.srv_addr)
```

Para cliente UPLOAD

```python
def _send_msg_3(self, skt, srv_num_seq):
    packet = MessageSerializer.third_msg_to_bytes(srv_num_seq)
    for _ in range(NUM_ATTEMPS_HANDSHAKE):
        try:
            skt.sendto(packet, self.srv_addr)
            data, self.srv_addr = skt.recvfrom(TAM_BUFFER)

            if not MessageSerializer._is_about_handhshake(data):
                return
        except socket.timeout:
            reconsider_timeout(skt)
            continue

    raise ConnectionError(
        "tried to reach the server several times without response")
```

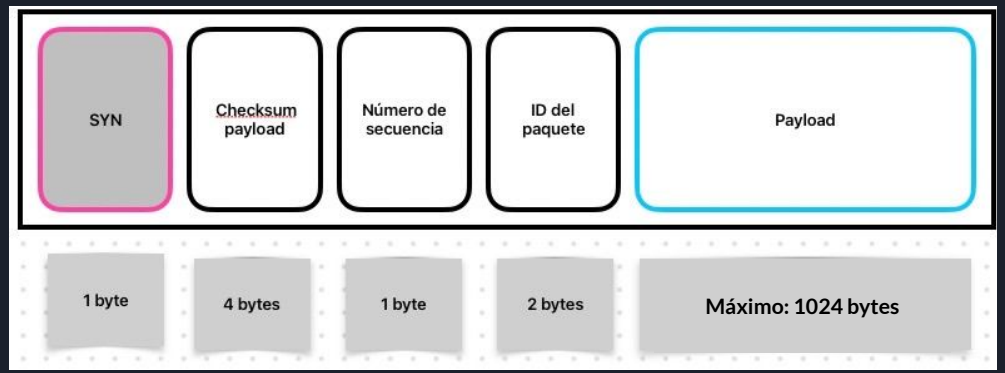
Para cliente DOWNLOAD

Protocolo de recuperación de errores

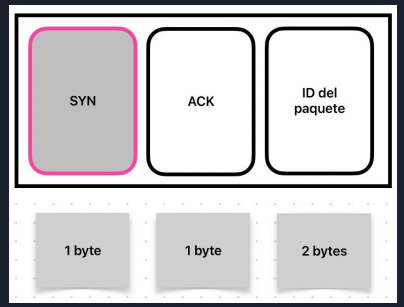
Selective Repeat

Estructura de contenido

Segmento



ACK



Selective Repeat

Implementación - Sender

```
data_segments = self._split_data(data)
data_segments.append(bytes())
ack_buffer = set()
timers = {}
current_sn = 0
```

```
win_end = min(current_sn + self.win_size, len(data_segments))
for sn in range(current_sn, win_end):
    if sn in timers or sn in ack_buffer:
        continue
    self._send_segment_sr(sn, self.pkt_id, data_segments[sn])
    timers[sn] = time.time()
```

Selective Repeat

Implementación - Sender

```
ack, ack_pkt_id = self._recv_ack_sr()
if ack_pkt_id < self.pkt_id:
    continue
```

```
if ack > current_sn:
    ack_buffer.add(ack)
    del timers[ack]
```

```
if ack == current_sn:
    current_sn = self._update_current_sn(ack, ack_buffer)
    del timers[ack]
```

Selective Repeat

Implementación - Sender

```
def _update_current_sn(self, ack_received, ack_buffer):
    current_sn = ack_received + 1
    while current_sn in ack_buffer:
        ack_buffer.remove(current_sn)
        current_sn += 1
    return current_sn
```

```
for sn in list(timers):
    if timers[sn] + TIMEOUT <= time.time():
        self._send_segment_sr(sn, self.pkt_id, data_segments[sn])
        timers[sn] = time.time()
```

Selective Repeat

Implementación - Receiver

```
data = bytearray()
segments_buffer = {}
end_segment_buffered = -1
expected_sn = 0
```

Número de secuencia del segmento de fin de paquete

```
seq_num, pkt_id, payload = self._recv_segment_sr()
if self.pkt_id != pkt_id:
    self._send_ack_sr(seq_num, pkt_id)
    continue
```

```
if seq_num < expected_sn:
    self._send_ack_sr(seq_num, pkt_id)
```

Selective Repeat

Implementación - Receiver

```
if seq_num > expected_sn:
    segments_buffer[seq_num] = payload
    if len(payload) == 0:
        end_segment_buffered = seq_num
    self._send_ack_sr(seq_num, pkt_id)
```

```
if seq_num == expected_sn:
    data.extend(payload)
    self._send_ack_sr(seq_num, pkt_id)
    if len(payload) == 0:
        self.pkt_id += 1
        return data
```

Selective Repeat

Implementación - Receiver

```
expected_sn = seq_num + 1
while expected_sn in segments_buffer:
    data.extend(segments_buffer.pop(expected_sn))
    if expected_sn == end_segment_buffered:
        self.pkt_id += 1
        return data
    expected_sn += 1
```

Stop-And-Wait

Dada la implementación anterior, implementamos Stop-and-Wait como un caso particular de Selective Repeat, con un tamaño de ventana 1.

Comportamiento de Stop-And-Wait:

- Se envía un solo segmento.
- El emisor espera su ACK antes de enviar el siguiente segmento.
- No se permite el envío de múltiples segmentos sin confirmación.

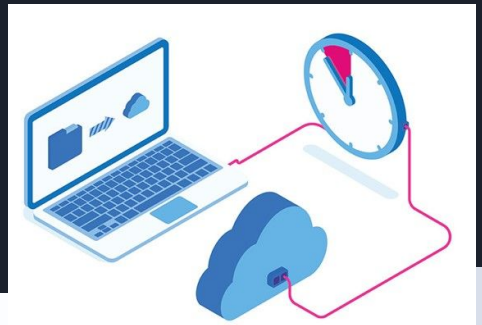
Selective Repeat con ventana de tamaño 1:

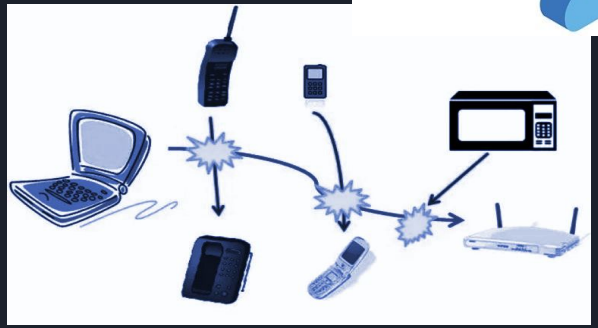
- Permite enviar sólo un segmento a la vez.
- Si llega su ACK, se avanza la ventana y se puede enviar el siguiente.
- Si no llega el ACK a tiempo, se retransmite ese único segmento

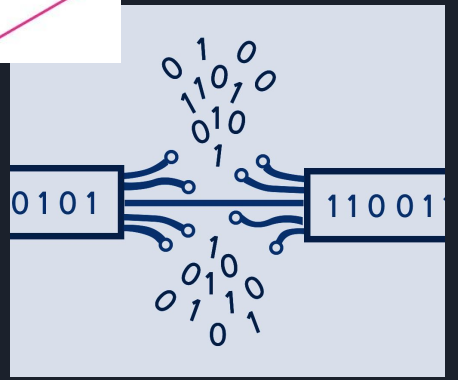
Simulación de topologías: Objetivos

- ¿Por qué sería necesario simular topologías de red?

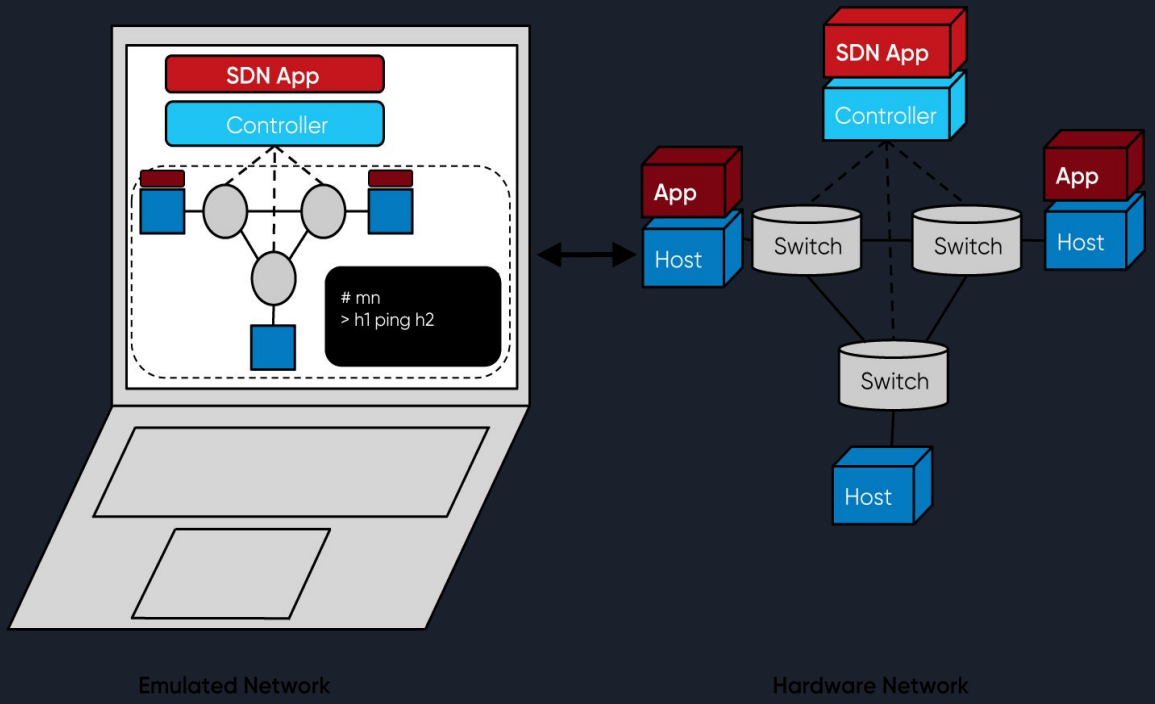
- Latencia
- jitter
- Reordenamiento
- Corrupción
- Pérdida







Simulación de topologías: *Mininet*



Software que nos permite diseñar y virtualizar diferentes topologías de red.

Nos permite reproducir escenarios determinados.

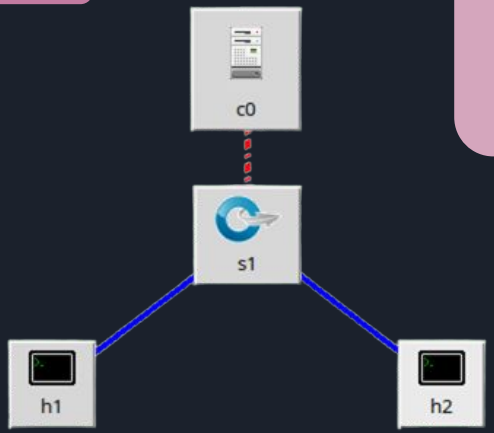
Tiene integración nativa con Wireshark

Plus: interfaz gráfica.

Simulación de topologías: Ensayos

Testeo en topologías con características factibles en escenarios reales.

single client



Nos permitirá directamente pedir efectos en el canal de comunicación que podrían surgir en topologías reales. Sin otros factores que alteren los valores pedidos

Determinamos el aproximado de delay total sobre el path, porcentaje de pérdida sobre el camino), menor ancho de banda en el camino, etc.

Simulación de topologías: Ensayos

Se testeó el comportamiento de la red en topologías con características factibles en escenarios reales.

single client

Variación de los detalles en ls links para someter al protocolo a redes crecientemente hostiles

Topología	Bandwidth	Delay	Jitter	Loss
A	100	1ms	0ms	1%
B	30	10ms	3ms	3%
C	10	20ms	10ms	5%

Simulación de topologías: Ensayos

single client

tamaño de archivo	tiempo con SR	tiempo con SW	dif enviado-recibido SR	dif enviado-recibido SW
1 KB	0.010	0.14	0	0
10 KB	0.027	0.63	0	0
100 KB	0.526	0.584	0	0
1 MB	2.952	5.743	0	0

Comportamiento observado en Topología A

SR= 0.5 (SW)

tamaño de archivo	tiempo con SR	tiempo con SW	dif enviado-recibido SR	dif enviado-recibido SW
1 KB	0.723	0.421	0	0
10 KB	1.026	1.200	0	0
100 KB	8.256	11.697	0	0
1 MB	80.875	95.242	0	0

Comportamiento observado en Topología B

SR= 0.7(SW)

Simulación de topologías: Ensayos

single client

tamaño de archivo	tiempo con SR	tiempo con SW	dif enviado-recibido SR	dif enviado-recibido SW
1 KB	0.628	0.619	0	0
10 KB	0.902	1.056	0	0
100 KB	9.066	16.708	0	0
1 MB	94.176	115.871	0	0

Comportamiento observado en Topología C

Sr= 0.8(SW)

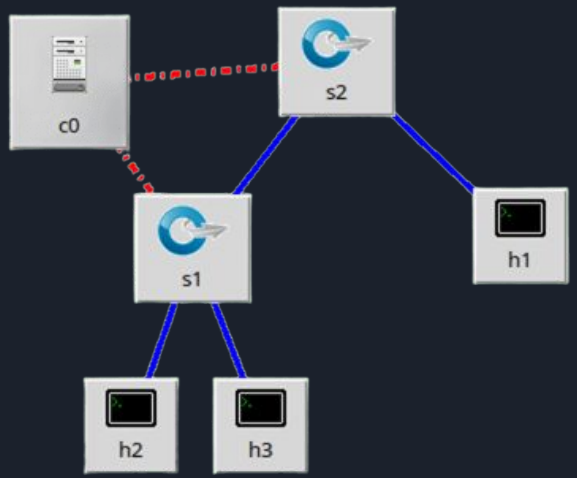
Observamos:

- La diferencia de performance (delay total) entre Selective Repeat y Stop-and-Wait tiende a ser mayor mientras mejor se comporta la red.

Simulación de topologías: Ensayos

Testeo en topologías con características factibles en escenarios reales.

multiple clients



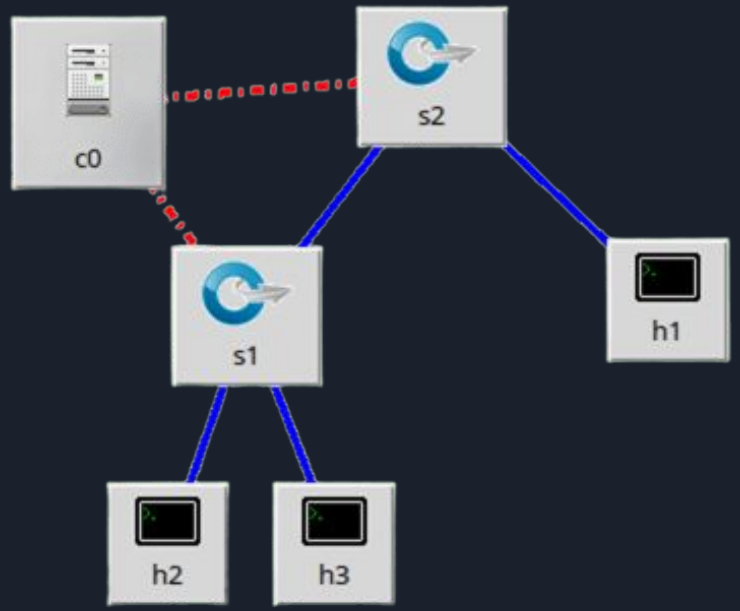
Nos permite evaluar el comportamiento del servidor atendiendo a más de un cliente.

De cierta forma sometemos la conexión a un entorno con cierto nivel de incertidumbre incorporada por el tráfico externo

Simulación de topologías: Ensayos

multiple clients

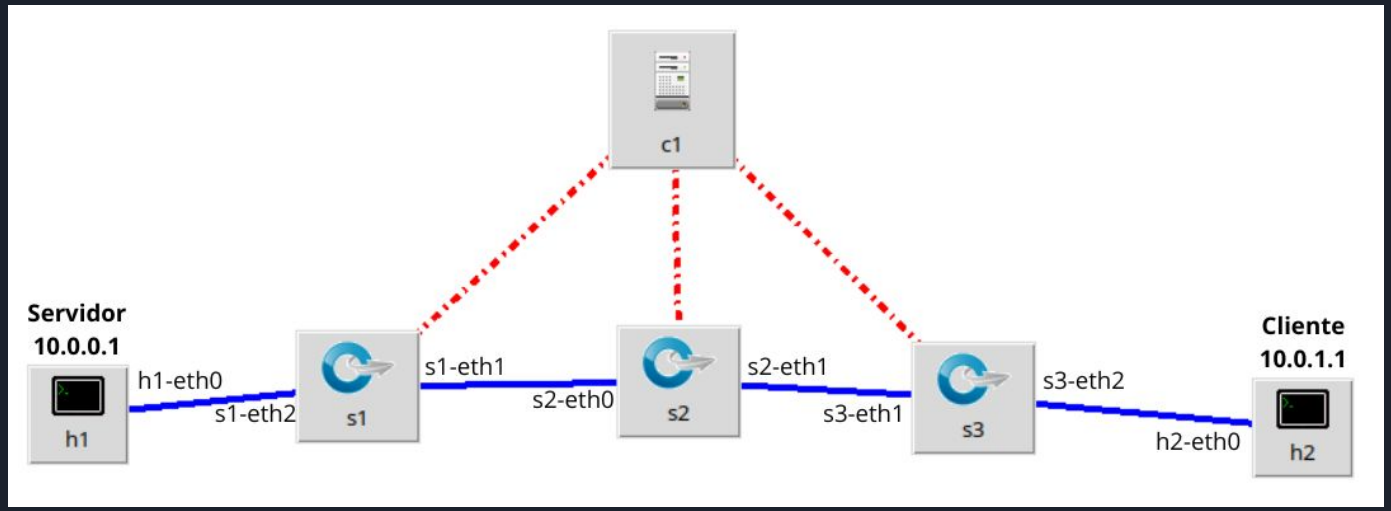
https://youtu.be/nVHvyXWkkuE
Comprobamos correcto funcionamiento con múltiples clientes realizando transferencias al mismo tiempo (h2 y h3)



Demo en vivo!

Anexo: Fragmentación en IPv4

Armado de la topología



Topología de red

- Variación del MTU en `s2-eth0`
- Pérdida de paquetes del 10% en `s3-eth2`

Anexo: Fragmentación en IPv4

Generación de tráfico con *iperf* de paquetes de 1400 bytes

Tráfico sin fragmentación:
Elección de un mínimo MTU de la red MAYOR al tamaño en bytes de los paquetes enviados:

MTU = 1500

Anexo: Fragmentación en IPv4

Generación de tráfico con *iperf* de paquetes de 1400 bytes

UDP:

Tamaño paquete IP = 1470B (payload) + 8B (header UDP) + 20B (header IP) = 1498B
Tamaño total = Tamaño paquete IP + 14B (Ethernet) = 1512B

No.	Time	Source	Destination	Protocol	Length	Info
882	16.022590828	10.0.1.1	10.0.0.1	UDP	1512	34880 → 5001 Len=1470
883	16.033544126	10.0.1.1	10.0.0.1	UDP	1512	34880 → 5001 Len=1470
884	16.044991994	10.0.1.1	10.0.0.1	UDP	1512	34880 → 5001 Len=1470

- UDP envía cada paquete tal cual lo recibe en la aplicación

Anexo: Fragmentación en IPv4

Generación de tráfico con *iperf* de paquetes de 1400 bytes

TCP:

Tamaño paquete IP = 1448B (payload) + 32B (header TCP) + 20 (header IP) = 1500B
Tamaño total = Tamaño paquete IP + 14 (Ethernet) = 1514B

No.	Time	Source	Destination	Protocol	Length	Info
7	0.001844769	10.0.1.1	10.0.0.1	TCP	1514	60972 → 5001 [ACK] Seq=2957 Ack=1 Win=42496 Len=1448
8	0.001865779	10.0.1.1	10.0.0.1	TCP	1514	60972 → 5001 [ACK] Seq=4405 Ack=1 Win=42496 Len=1448
9	0.001886728	10.0.1.1	10.0.0.1	TCP	1514	60972 → 5001 [ACK] Seq=5853 Ack=1 Win=42496 Len=1448

- TCP acumula datos y envía segmentos según MSS y control de congestión (payload de capa de aplicación puede aumentar)

Anexo: Fragmentación en IPv4

Generación de tráfico con *iperf* de paquetes de 1400 bytes

Tráfico con fragmentación:
Elección de un mínimo MTU de la red MENOR al tamaño en bytes de los paquetes enviados:

$MTU_1 = 500$
$MTU_2 = 100$

Anexo: Fragmentación en IPv4

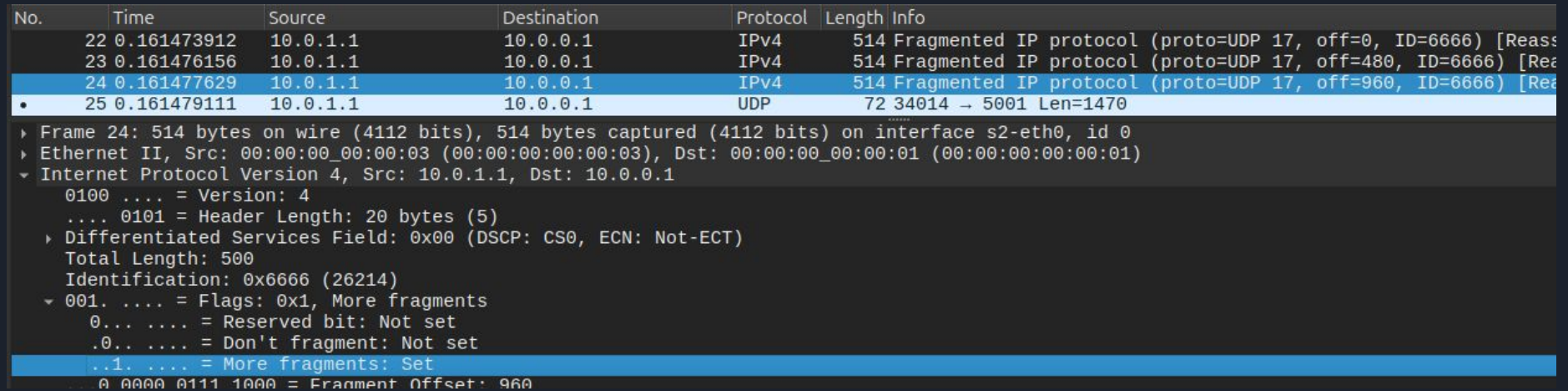
Generación de tráfico con *iperf* de paquetes de 1400 bytes

UDP (MTU_1 = 500):

Tamaño paquete UDP = 1470B (payload) + 8B (header UDP) = 1478B > 500 → **Fragmentación**

Tamaño de cada fragmento IP = 480B (payload) + 20B (header IP) = 500B

Tamaño total de último fragmento = 38B (payload restante) + 8B (header UDP) + 20B (header IP) + 14B (Ethernet) = 72B



Anexo: Fragmentación en IPv4

Generación de tráfico con *iperf* de paquetes de 1400 bytes

UDP (MTU_2 = 100):

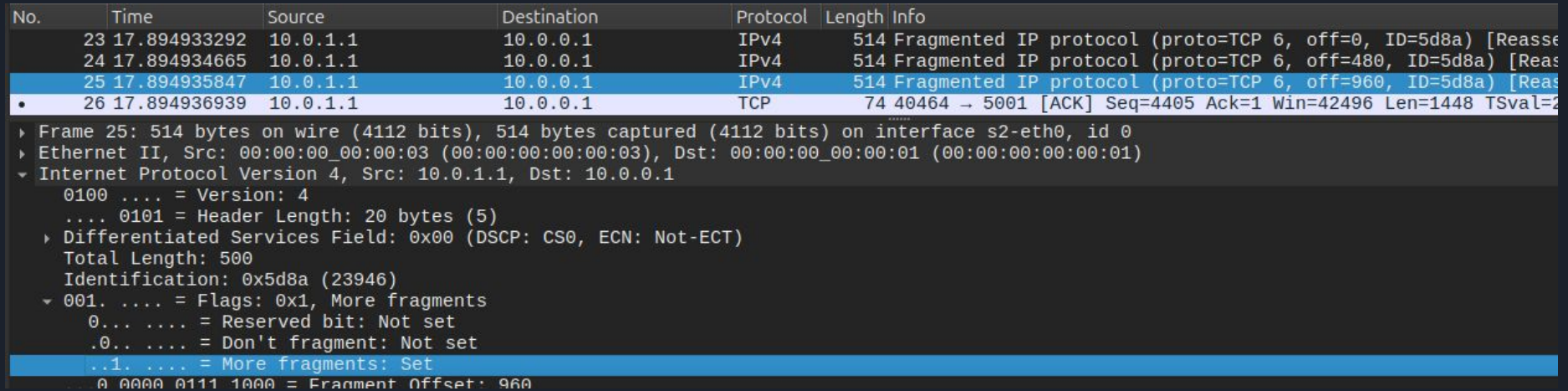
No.	Time	Source	Destination	Protocol	Length	Info
20	0.010891047	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=UDP 17, off=0, ID=5767) [F
21	0.010894554	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=UDP 17, off=80, ID=5767) [
22	0.010896247	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=UDP 17, off=160, ID=5767)
23	0.010897720	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=UDP 17, off=240, ID=5767)
24	0.010899353	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=UDP 17, off=320, ID=5767)
25	0.010900956	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=UDP 17, off=400, ID=5767)
26	0.010902509	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=UDP 17, off=480, ID=5767)
27	0.010903982	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=UDP 17, off=560, ID=5767)
28	0.010905304	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=UDP 17, off=640, ID=5767)
29	0.010906607	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=UDP 17, off=720, ID=5767)
30	0.010908190	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=UDP 17, off=800, ID=5767)
31	0.010909653	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=UDP 17, off=880, ID=5767)
32	0.010911125	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=UDP 17, off=960, ID=5767)
33	0.010912448	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=UDP 17, off=1040, ID=5767)
34	0.010913750	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=UDP 17, off=1120, ID=5767)
35	0.010915233	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=UDP 17, off=1200, ID=5767)
36	0.010916686	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=UDP 17, off=1280, ID=5767)
37	0.010918128	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=UDP 17, off=1360, ID=5767)
38	0.010919471	10.0.1.1	10.0.0.1	UDP	72	56007 → 5001 Len=1470

- Aumento de la cantidad de paquetes a medida que aumenta la fragmentación

Anexo: Fragmentación en IPv4

Generación de tráfico con *iperf* de paquetes de 1400 bytes

TCP (MTU_1 = 500):



Anexo: Fragmentación en IPv4

Generación de tráfico con *iperf* de paquetes de 1400 bytes

TCP (MTU_2 = 100):

No.	Time	Source	Destination	Protocol	Length	Info
105	12.919710749	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=TCP 6, off=0, ID=672c) [Re
106	12.919712021	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=TCP 6, off=80, ID=672c) [F
107	12.919713123	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=TCP 6, off=160, ID=672c) [
108	12.919714185	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=TCP 6, off=240, ID=672c) [
109	12.919715267	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=TCP 6, off=320, ID=672c) [
110	12.919716309	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=TCP 6, off=400, ID=672c) [
111	12.919717341	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=TCP 6, off=480, ID=672c) [
112	12.919718393	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=TCP 6, off=560, ID=672c) [
113	12.919719415	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=TCP 6, off=640, ID=672c) [
114	12.919720447	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=TCP 6, off=720, ID=672c) [
115	12.919721489	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=TCP 6, off=800, ID=672c) [
116	12.919722531	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=TCP 6, off=880, ID=672c) [
117	12.919723563	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=TCP 6, off=960, ID=672c) [
118	12.919724595	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=TCP 6, off=1040, ID=672c)
119	12.919725647	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=TCP 6, off=1120, ID=672c)
120	12.919726759	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=TCP 6, off=1200, ID=672c)
121	12.919727831	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=TCP 6, off=1280, ID=672c)
122	12.919728893	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=TCP 6, off=1360, ID=672c)
123	12.919730125	10.0.1.1	10.0.0.1	TCP	74	34294 → 5001 [ACK] Seq=7301 Ack=1 Win=42496 Len=1448 TSv

Anexo: Fragmentación en IPv4

Generación de tráfico con *iperf* de paquetes de 1400 bytes

Tráfico con pérdida de paquetes

TCP:

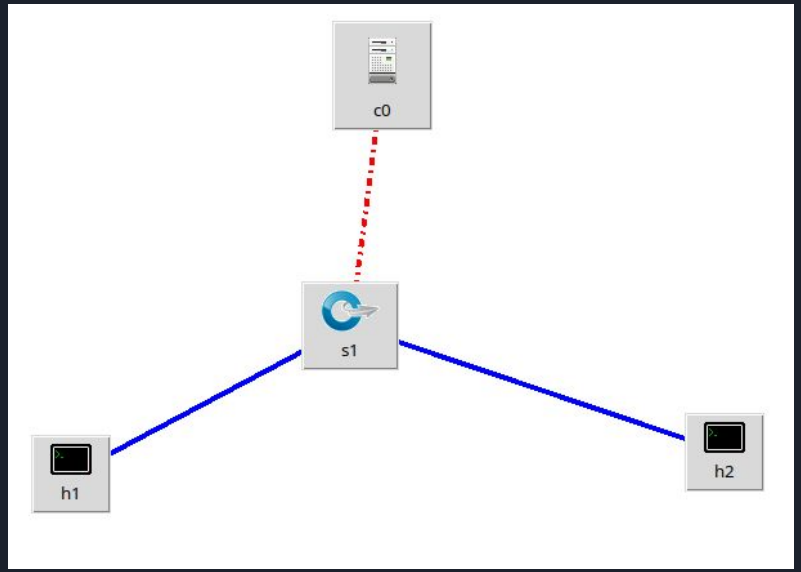
No.	Time	Source	Destination	Protocol	Lengt	Info
8	0.001819273	10.0.0.1	10.0.1.1	TCP	94	5001 → 45816 [PSH, ACK] Seq=1 Ack=61 Win=43520 Len=
646	0.206172933	10.0.0.1	10.0.1.1	TCP	94	[TCP Retransmission] 5001 → 45816 [PSH, ACK] Seq=1

- Retransmisión del paquete (de ACK en este caso)

Comparación de los Protocolos

El siguiente estudio se realiza con las siguientes características:

- h1: Server siendo receiver
- h2: Cliente siendo sender
- En una topología simple hecha con mininet.
- TIMEOUT: 0.001



Archivos y pérdida de paquetes

Porcentaje de pérdida de fragmentos

- 0%
- 10%
- 20%

Archivos

- 2.6MB
- 5.2MB
- 20.2MB

2_6MB.jpg

5_2MB.jpg

20_2MB.mp4

Capturas en Wireshark

Stop and Wait

No.	Time	Source	Destination	Protocol	Length	Info
64	18.910…	10.0.0.2	10.0.0.1	UDP	10…	44048 → 39435 Len=1032
65	18.910…	10.0.0.1	10.0.0.2	UDP	46	39435 → 44048 Len=4
66	18.910…	10.0.0.2	10.0.0.1	UDP	10…	44048 → 39435 Len=1032
67	18.913…	10.0.0.2	10.0.0.1	UDP	10…	44048 → 39435 Len=1032
68	18.913…	10.0.0.1	10.0.0.2	UDP	46	39435 → 44048 Len=4
69	18.913…	10.0.0.2	10.0.0.1	UDP	10…	44048 → 39435 Len=1032
70	18.913…	10.0.0.1	10.0.0.2	UDP	46	39435 → 44048 Len=4
71	18.913…	10.0.0.2	10.0.0.1	UDP	50	44048 → 39435 Len=8
72	18.913…	10.0.0.1	10.0.0.2	UDP	46	39435 → 44048 Len=4

Selective Repeat

No.	Time	Source	Destination	Protocol	Length	Info
…	0.00…	10.0.0.1	10.0.0.2	UDP	46	48558 → 60278 Len=4
…	0.00…	10.0.0.2	10.0.0.1	UDP	1…	60278 → 48558 Len=1032
…	0.00…	10.0.0.2	10.0.0.1	UDP	1…	60278 → 48558 Len=1032
…	0.00…	10.0.0.1	10.0.0.2	UDP	46	48558 → 60278 Len=4
…	0.00…	10.0.0.1	10.0.0.2	UDP	46	48558 → 60278 Len=4
…	0.00…	10.0.0.2	10.0.0.1	UDP	1…	60278 → 48558 Len=1032
…	0.00…	10.0.0.2	10.0.0.1	UDP	1…	60278 → 48558 Len=1032
…	0.00…	10.0.0.2	10.0.0.1	UDP	1…	60278 → 48558 Len=1032
…	0.00…	10.0.0.2	10.0.0.1	UDP	1…	60278 → 48558 Len=1032
…	0.00…	10.0.0.1	10.0.0.2	UDP	46	48558 → 60278 Len=4
…	0.00…	10.0.0.1	10.0.0.2	UDP	46	48558 → 60278 Len=4
…	0.00…	10.0.0.2	10.0.0.1	UDP	50	60278 → 48558 Len=8
…	0.00…	10.0.0.1	10.0.0.2	UDP	46	48558 → 60278 Len=4

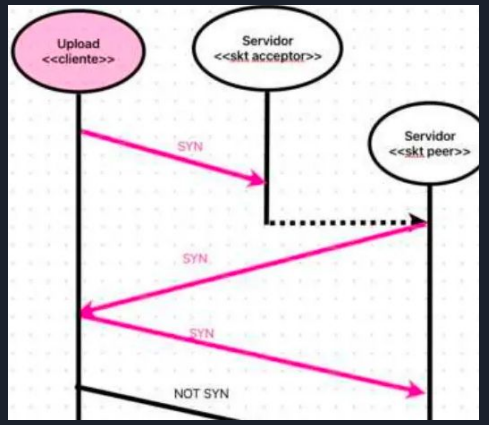
Ejemplo completo

Sender
1032= 8 Header + 1024 Segment
Header de 8 Bytes:

- SYN: 1b
- Checksum: 4b
- Numero de secuencia: 1b
- ID de paquete: 2b

Receiver
4 bytes

- SYN: 1b
- Numero de secuencia: 1b
- ID de paquete: 2b



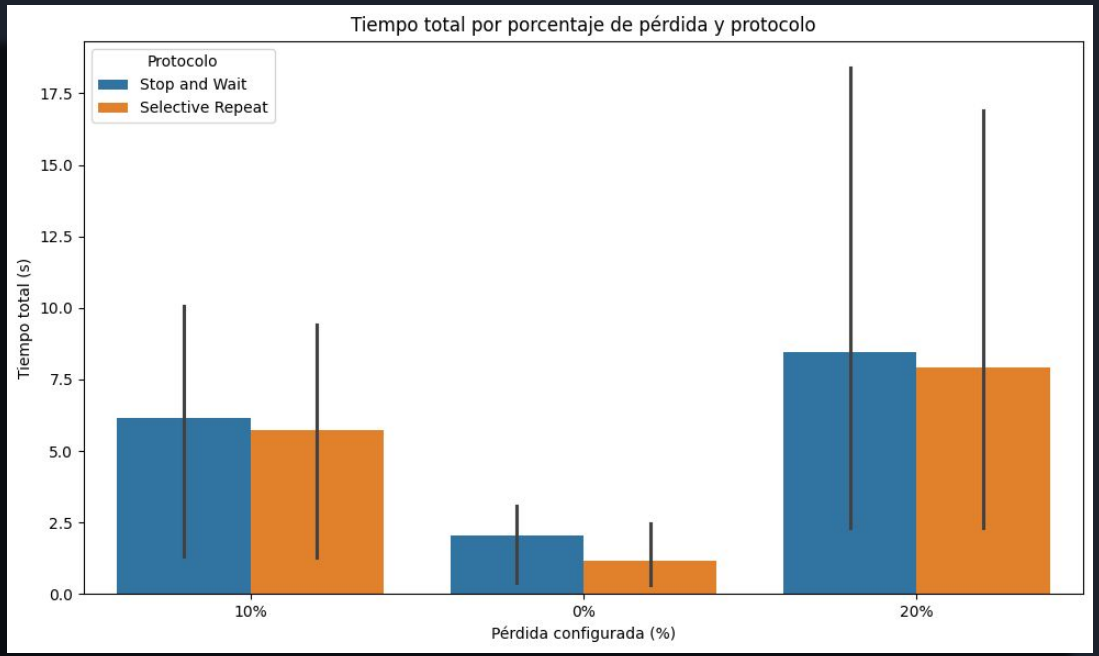
No.	Time	Source	Destination	Protocol	Length	Info
3	0.0012…	10.0.0.2	10.0.0.1	UDP	70	53302 → 12345 Len=28
4	0.0017…	10.0.0.1	10.0.0.2	UDP	45	47345 → 53302 Len=3
5	0.0025…	10.0.0.2	10.0.0.1	UDP	44	53302 → 47345 Len=2
64	18.910…	10.0.0.2	10.0.0.1	UDP	10…	44048 → 39435 Len=1032
65	18.910…	10.0.0.1	10.0.0.2	UDP	46	39435 → 44048 Len=4
66	18.910…	10.0.0.2	10.0.0.1	UDP	10…	44048 → 39435 Len=1032
67	18.913…	10.0.0.2	10.0.0.1	UDP	10…	44048 → 39435 Len=1032
68	18.913…	10.0.0.1	10.0.0.2	UDP	46	39435 → 44048 Len=4
69	18.913…	10.0.0.2	10.0.0.1	UDP	10…	44048 → 39435 Len=1032
70	18.913…	10.0.0.1	10.0.0.2	UDP	46	39435 → 44048 Len=4
71	18.913…	10.0.0.2	10.0.0.1	UDP	50	44048 → 39435 Len=8
72	18.913…	10.0.0.1	10.0.0.2	UDP	46	39435 → 44048 Len=4

Tabla de Resultados

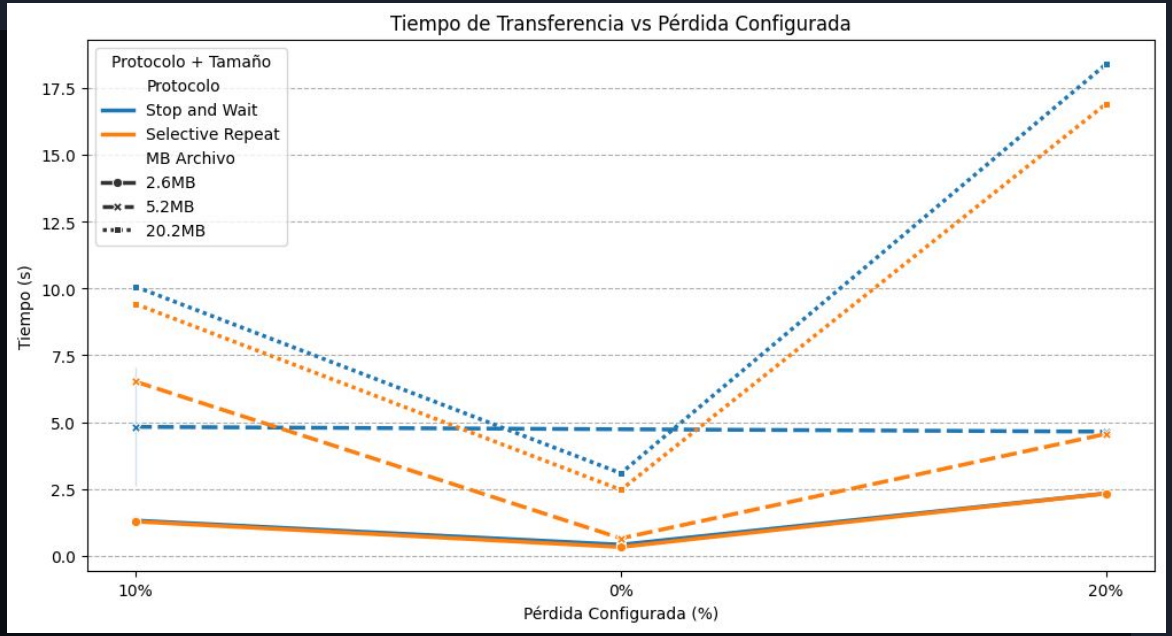
	Protocolo	MB Archivo	Loss	Tiempo_total_s	Total_bytes	RTT_estimado_s	Total_fragmentos	Bytes_por_segundo	Tasa_perdida_%
0	Stop and Wait	2.6MB	10%	1.311	3111556	0.000713	2873	2366823.68	5.37
1	Selective Repeat	2.6MB	10%	1.274	3115719	0.000896	2867	2439047.21	5.61
2	Selective Repeat	2.6MB	0%	0.319	2791390	0.000617	2569	8708082.32	0.03
3	Stop and Wait	2.6MB	0%	0.408	2788177	0.000388	2569	6814705.34	0.03
4	Stop and Wait	2.6MB	20%	2.326	3394207	0.000492	3140	653647.21	9.62
5	Selective Repeat	2.6MB	20%	2.326	3394207	0.000492	3140	653647.21	9.62
6	Stop and Wait	5.2MB	0%	2.607	6161172	0.000516	5689	2360633.09	5.04
7	Selective Repeat	5.2MB	0%	0.645	5583115	0.000447	5138	8624487.98	0.02
8	Stop and Wait	5.2MB	10%	7.035	6151539	0.000428	5680	873726.10	5.12
9	Selective Repeat	5.2MB	10%	6.511	6177214	0.000700	5686	948406.05	5.25
10	Stop and Wait	5.2MB	20%	4.643	6810889	0.000559	6299	1466190.14	9.81
11	Selective Repeat	5.2MB	20%	4.557	6916495	0.000658	6364	827590.94	10.34
12	Stop and Wait	20.2MB	0%	3.083	21421606	0.000108	19738	6943438.37	0.01
13	Selective Repeat	20.2MB	0%	2.461	21464687	0.000653	19755	8713479.95	0.00
14	Stop and Wait	20.2MB	10%	10.070	23656751	0.000514	21842	1366874.11	4.97
15	Selective Repeat	20.2MB	10%	9.421	23747663	0.000517	21856	2519612.45	4.83
16	Stop and Wait	20.2MB	20%	18.408	26318168	0.000028	24346	1429414.21	9.91
17	Selective Repeat	20.2MB	20%	16.922	26457821	0.001567	24350	1563003.44	10.25

Comparación

Promedio de todos los archivos.

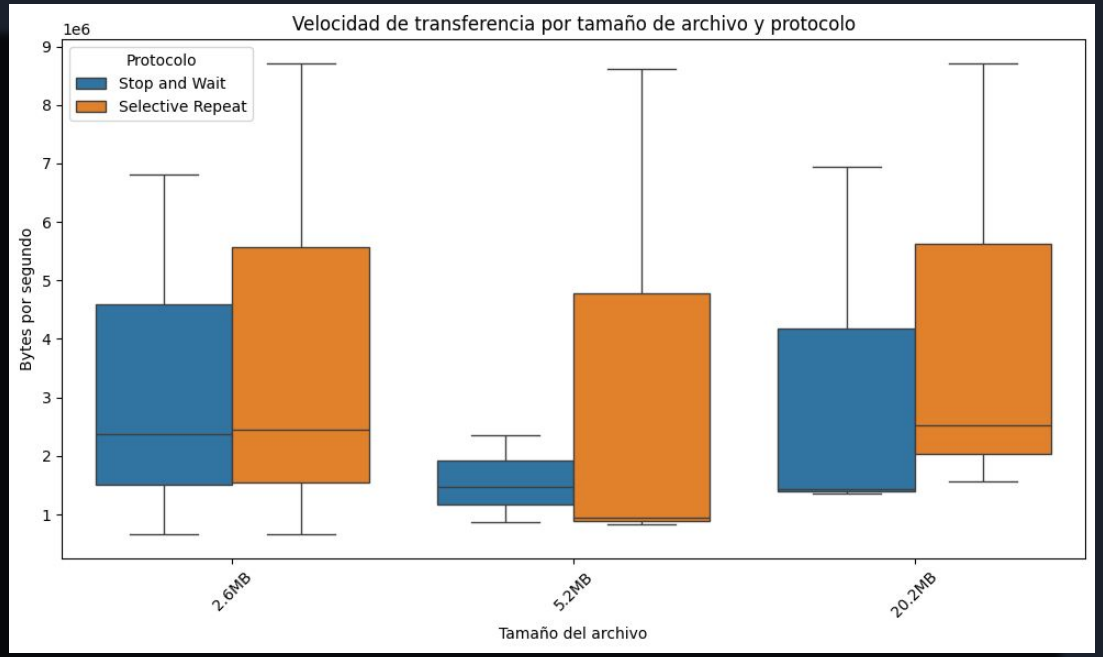


Comparación

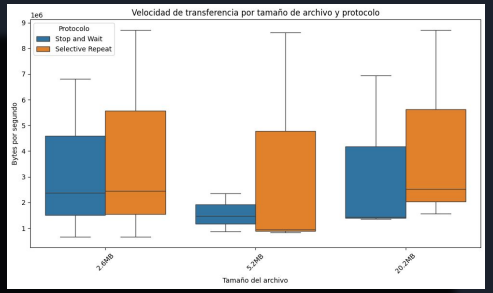


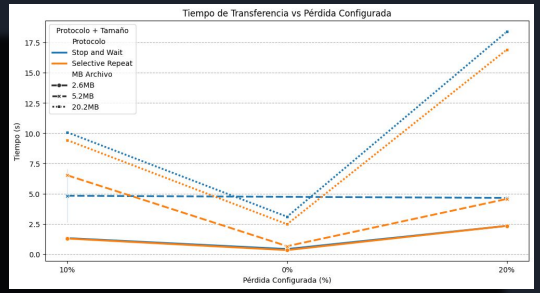
Comparación

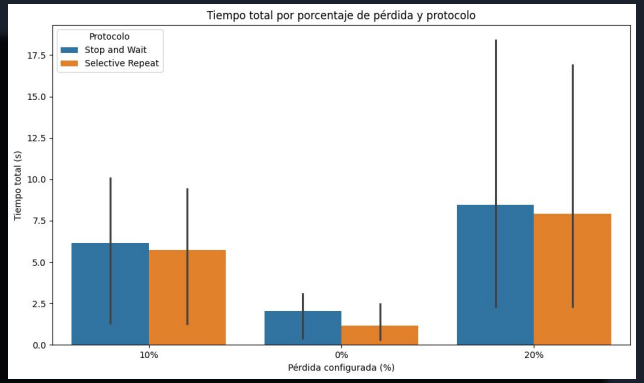
Selective Repeat siempre manda más Bytes por segundo.

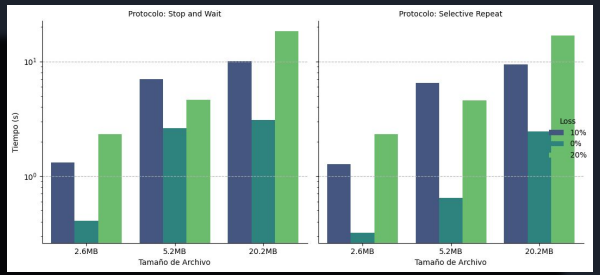


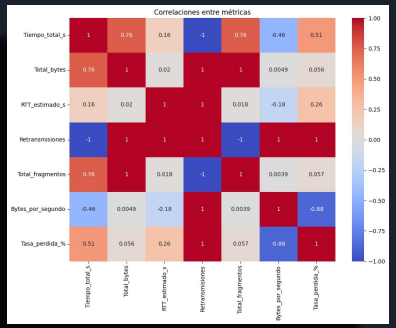
Conclusiones

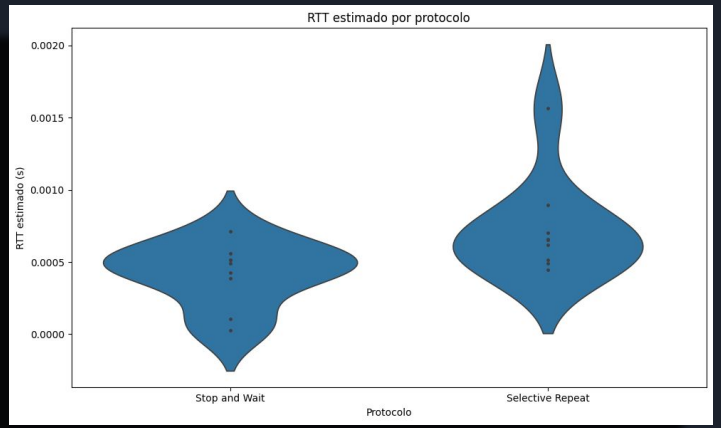












Dificultades encontradas

- Cumplir con el uso de herramientas de software para la simulación de topologías que no tenían un adecuado mantenimiento, con varios paquetes requeridos obsoletos e inclusive no disponibles, errores en funcionalidades básicas como guardar y cargar topologías.

- Replicar casos pensados teóricamente en topologías concretas.

- La implementación de la finalización de conexión resultó especialmente compleja pues no se lograba diferenciar un punto claro para dejar de solicitar ACKs de los últimos paquetes.

- Diseñar una transición limpia entre la comunicación llevada a cabo en el handhsake a la transmisión de la lógica de las aplicaciones.

Gracias