Ayuda Práctica: Capa de Transporte, TCP

Matías Robles [*], Andres Barbieri [**]

22 de abril de 2020

[*] mrobles@info.unlp.edu.ar
[**] barbieri@cespi.unlp.edu.ar

1. TCP (Transport Control Protocol)

TCP (Transport Control Protocol) es un protocolo de la capa de transporte que ofrece un servicio orientado a conexión. Las funcionalidades que ofrece son:

- Establecimiento y cierre de conexión.
- Detección de errores en bits de datos y encabezado, mediante checksum.
- Control, detección de errores, pérdidas, duplicados, mediante Go-Back-N o Selective-Repeat.
- Control de flujo.
- Control de congestión.
- Multiplexación mediante números de puertos.

La unidad de datos que envía o recibe, PDU (Protocol Data Unit), es conocido con el nombre de segmento TCP o directamente segmento. Las aplicaciones que requieran de una entrega fiable y orden de secuencias de datos deberían utilizar este protocolo. La figura 1 presenta el formato de segmento TCP. Este protocolo esta definido en el documento [RFC-793]. El número de protocolo es 6 (0x06 en hexadecimal) cuando se encapsula en el Protocolo de Internet (IP).

```
? grep tcp /etc/protocols
tcp 6 TCP # transmission control protocol
...
```

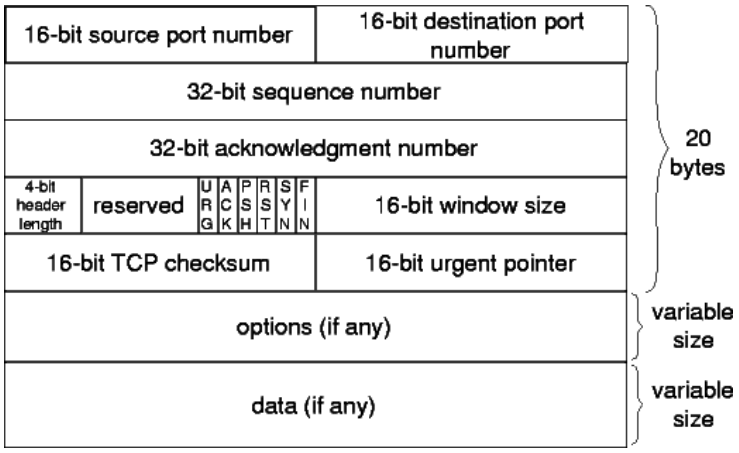

Figura 1: Diagrama de Segmento TCP.

2. Ejemplos Sencillos con TCP

Para estos ejemplos se utilizará la plataforma GNU/Linux con los comandos Netcat `nc(1)`, `telnet(1)`, se capturará y visualizará tráfico con las herramientas `tcpdump(8)`, `WIRESHARK(1)` o `ETHEREAL(1)` [COM05]. Los mismos se ejecutan en dos topologías de prueba, mostradas en las figuras 2 y 3.

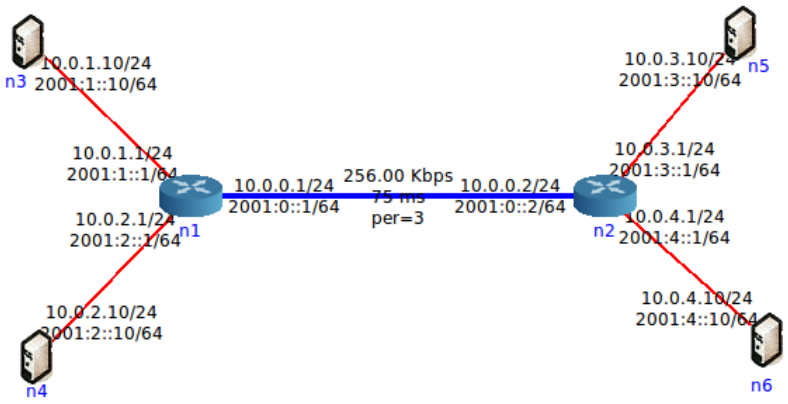

Figura 2: Topología de test uno.

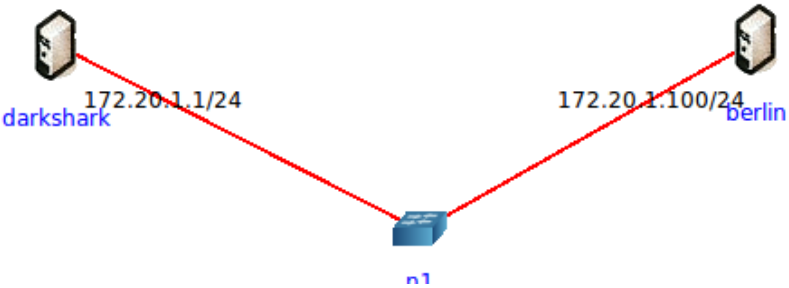

Figura 3: Topología de test dos.

2.1. Verificar los Procesos TCP

Primero se verifican los procesos utilizando comunicación TCP con el comando `netstat(8)`.

```
root@berlin:~# netstat -atnp
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address State   PID/Program name
```

```
tcp        0      0 0.0.0.0:37              0.0.0.0:*       LISTEN 5422/inetutils-inet
tcp        0      0 0.0.0.0:7               0.0.0.0:*       LISTEN 5422/inetutils-inet
tcp        0      0 0.0.0.0:9               0.0.0.0:*       LISTEN 5422/inetutils-inet
tcp        0      0 0.0.0.0:13              0.0.0.0:*       LISTEN 5422/inetutils-inet
tcp        0      0 0.0.0.0:19              0.0.0.0:*       LISTEN 5422/inetutils-inet
tcp        0      0 10.20.1.100:53          0.0.0.0:*       LISTEN 5110/named
tcp        0      0 172.20.1.100:53         0.0.0.0:*       LISTEN 5110/named
tcp        0      0 127.0.0.1:53            0.0.0.0:*       LISTEN 5110/named
tcp        0      0 127.0.0.1:953           0.0.0.0:*       LISTEN 5110/named
tcp6       0      0 :::80                   :::*            LISTEN 5250/apache2
tcp6       0      0 :::22                   :::*            LISTEN 5195/sshd

root@berlin:~# netstat -atp
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address State   PID/Program name
tcp        0      0 *:time                  *:*             LISTEN 5422/inetutils-inet
tcp        0      0 *:echo                  *:*             LISTEN 5422/inetutils-inet
tcp        0      0 *:discard               *:*             LISTEN 5422/inetutils-inet
tcp        0      0 *:daytime               *:*             LISTEN 5422/inetutils-inet
tcp        0      0 *:chargen               *:*             LISTEN 5422/inetutils-inet
tcp        0      0 10.20.1.100:domain      *:*             LISTEN 5110/named
tcp        0      0 berlin.cities.or:domain *:*             LISTEN 5110/named
tcp        0      0 localhost.locald:domain *:*             LISTEN 5110/named
tcp        0      0 localhost.localdoma:953 *:*             LISTEN 5110/named
tcp6       0      0 *:www                   *:*             LISTEN 5250/apache2
tcp6       0      0 *:ssh                   *:*             LISTEN 5195/sshd
```

Se puede utilizar también el comando `ss(8)`.

```
root@berlin:~# ss -a
State      Recv-Q Send-Q           Local Address:Port               Peer Address:Port
LISTEN     0      0                    *:echo                          *:*
LISTEN     0      0                    *:discard                       *:*
LISTEN     0      0                    *:daytime                       *:*
LISTEN     0      0                    *:chargen                       *:*
LISTEN     0      0                    *:ssh                           *:*
LISTEN     0      0                    *:time                          *:*
LISTEN     0      0                   :::ssh                          :::*
...
```

El nombre de los servicios se resuelve de acuerdo al archivo `/etc/services(5)`.

2.2. Levantar Proceso TCP

Se levanta un servicio TCP, se chequea que este esperando “conexiones”.

```
root@berlin:~# nc -l -p 11111

root@berlin:~# netstat -atnp | grep 1111
tcp        0      0 0.0.0.0:11111           0.0.0.0:*     LISTEN      5456/nc
```

En las versiones más nuevas de netcat el switch `-p` es incompatible con `-l`, por lo que se debe indicar el comando sin el parámetro `-p`. Luego se generan "conexiones", se tipea entrada y se recibe salida de texto. Previo a esto se capturan los segmentos TCP.

```
root@darkshark:~# tcpdump -n -i tap0 -s 1500 -w 00-tcp.pcap port 11111

andres@darkshark:~? telnet 172.20.1.100 11111
Trying 172.20.1.100...
Connected to 172.20.1.100.
Escape character is '^]'.

root@berlin:~# netstat -atnp | grep 11111
tcp        0      0 172.20.1.100:11111    172.20.1.1:41749    ESTABLISHED5456/nc

andres@darkshark:~? netstat -atnp | grep 11111
tcp        0      0 172.20.1.1:41749      172.20.1.100:11111 ESTABLISHED 9447/telnet
```

En la figura 4 se muestra un segmento de la captura `00-tcp.pcap`.

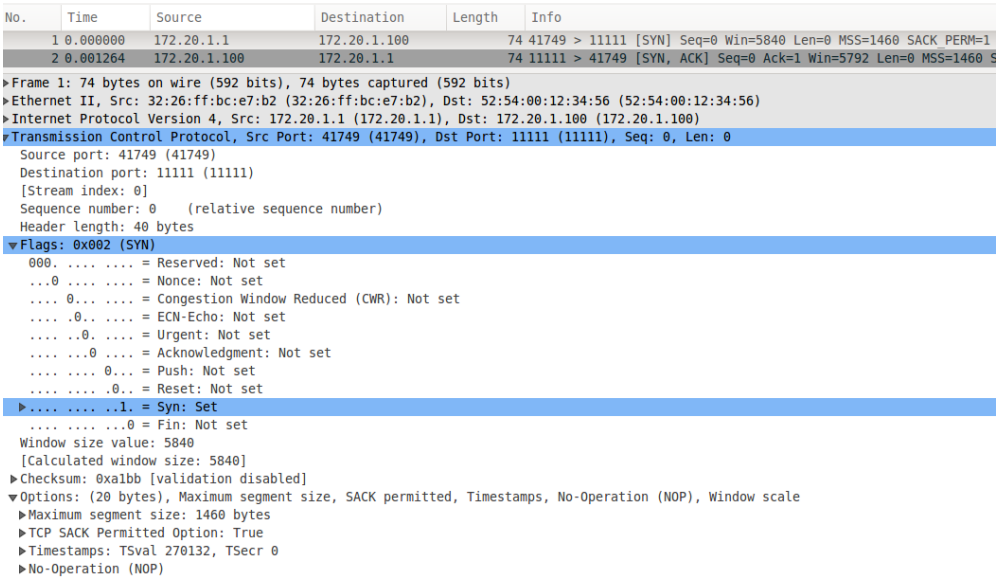

Figura 4: Segmento TCP de la captura inicial.

Para las pruebas, lo "tipeado" por el cliente se marca en letras MAYÚSCULAS.

```
andres@darkshark:~? telnet 172.20.1.100 11111
Trying 172.20.1.100...
Connected to 172.20.1.100.
Escape character is '^]'.
hola
HOLA COMO VA?,
bien
OK,
CHAU
^]

telnet> quit
Connection closed.

root@berlin:~# nc -l -p 11111
hola
HOLA COMO VA?,
bien
OK,
CHAU
root@berlin:~# netstat -atnp | grep 11111

andres@darkshark:~? netstat -atnp | grep 11111
tcp        0      0 172.20.1.1:41749   172.20.1.100:11111      TIME_WAIT   -

root@berlin:~# netstat -atnp | grep 11111
```

La figura 5 presenta el flujo de los segmentos de la captura `00-tcp.pcap`.

En el ejemplo se ve que una vez que se establece la conexión, el servidor no brinda más acceso a otros clientes. Esto se debe a un funcionamiento particular del comando Netcat. Los servicios TCP en general no funcionan de esta forma.

```
root@berlin:~# nc -l 11111

andres@darkshark:~? telnet 172.20.1.100 11111
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.

root@darkshark:~# tcpdump -n -i tap0 -s 1500 -w 00-tcp-rst.pcap port 11111

andres@darkshark:~? telnet 172.20.1.100 11111
Trying 172.20.1.100...
telnet: Unable to connect to remote host: Connection refused
```

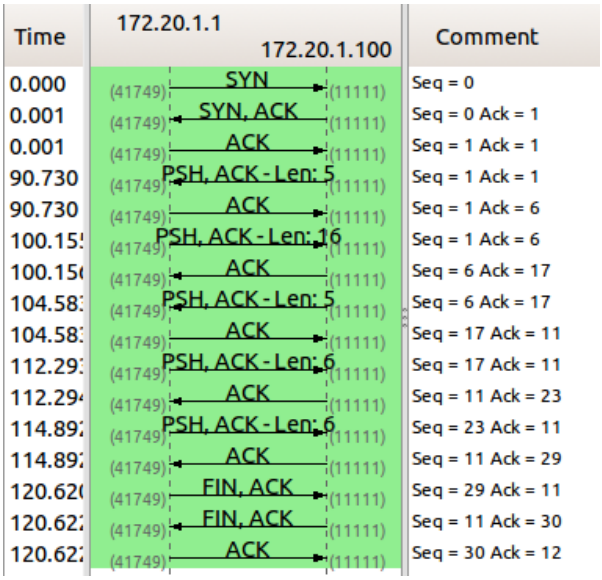

Figura 5: Diagrama de flujo de la sesión TCP de la captura.

```
root@berlin:~# netstat -atnp | grep 11111
tcp        0      0 172.20.1.100:11111    172.20.1.1:48749    ESTABLISHED5456/nc

andres@darkshark:~? netstat -atnp | grep 11111
tcp        0      0 172.20.1.1:48749      172.20.1.100:11111 ESTABLISHED 9447/telnet
```

No quedó el proceso manteniendo el estado LISTEN. De esta forma se ve un envío de un mensaje RST (reset), para indicar que no hay más procesos escuchando allí. Si se quiere mantener la posibilidad de recibir nuevas conexiones con Netcat, aunque serán atendidas de forma secuencia, se puede usar la opción `-k`.

```
root@berlin:~# nc -l 11111 -k

andres@darkshark:~?  telnet 172.20.1.100 11111
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.

andres@darkshark:~?  telnet 172.20.1.100 11111
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
```

Otra forma de ver un RST es lanzando la conexión a otro puerto donde sabemos que no hay nada escuchando.

```
andres@darkshark:~? telnet 172.20.1.100 50000

Trying 172.20.1.100...
telnet: Unable to connect to remote host: Connection refused
```

Los RST confirman indicando que esperan el siguiente número de secuencia y en genera van establecidos a (0) cero en su secuencia.

3. Ejemplo de TCP múltiple

Los servicios TCP, como se menciono antes, funcionan tradicionalmente manteniendo el puerto del servidor activo, pudiendo aceptar accesos múltiples. Este servicio se brinda generando una nueva tarea de atención por cliente que se conecta dando concurrencia a diferencia de un servicio secuencial. Si se genera la conexión contra un servicio simple de TCP como ECHO, servido en este caso por el super daemon `inetd(8)`, se puede observar que se atienden conexiones TCP simultáneas.

```
root@berlin:~# grep echo /etc/services
echo 7/tcp
echo 7/udp

root@berlin:~# cat /etc/inetd.conf

#
#       inetd internal services
#
...
echo     dgram  udp4 nowait root internal
...

root@berlin:~# /etc/init.d/inetutils-inetd restart

root@berlin:~# netstat -atnp | grep :7
tcp        0      0 0.0.0.0:7               0.0.0.0:* LISTEN    5422/inetutils-inet

root@berlin:~# tcpdump -n -i tap0 -s 1500 -w 01-tcp-mux.pcap port 7
```

Se establecen 2 (dos) conexiones y se mantiene el puerto aún en estado de espera por nuevas.

```
andres@darkshark:~? telnet 172.20.1.100 7

andres@darkshark:~? telnet 172.20.1.100 7

root@berlin:~# netstat -atnp | grep :7
tcp        0      0 0.0.0.0:7      0.0.0.0:*         LISTEN     5422/inetutils-inet
```

```
tcp        0      0 172.20.1.100:7 172.20.1.1:60244  ESTABLISHED5479/-echo [172.20.1.1]
tcp        0      0 172.20.1.100:7 172.20.1.1:60243  ESTABLISHED5478/-echo [172.20.1.1]
```

Luego se envían datos. Como el servicio replica lo que envia el cliente se tiene el efecto del eco.

```
andres@darkshark:~? telnet 172.20.1.100 7
Trying 172.20.1.100...
Connected to 172.20.1.100.
Escape character is '^]'.
hola
hola
^]
telnet> quit
Connection closed.

andres@darkshark:~? telnet 172.20.1.100 7
Trying 172.20.1.100...
Connected to 172.20.1.100.
Escape character is '^]'.
AAAA
AAAA
DDDDD
DDDDD
^]
telnet> quit
Connection closed.
```

4. Ejemplo de Análisis de Estados TCP

El diagrama de estados TCP se pueden ver en la figura 6. El estado inicial del servidor es **LISTEN**, este estado se crea a partir de que el servidor hace la llamada de sockets `listen(2)`. Una vez invocada y procesada la llamada, el proceso comunica al SO (o donde resida la implementación TCP) que establezca las conexiones al puerto indicado con la llamada anterior, `bind(2)`. Ambas en un sistema UN*X son *system calls*.

4.1. Establecimiento de Conexión

El cliente inicia su estado a partir de la llamada `connect(2)` mediante la cual comunica al SO que establezca la conexión TCP enviando un **SYN**. El estado es **SYN_SENT**. En la realidad este estado dura muy poco tiempo, ya que la conexión se establece de forma rápida. Este estado se puede ver en el sistema filtrando el puerto de entrada en el servidor o intentando una conexión hacia un host inalcanzable que no envíe **RST** y no acepte la conexión. Los routers no deberían detectar que el host es inalcanzable sino generarían un ICMP Host Unreachable (para evitarlo se podría filtrar con la herramienta `iptables(8)`). El cliente se dice que realiza un *Active Open*.

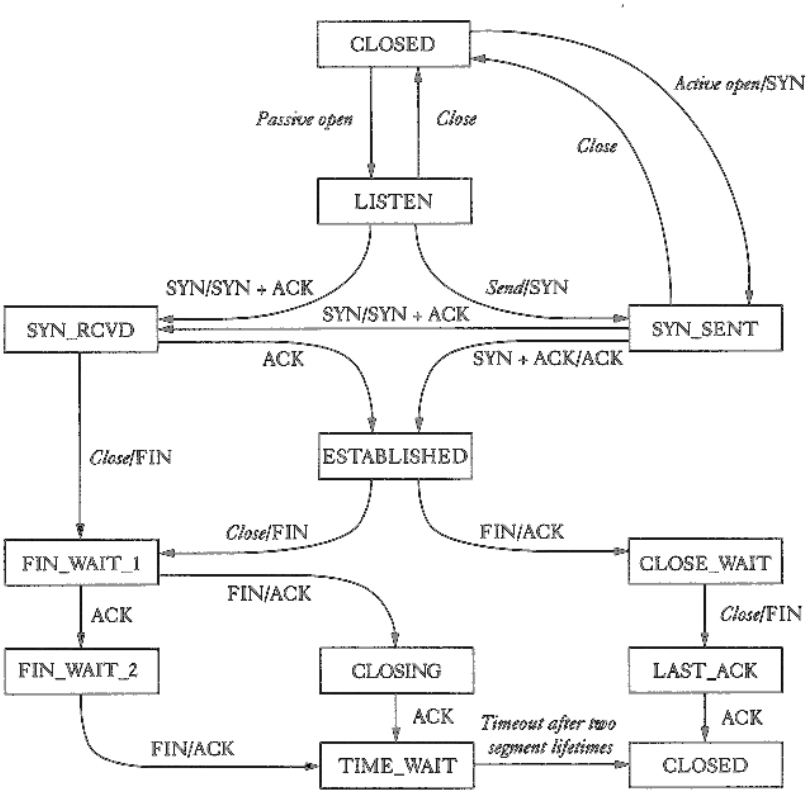

Figura 6: Máquina de estados reducida de TCP.

```
root@berlin:~# iptables -I INPUT --proto TCP --dport 7 -j DROP

root@darkshark:~# tcpdump -n -i tap0 -s 1500 -w 02-tcp-synsent.pcap port 7

andres@darkshark:~? telnet 172.20.1.100 7
Trying 172.20.1.100...

andres@darkshark:~? netstat -atnp | grep :7
tcp        0      1 172.20.1.1:38193 172.20.1.100:7 SYN_SENT  15361/telnet
```

Luego de un tiempo ...

```
andres@darkshark:~? telnet 172.20.1.100 7
Trying 172.20.1.100...
telnet: Unable to connect to remote host: Connection timed out
```

En GNU/Linux por default intenta las veces de acuerdo a lo establecido en la variable de kernel `tcp_syn_retries`. El tiempo de espera entre intento de conexión (backoff) es de crecimiento exponencial.

El tiempo total con 5 intentos que se indica es de aprox. 180 seg. Para la captura `03-tcp-synsent.pcap` se observa: 3+6+12+24+48+96 = 189 seg. En la figura 7 se ve el diagrama de flujo de la conexión fallida.

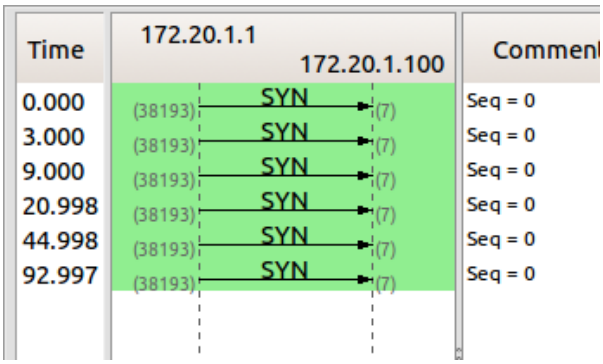

Figura 7: Flujo de segmentos de la conexión fallida.

```
root@darkshark:~# sysctl net.ipv4.tcp_syn_retries
net.ipv4.tcp_syn_retries = 5

root@darkshark:~# sysctl net.ipv4.tcp_syn_retries=1
net.ipv4.tcp_syn_retries = 1

root@darkshark:~# date;telnet 1.1.1.1 7;date
Sun Apr 25 16:37:44 ART 2010
Trying 1.1.1.1...
telnet: Unable to connect to remote host: Connection timed out
Sun Apr 25 16:37:53 ART 2010
```

En el caso que el SYN fuese aceptado, una vez que logra alcanzar el servidor y se recibe, va a pasar el servicio a estado **SYN_RECVD**, aunque va a mantener el estado **LISTEN** para otros clientes. Para ver este estado en el servidor se puede filtrar la entrada de la respuesta del **SYN+ACK** desde el servidor en el cliente y habilitar la entrada en el servidor que se había denegado.

```
root@berlin:~# iptables -D INPUT --proto TCP --dport 7 -j DROP

root@darkshark:~# iptables -I INPUT --proto TCP --sport 7 -j DROP

root@darkshark:~# tcpdump -n -i tap0 -s 1500 -w 03-tcp-synrecvd.pcap port 7

andres@darkshark:~? netstat -atnp | grep :7
tcp        0      1 172.20.1.1:56982  172.20.1.100:7    SYN_SENT 16396/telnet

root@berlin:~# netstat -atnp  | grep ":7"
```

```
tcp        0      0 0.0.0.0:7           0.0.0.0:*         LISTEN      5422/inetutils-inet
tcp        0      0 172.20.1.100:7      172.20.1.1:56982  SYN_RECV    -
```

En este caso se dice que el servidor realiza un *Passive Open*. De la misma forma que se puede controlar la cantidad de **SYN** se puede hacerlo con los **SYN+ACK**.

```
root@berlin:~# sysctl net.ipv4.tcp_synack_retries
net.ipv4.tcp_synack_retries = 5
```

Si luego se saca el filtro y se intenta nuevamente la conexión, se observa que se establece. pasando en ambos lados al estado **ESTABLISHED**.

```
root@darkshark:~# tcpdump -n -i tap0 -s 1500 -w 04-tcp-estab.pcap port 7

root@darkshark:~# iptables -D INPUT --proto TCP --sport 7 -j DROP

andres@darkshark:~? telnet -b172.20.1.1 172.20.1.100 7
Trying 172.20.1.100...
Connected to 172.20.1.100.
Escape character is '^]'.

root@berlin:~# netstat -atnp  | grep ":7"
tcp        0      0 0.0.0.0:7       0.0.0.0:*         LISTEN     5422/inetutils-inet
tcp        0      0 172.20.1.100:7  172.20.1.1:39111  ESTABLISHED5549/-echo [172.20.

andres@darkshark:~? netstat -atn | grep ":7"
tcp        0      0 172.20.1.100:39111  172.20.1.1:7  ESTABLISHED
```

Otra forma de ver el funcionamiento de "Timeout of Connection Establishment" previo al establecimiento de la conexión, es también filtrando los ICMP de inalcanzable que podrían generar los routers intermedios cuando se hace el test sobre un host inexistente en otra red.

```
root@darkshark:~# iptables -I INPUT --proto ICMP -j DROP
```

4.2. Cierre de Conexión

De forma similar se puede realizar el análisis de los estados al momento de cerrar la conexión. Esto es un poco más complicado de hacerlo con los programas tradicionales ya que se requiere controlar cuando realizar la llamada `close(2)` o `shutdown(2)`.
La primer prueba se realiza generando un filtro posterior al establecimiento de la conexión.

```
andres@darkshark:~? telnet 172.20.1.100 7
Trying 172.20.1.100...
Connected to 172.20.1.100.
Escape character is '^]'.
^]
```

```
root@berlin:~# iptables -A INPUT --proto TCP --dport 7 -j DROP

andres@darkshark:~?
...
^]

telnet> quit
Connection closed.
```

En este caso el cliente ha enviado el **FIN** pasando al estado **FIN_WAIT1**, luego espera recibir el **ACK** para pasar al estado **FIN_WAIT2**.

```
andres@darkshark:~? netstat -atnp | grep :7
tcp        0      1 172.20.1.1:39111        172.20.1.100:7        FIN_WAIT1   -
```

Si recibe un **FIN+ACK** el cliente pasa directamente al estado **TIME_WAIT** debido al *Active Close*. Una vez que se des-filtra del lado de quién cerró la conexión de forma activa (el cliente) pasa al estado **TIME_WAIT**. Sino permanecería en el estado **FIN_WAIT1** hasta vencer temporizador que se verá más adelante.

```
andres@darkshark:~? netstat -atnp | grep :7
tcp        0      0 172.20.1.1:39111        172.20.1.100:7          TIME_WAIT   -
```

El servidor (*Passive Close*) directamente pasa la conexión a estado cerrado, lo cual elimina la máquina de estados y los recursos utilizados. No se verá con los comandos.

Si el otro extremo no llegase a enviar el **ACK** o el **FIN+ACK**, el que inició el cierre permanecerá en el estado **FIN_WAIT1/2** infinitamente. Para evitar esto las implementaciones luego de un tiempo pasan a **CLOSED** liberando los recursos. En GNU/Linux el control de este tiempo desde el estado **FIN_WAIT2** se realiza con el parámetro mostrado a continuación (por default 60 segundos):

```
root@darkshark:~# sysctl net.ipv4.tcp_fin_timeout
net.ipv4.tcp_fin_timeout = 60
```

Para salir del estado **FIN_WAIT1** (cerrarse) espera más tiempo, el mismo que estaría en **TIME_WAIT**. Para salir de estos estados se espera 2MSL (2 * Maximum Segment Lifetime) para pasar a **CLOSED**. Este tiempo es para evitar problemas con "Re-encarnaciones de segmentos TCP". MSL originalmente definido arbitrariamente en 2 min. Durante este tiempo no se debería poder utilizar la tupla (SRC-IP:src-port,DST-IP:dst-port), pero las implementaciones lo evitan no permitiendo usar el par (SRC-IP:src-port), indicando el error "Address already in use".

```
andres@darkshark:~? netstat -atnp | grep :7
tcp        0      0 172.20.1.1:39111        172.20.1.100:7    TIME_WAIT   -

andres@darkshark:~? nc -p 39111 172.20.1.100 7
nc: bind failed: Address already in use
```

Esto no es en general problema, pues el que cierra la conexión es el cliente y no tendría problema en utilizar cualquier otro puerto efímero en la siguiente conexión. Si sucediese en un servidor y este se cierra/"muere" repentinamente para volver a levantarlo en el puerto se debería esperar un tiempo. Este tiempo se podría evitar al momento de crear el socket con una opción `SO_REUSEADDR`.

```
SOCKET(2)

sock = socket(AF_INET, SOCK_STREAM, 0);

GETSOCKOPT/GETSOCKOPT(2)

int optbuf = 1;
setsockopt(sock, SOL_SOCKET,SO_REUSEADDR,(char*) &optbuf,
           (socklen_t) sizeof(optbuf))
```

También se puede configurar el re-uso de forma global, por ejemplo en GNU/Linux mediante las siguientes opciones.

```
root@n3~:# sysctl net.ipv4.tcp_tw_recycle
net.ipv4.tcp_tw_recycle = 0
root@n3~:# sysctl net.ipv4.tcp_tw_reuse
net.ipv4.tcp_tw_reuse = 0

root@n3~:# sysctl net.ipv4.tcp_tw_recycle=1
net.ipv4.tcp_tw_recycle = 1
root@n3~:# sysctl net.ipv4.tcp_tw_reuse=1
net.ipv4.tcp_tw_reuse = 1

root@n3~:# netstat -atnp | grep 8001;date
Mon Apr 27 06:59:37 PDT 2015

root@n3~:# nc -p 8001 10.0.3.10 7
aaa
aaa
^C
root@n3~:# netstat -atnp | grep 8001;date
tcp        0      0 10.0.1.10:8001       10.0.3.10:7             TIME_WAIT   -
Mon Apr 27 06:59:53 PDT 2015
root@n3~:# nc -p 8001 10.0.3.10 7
^C

root@n3~:# netstat -atnp | grep 8001;date
Mon Apr 27 07:00:16 PDT 2015
```

Si el filtro se pusiese en el cliente en lugar del servidor, el servidor queda en estado **LAST_ACK** ya que ha llamado a `close(2)` y ha mandado su **FIN+ACK**.

```
andres@darkshark:~? telnet 172.20.1.100 7
Trying 172.20.1.100...
Connected to 172.20.1.100.
Escape character is '^]'.

root@darkshark:~# iptables -I INPUT --proto TCP --sport 7 -j DROP

andres@darkshark:~?
^]

telnet> quit
Connection closed.

andres@darkshark:~? netstat -atnp | grep :7
tcp        0      1 172.20.1.1:56040 172.20.1.100:7   FIN_WAIT1   -

root@berlin:~# netstat -atnp  | grep :7
tcp        0    0 0.0.0.0:7           0.0.0.0:*        LISTEN   5422/inetutils-inet
tcp        0    1 172.20.1.100:7      172.20.1.1:56040 LAST_ACK -
```

Debido al filtro en el cliente quedan en **FIN_WAIT1** y **LAST_ACK**.
Para ver los estados **FIN_WAIT2** y **CLOSE_WAIT** una alternativa es tener control de la aplicación para indicarle cuando se quiere realizar un `close(2)`. A continuación se muestra un ejemplo con un ejecutable concebido para este propósito. Para proseguir con cada paso se debe dar un `<ENTER>`.
Primero lanzar el servidor:

```
root@berlin:~# ./tcp-write-server --help
usage: ./tcp-write-server [-l|--listen <listen-ip>] [-p|--port <port>] [-b|--blog <num>]
       [-s|--size <size>] [-i|--infinite] [-d|--delay <sec>] [-r|--reuse]
       [--help|-h]
default port 8000, default listen 0.0.0.0, default backlog 1, default size 32
default delay 0

root@berlin:~# ./tcp-write-server
Antes de socket()
<ENTER>
Antes de bind()
<ENTER>
Antes de listen()
<ENTER>
Antes de accept()
<ENTER>
Waiting connection on 0.0.0.0:8000 reuse:0
```

Luego lanzar el cliente:

```
root@darkshark:~# tcpdump -n -i tap0 -s 1500 -w 05-tcp-ad-hoc.pcap port 8000

andres@darkshark:~? ./tcp-read-client
usage: ./tcp-read-client -d|--dst <dest-ip> [-p|--dport <port>] [-s|--size <data size>]
       [-f|--from <src-ip>] [-z|--sport <src-port>]
default port 8000, default size 32

andres@darkshark:~? ./tcp-read-client -d 172.20.1.100
Antes de socket()
<ENTER>
Antes de bind()
<ENTER>
Antes de connect()
<ENTER>
Antes de read()
<ENTER>
Readed (32) 0/32
buffer: [!"#$%&'()*+,-./0123456789:;<=>?@] size: [32]
```

Ver los estados.

```
andres@darkshark:~? netstat -atnp | grep :8000
tcp        0      0 172.20.1.1:37039  172.20.1.100:8000  ESTABLISHED 14233/tcp-read-clie
```

Finalizar la conexión desde el cliente.

```
andres@darkshark:~?
...

Readed (32) 0/32
buffer: [!"#$%&'()*+,-./0123456789:;<=>?@] size: [32]
Antes de close()
<ENTER>
Antes de exit()
```

Esperar en el servidor y ver los estados.

```
root@berlin:~#
...
Connected from 172.20.1.1:48245
FINISH: Success
Antes de close()

andres@darkshark:~? netstat -atnp | grep :8000
tcp        0      0 172.20.1.1:37039  172.20.1.100:8000  FIN_WAIT2   -
```

```
root@berlin:~# netstat -atnp | grep :8000
tcp        0   0 0.0.0.0:8000      0.0.0.0:*          LISTEN     5361/tcp-write-serv
tcp        1   0 172.20.1.100:8000 172.20.1.1:37039   CLOSE_WAIT 5361/tcp-write-serv
```

Después de un tiempo (60 seg.) del estado **FIN_WAIT2** pasa solo a **CLOSED** en el cliente.

```
andres@darkshark:~? netstat -atnp | grep :8000
```

El servidor permanecerá en el estado **CLOSE_WAIT** hasta que realice el cierre.

```
root@berlin:~# netstat -atnp | grep :8000
tcp        0   0 0.0.0.0:8000      0.0.0.0:*          LISTEN     5361/tcp-write-serv
tcp        1   0 172.20.1.100:8000 172.20.1.1:37039   CLOSE_WAIT 5361/tcp-write-serv

root@berlin:~#
...
Connected from 172.20.1.1:48245
FINISH: Success
Antes de close()
<ENTER>
```

En este caso como la máquina de estados del cliente desapareció nunca recibirá un ACK del cliente.

```
root@berlin:~# netstat -atnp | grep :8000
tcp        0   0 0.0.0.0:8000       0.0.0.0:*         LISTEN     5361/tcp-write-serv
tcp        1   1 172.20.1.100:8000  172.20.1.1:37039  LAST_ACK   -
```

Otra forma de ver los estados `FIN_WAIT2` y `CLOSE_WAIT` es por ejemplo enviando a dormir a una de las partes, cliente o servidor, y luego terminando el otro extremo. Esto genera que el extremo que terminamos, envía el FIN y el stack TCP confirma con en ACK. Pero como la aplicación del otro extremo esta en estado dormida no va a enviar el EOF(End of FILE) para cerrar el stream, pues no se le ha podido notificar del EOF del otro extremo. En este caso hasta que no se despierte quedarán en estos dos estados. El extremo que terminamos quedará en `CLOSE_WAIT`. A continuación un ejemplo. El servidor en el port 7000 se activa.

```
root@n5:~# nc -l 7000
```

Se conecta el cliente.

```
root@n3:~# telnet 10.0.3.10 7000
Trying 10.0.3.10...
Connected to 10.0.3.10.
Escape character is '^]'.
^]

root@n5:~#  netstat -atnp | grep 7000
tcp        0      0 0.0.0.0:7000          0.0.0.0:*            LISTEN      242/nc
tcp        0      0 10.0.3.10:7000        10.0.1.10:39970      ESTABLISHED 242/nc
```

Enviamos a “dormir” al cliente.

```
root@n3:~#
^]
telnet> ^Z
[1]+  Stopped                 telnet 10.0.3.10 7000
```

Terminamos al servidor.

```
root@n5:~# killall nc ### Mata proceso server

root@n5:~# netstat -atnp | grep 7000
tcp        0      0 10.0.3.10:7000          10.0.1.10:39979 FIN_WAIT2   -
```

El servidor termino, el stack TCP/IP envío el FIN y se confirmo desde el extremo del cliente, pero aún resta recibir el FIN por parte del cliente. El cliente esta en “sleep” pero el stack TCP/IP queda a espera del EOF del cliente para cerrar en el otro sentido.

```
root@n5:~# netstat -atnp | grep 7000
tcp        1      0 10.0.1.10:39979   10.0.3.10:7000        CLOSE_WAIT  146/telnet
```

Si capturamos el tráfico solo vimos el cierre en un sentido, de servidor a cliente.

```
root@n5:~#  tcpdump -n -i eth0 tcp
...
06:06:31.126618 IP 10.0.3.10.7000 > 10.0.1.10.39979: Flags [F.], seq 1939519247,
 ack 4056951020, win 905, options [nop,nop,TS val 822621 ecr 806814], length 0
06:06:31.277949 IP 10.0.1.10.39979 > 10.0.3.10.7000: Flags [.], ack 1, win 913,
 options [nop,nop,TS val 822640 ecr 822621], length 0
```

Al levantar el cliente este es notificado del EOF desde el stack TCP/IP, y envía su FIN para finalmente cerrar en los dos sentidos.

```
root@n3:~# fg %1
telnet 10.0.3.10 7000
Connection closed by foreign host.
```

En la captura se ve el resto del tráfico de cierre.

```
...
06:07:22.371592 IP 10.0.1.10.39979 > 10.0.3.10.7000: Flags [F.], seq 4056951020,
 ack 1939519248, win 913, options [nop,nop,TS val 835414 ecr 822621], length 0
06:07:22.371660 IP 10.0.3.10.7000 > 10.0.1.10.39979: Flags [.], ack 1, win 905,
 options [nop,nop,TS val 835432 ecr 835414], length 0
```

Ahora los estados, de acuerdo a quién hizo el cierre activo, el servidor en este caso, queda en espera para el reuso de este port.

```
root@n3:~# netstat -atnp | grep 7000

root@n5:~# netstat -atnp | grep 7000
tcp        0      0 10.0.3.10:7000          10.0.1.10:39979         TIME_WAIT   -
```

4.3. Half-Close (Cierre a medias) de Conexión

TCP contempla, como se vio en los ejemplos anteriores, el cierre en 4 segmentos, FIN,ACK,FIN,ACK, aunque si ambas partes desean cerrar de inmediato la conexión se puede realizar en 3, FIN,ACK+FIN,ACK, donde el primer FIN también tiene un ACK de lo anteriormente transmitido (ver captura `00-tcp.pcap`). Entonces, para cerrar completamente (*Full-Close*) una conexión ambos extremos deben enviar el FIN. Cuando solo un extremo desea cerrar, pero el otro necesita seguir enviando información se puede intercambiar un FIN y luego un ACK y se tiene el cierre en un sentido *Half-Close*. Un uso posible es cuando se combina en un stream TCP la semántica de los pipes y redirecciones de UN*X. Por ejemplo si se envían datos y recién el extremo remoto los puede procesar cuando los a recibido a todos, en este caso indicado con un EOF(-1). El ejemplo que se utiliza en los textos es el uso de `rsh(1)` (Remote Shell) , antecesor de `ssh(1)` (Secure Shell). Se tiene remotamente el comando `sort(1)`, `md5sum(1)` o `sha256sum(1)`, y luego un archivo local que se desea procesar de forma remota. Se envía el archivo y recién cuando se ha pasado completamente se envía el EOF, que produce el cierre del stream TCP en un sentido, pero permite que el otro extremo procese los datos y envíe la respuesta.

```
local# rsh remote sort < datafile
local# rsh remote sha256sum < datafile
```

A continuación se muestra un ejemplo con ssh.

```
root@n3~# cat /tmp/uno.txt
1
root@n3:~# ssh user@10.0.3.10 md5sum < /tmp/uno.txt
user@10.0.3.10's password:
b026324c6904b2a9cb4b88d6d61c81d1  -

root@n3~# tcpdump -n -i eth0 tcp
...
19:44:51.621868 IP 10.0.1.10.53926 > 10.0.3.10.22: Flags [P.],seq 1968:2032,ack 2128,
 win 1270, options [nop,nop,TS val 425751 ecr 425732], length 64
19:44:51.622038 IP 10.0.1.10.53926 > 10.0.3.10.22: Flags [F.], seq 2032, ack 2128,
 win 1270, options [nop,nop,TS val 425751 ecr 425732], length 0
19:44:51.773231 IP 10.0.3.10.22 > 10.0.1.10.53926: Flags [.], ack 2033,
 win 1404, options [nop,nop,TS val 425769 ecr 425751], length 0
19:44:51.775892 IP 10.0.3.10.22 > 10.0.1.10.53926: Flags [F.], seq 2128, ack 2033,
 win 1404, options [nop,nop,TS val 425770 ecr 425751], length 0
19:44:51.775979 IP 10.0.1.10.53926 > 10.0.3.10.22: Flags [.], ack 2129,
 win 1270, options [nop,nop,TS val 425789 ecr 425770], length 0

root@n3~# tcpdump -n -i eth0 tcp -w 06-tcp-half-close.pcap
```

5. Ejemplo de Análisis de Flags TCP

Los flags TCP definidos en la primera RFC son:

URG (urgent) flag (1 bits): si esta seteado indica que el campo **Urgent Pointer** tiene datos válidos. Este campo "apunta" (tiene un offset dentro del payload del segmento actual) indicando donde terminan los datos urgentes. El receptor deberá leer hasta este punto para recuperar los datos urgentes. El proceso receptor será interrumpido por el SO (si este se ha registrado para este evento) al recibir el módulo TCP un segmento con datos urgentes. Este puntero es utilizado por algunas implementaciones como una manera de simular datos fuera de banda OOB (out-of-band). Por ejemplo implementaciones de BSD sacan el byte apuntado por URG PTR (último byte de datos urgentes) del stream (flujo) TCP tradicional y solo se lo entregan a la aplicación si se llama la recepción, `recv(2)/read(2)`, con la opción `MSG_OOB`. TCP en su definición **NO soporta** datos OOB, si se desea esto se debería utilizar otra conexión y recibir con `select(2)`). Para obtener el comportamiento estándar se debería crear el socket TCP al llamar `socket(7)` con la opción `SO_OOBINLINE`, manteniendo los datos urgentes dentro del stream TCP único. Es utilizado cuando se envían datos y estos deben ser procesados tan pronto como sea posible, por ejemplo mediante una conexión remota (telnet/rsh/ssh) se desea enviar una interrupción (abortar) la salida en "pantalla" de un proceso y de esta manera requerir respuesta inmediata.

PSH (push) flag (1 bits): si esta seteado indica que los datos en buffer de recepción deben ser entregados al proceso de espacio de usuario en el próximo `read(2)`. Cuando una aplicación hace varios `write(2)` o `send(2)` sin configurar el **PSH**, TCP podría agruparlos y encolarlos internamente, sin entregarlos inmediatamente a la aplicación. Con este flag se indica que no agrupe y lleve los datos a la aplicación si esta los solicita. TCP debe tratar de enviar segmentos con con el máximo tamaño para obtener mejor performance. Su uso se puede ver en las capturas anteriores.

ACK (acknowledgement) flag (1 bits): el flag **ACK** indica que el campo **ACK Number** es válido. El segmento esta confirmando datos. Su uso se puede ver en las capturas anteriores: `00-tcp.pcap`.

RST (reset) flag (1 bits): resetea la conexión TCP del receptor. Paquetes erróneos son respondidos con segmentos con este flag, por ejemplo uno fuera de secuencia o un ACK de un mensaje que nunca se envió. Su uso se puede ver en las capturas anteriores: `00-tcp-rst.pcap`.

SYN (synchronize) flag (1 bits): configurado en la apertura de una conexión TCP. Su uso se puede ver en las capturas anteriores: `00-tcp.pcap`.

FIN (finished) flag (1 bits): configurado cuando no se va a enviar más datos. Usados en el cierre de la conexión. Su uso se puede ver en las capturas anteriores: `00-tcp.pcap`.

Ejemplos utilizando el Puntero de datos Urgentes de TCP. En este caso se envía un dato de control desde la herramienta `telnet(1)`.

```
andres@darkshark:~? man telnet

TELNET(1)
...
send synch: Sends the TELNET SYNCH sequence.  This sequence causes the remote system
to discard all previously typed (but not yet read) input.  This sequence is sent as
TCP urgent data (and may not work if the remote system is a 4.2BSD system -- if it
```

```
doesn't work, a lower case ``r'' may be echoed on the terminal).
...


root@darkshark:~# tcpdump -n -i tap0 -w 00-tcp-urg.pcap port 7

andres@darkshark:~? telnet 172.20.1.100 7
Trying 172.20.1.100...
Connected to 172.20.1.100.
Escape character is '^]'.
ddd
ddd
dddd
dddd
^]

telnet> send synch
sss
ss
^]

telnet> quit
Connection closed.
```

6. Opciones TCP

Dentro de las opciones que se pueden encontrar en el encabezado de TCP se pueden mencionar las siguientes (`http://www.iana.org/assignments/tcp-parameters/tcp-parameters.xhtml`).

- End of Option List (0), RFC-793.
- No-Operation (1), RFC-793.
- MSS(Maximum Segment Size) (2), RFC-793.
- Window Scale (3) RFC-7323.
- SACK(Selective ACK) (4,5) RFC-2018.
- Timestamps (8).

Las opciones 1 y 0 se utilizan para completar el encabezado a 32bits e indicar el fin de las opciones. El MSS se utiliza para poder obtener una mejor utilización de los recursos. A continuación se muestra un ejemplo.

6.1. MSS (Maximum Segment Size)

depende el MTU con el primer hop, pero nunca será menor que el valor indicado por la variable

```
root@n3:~# sysctl net.ipv4.route.min_adv_mss
net.ipv4.route.min_adv_mss = 256
```

Al realizar PMTU discovery se rige por las siguientes variables.

```
root@n3:~# sysctl net.ipv4.tcp_base_mss
net.ipv4.tcp_base_mss = 512

root@n3:~# sysctl net.ipv4.tcp_mtu_probing
et.ipv4.tcp_mtu_probing = 0
```

Para ver como funciona el MSS en las opciones de TCP se puede ver el siguiente ejemplo:

```
root@n5:~# nc -l 7000

root@n3:~# ifconfig eth0 | grep MTU
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
root@n3:~# ifconfig eth0 mtu 1000
root@n3:~# ifconfig eth0 | grep MTU
          UP BROADCAST RUNNING MULTICAST  MTU:1000  Metric:1
root@n3:~# nc 10.0.3.10 7000

root@n5:~## tcpdump -n -i eth0 tcp -vvv
06:39:21.608761 IP (tos 0x0, ttl 62, id 14683, offset 0, flags [DF], proto TCP (6),
 length 60) 10.0.1.10.39991 > 10.0.3.10.7000: Flags [S], cksum 0xaab3 (correct),
 seq 1873357004, win 9600, options [mss 960,sackOK,TS val 1315223 ecr 0,nop,wscale 4],
 length 0
06:39:21.608851 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto TCP (6),
 length 60) 10.0.3.10.7000 > 10.0.1.10.39991: Flags [S.], cksum 0x1842 (incorrect -> 0xb42e),
 seq 3572234949, ack 1873357005, win 14480, options [mss 1460,sackOK,TS val 1315242
 ecr 1315223,nop,wscale 4], length 0
06:39:21.759840 IP (tos 0x0, ttl 62, id 14684, offset 0, flags [DF], proto TCP (6),
 length 52) 10.0.1.10.39991 > 10.0.3.10.7000: Flags [.], cksum 0x190a (correct), seq 1,
 ack 1, win 600, options [nop,nop,TS val 1315261 ecr 1315242], length 0

root@n3:~# ifconfig eth0 mtu 1500
```

La resta sobre la MTU es de 40 bytes, considerando un 20bytes de header IP y 20 de header TCP. En general el valor es $1500 - 40 = 1460$ bytes.

6.2. Timestamps

De acuerdo a RFC-1323 y al reemplazo, RFC-7323, permite la medición de RTT, Round-Trip Time Measurement (RTTM) de una forma más eficiente y brinda protección contra los problemas que puede generar el reinicio de los números de secuencia en enlaces con alto bandwidth, Protection Against Wrapped Sequences (PAWS). En GNU/Linux se establece con la variable.

```
root@n3:~# sysctl net.ipv4.tcp_timestamps
 net.ipv4.tcp_timestamps = 1
```

6.3. SACK(Selective ACK)

De acuerdo a RFC-2018, permite confirmar por sub-intervalos con un algoritmo de Selective-Repeat. El ejemplo se ve en la sección de Flow Control.

6.4. Window Scaling

De acuerdo RFC-7323, TCP Extensions for High Performance, permite establecer una valor de la ventana para control de flujo más importante que los 64K del campo **Win**. Permite mediante una opción multiplicar este valor por una potencia de 2. El ejemplo se ve en la sección de TCP Performance.

7. Control de Errores y de Flujo (Error/Flow Control)

En la siguiente sección se muestra un ejemplo para poder analizar como TCP hace el control de errores y las retransmisiones y como va variando el valor de la ventana de recepción para hacer control de flujo.

```
root@n5:~# nc -l 7000 | base64

root@n3:~# nc 10.0.3.10 7000  < /dev/urandom

root@n5:~# nc -l 7000 | base64
...
n0KfSop04yCYEo6qjgIUS6cIVlhjfSWHrApf7VyHuelCO6LYJ4vD/gtIKgO0IsF1IjvKfrsrVJVg
m2EyuukUwQxn6OUEGpHyp1rr2Sp7j7XuaeVdj070mv/A68d+flcrIbmSiJjmMb5Z75VWVj5hRvRw
bJyGDvX/84sRnaJrha3OlCmsq8IAzLa06+Dl0TljBqL8bL3ZQzUR47SpxuCjEL+PWTHflKX4dzSH
9372GSQ748ewklfpy4VpdwXTcYQ4e9xwumciNLzohzxzP7HYJogZ
^Z
...
```

Luego que se envío a dormir el proceso, este no va a leer datos del buffer de entrada de TCP por lo cual la ventana tendrá que ir “cerrándose”, en este caso hasta llegar a 0 (cero).

```
root@n5:~# jobs
[1]+  Stopped                 nc -l 7000 | base64
```

Luego se reestablece, por lo cual la ventana vuelve a permitir el envío de datos.

```
root@n5:~# fg %1
m4KceZcRKLBScRfJsOBoRfD6BBcTKRffUY+zLMg23fDSIxhWHY/yumxm9FffBRqMzztPh+8bwQBa
hewdMrgps9aRII8U3e+HgQQYta3x9jjdT3cYqIzma9BsseOBRUcd9pv/+F3Nz027O3IWcq9R..

root@n5:~# tcpdump -n -i eth0 tcp -s 100 -w 07-tcp-flow-control.pcap
...
```

La figura 8 muestra el gráfico de números de secuencia (eje y) acorde al tiempo (eje x) de la captura `07-tcp-flow-control.pcap`. Se puede observar en el mismo los eventos de pérdida, la recuperación con la retransmisión y el momento en que el proceso deja de leer datos del buffer de entrada.

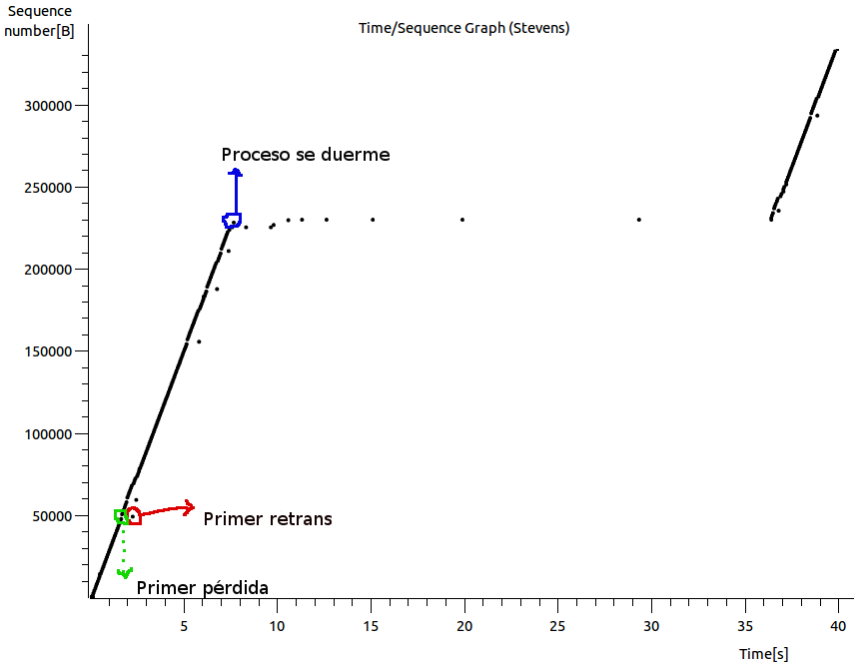

Figura 8: Diagrama de números de secuencia en el tiempo.

En la figura 9 se muestra el segmento de la primer retransmisión, con el ACK duplicado previo de la captura `07-tcp-flow-control.pcap`.

En la figura 10 se muestra el segmento donde la ventana que publica el receptor se "cierra", Win=0 de la captura `07-tcp-flow-control.pcap`.

El ejemplo se puede repetir desactivando el Window Scaling. Ejemplo sin window scaling. En este caso no se envía a dormir al proceso.

```
root@n5:~# sysctl net.ipv4.tcp_window_scaling
net.ipv4.tcp_window_scaling = 1
root@n5:~# sysctl net.ipv4.tcp_window_scaling=0
```

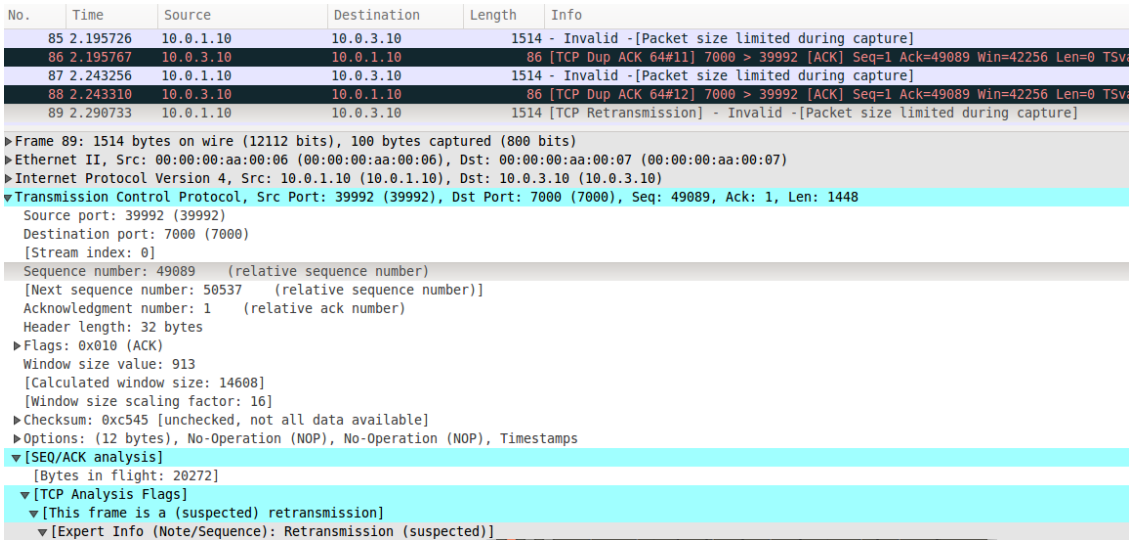

Figura 9: Segmento retransmitido de la captura.

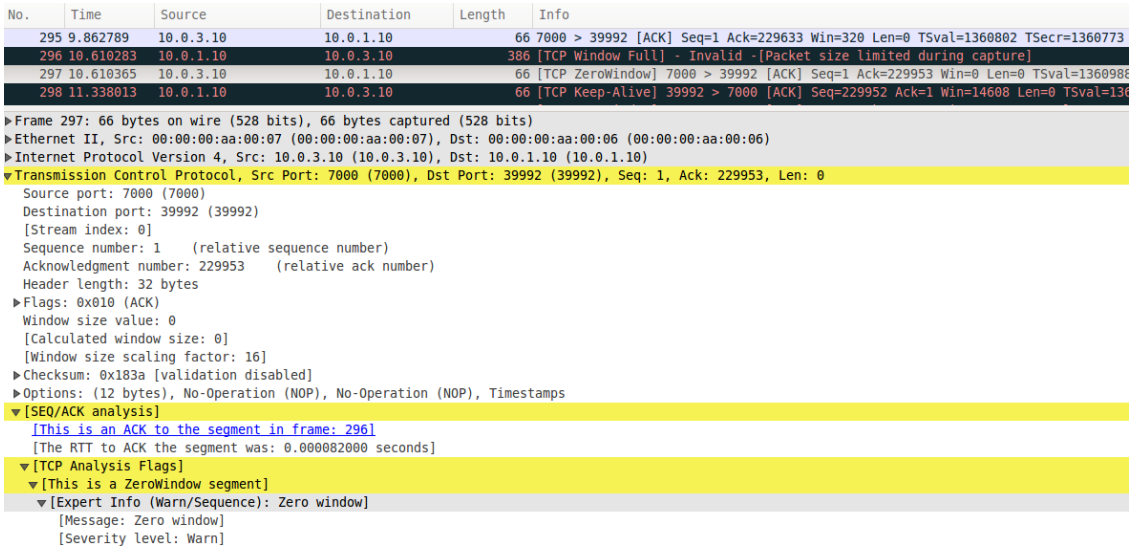

Figura 10: Segmento de la captura con Win=0.

```
net.ipv4.tcp_window_scaling = 0
```

```
root@n5:~# nc -l 7000 | base64
```

```
root@n5:~# nc 10.0.3.10 7000  < /dev/urandom
```

```
root@n5:~# tcpdump -n -i eth0 tcp -s 100 -w 07-tcp-flow-control-no-ws.pcap
```

En este caso no se ve reflejada la diferencia debido a que el ancho de banda es bajo. Si se incrementa el ancho de banda y el delay a: 200Mbps y 150ms en el enlace intermedio, se puede llegar a apreciar la diferencia realizando test con `iperf`.

```
root@n5:~# sysctl net.ipv4.tcp_no_metrics_save=1
net.ipv4.tcp_no_metrics_save = 1
root@n5:/tmp/pycore.58156/n5.conf#

root@n5:~# sysctl net.ipv4.tcp_window_scaling=0
net.ipv4.tcp_window_scaling = 0

root@n5:~# iperf -s
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  4] local 10.0.3.10 port 5001 connected with 10.0.1.10 port 33928
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0-15.0 sec   218 MBytes   122 Mbits/sec

root@n5:~# sysctl net.ipv4.tcp_window_scaling=1
net.ipv4.tcp_window_scaling = 1
root@n5:~# iperf -s
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  4] local 10.0.3.10 port 5001 connected with 10.0.1.10 port 33929
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0-15.0 sec   341 MBytes   190 Mbits/sec
```

7.1. TCP Flow Control y Windowing

En el siguiente ejemplo se muestra como puede sufrir de mala performance una sesión TCP con un incorrecto valor de ventana. Primero se muestra el rendimiento que se obtiene con el valor default y luego con un valor demasiado chico. Primero con la configuración default, con la configuración de la red de la práctica es suficiente.

```
root@n5:~# iperf -s
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
```

```
[  4] local 10.0.3.10 port 5001 connected with 10.0.1.10 port 59404
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0-18.4 sec   512 KBytes  228 Kbits/sec

root@n3:~# iperf -c 10.0.3.10
------------------------------------------------------------
Client connecting to 10.0.3.10, TCP port 5001
TCP window size: 21.0 KByte (default)
------------------------------------------------------------
[  3] local 10.0.1.10 port 59404 connected with 10.0.3.10 port 5001
[ ID] Interval       Transfer     Bandwidth
...
```

Luego con la configuración demasiado chica.

```
root@n5~# tcpdump -n -i eth0 tcp -s 100 -w 07-tcp-win-2k.pcap

root@n5~:# iperf -s -w 2
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 4.00 KByte (WARNING: requested 2.00 KByte)
------------------------------------------------------------
[  4] local 10.0.3.10 port 5001 connected with 10.0.1.10 port 59408
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0-66.8 sec   256 KBytes  31.4 Kbits/sec

root@n3:~# iperf -c 10.0.3.10 -t 20 -i 10
------------------------------------------------------------
Client connecting to 10.0.3.10, TCP port 5001
TCP window size: 21.0 KByte (default)
------------------------------------------------------------
[  3] local 10.0.1.10 port 59408 connected with 10.0.3.10 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec   128 KBytes   105 Kbits/sec
[  3] 10.0-20.0 sec  0.00 Bytes  0.00 bits/sec
[  3] 20.0-30.0 sec  0.00 Bytes  0.00 bits/sec
[  3]  0.0-63.1 sec   256 KBytes  33.2 Kbits/sec
```

Si se observa la captura obtenida se puede ver que los valores de ventana se mantienen chicos, a diferencia de una captura sin modificar este parámetro.

```
root@n5~# tshark -r /tmp/tcp-win-2k.pcap -R "ip.src==10.0.3.10"
                 -T fields -e tcp.window_size | less
1448
1424
```

```
1424
1424
...
```

Para enlaces con un producto *Bandwidth* × *Delay* (BDP) grande puede ser necesario hacer una sintonización de algunas variables del kernel para permitir tamaños de ventana realmente grandes. Los Tamaño default del Rx y Tx buffers se pueden consultar y modificar.

```
root@n5:/tmp# sysctl net.core.rmem_max
net.core.rmem_max = 131071
root@n5:/tmp# sysctl net.core.wmem_max
net.core.wmem_max = 131071
```

Si se desea configurar un tamaño mayor al permitido no se admite.

```
root@n5:/tmp# iperf -s -w 2M
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size:  256 KByte (WARNING: requested 2.00 MByte)
------------------------------------------------------------
^C
```

Reconfigurando las variables del kernel lo permitirá.

```
root@n5:/tmp#  sysctl net.core.wmem_max=4131071
net.core.wmem_max = 4131071
root@n5:/tmp#  sysctl net.core.rmem_max=4131071
net.core.rmem_max = 4131071
```

Existe en el sistema también la posibilidad de realiza un auto-tuning.

```
root@n5:/tmp# sysctl net.ipv4.tcp_moderate_rcvbuf
net.ipv4.tcp_moderate_rcvbuf = 1
```

En el caso de estar habilitado el sistema va manejando la cantidad de memoria de los buffers de acuerdo a la memoria disponible en el sistema, mediante los siguientes arreglos de parámetros.

```
root@n5:/tmp# sysctl net.ipv4.tcp_rmem
net.ipv4.tcp_rmem = 4096 87380 901728
root@n5:/tmp# sysctl net.ipv4.tcp_wmem
net.ipv4.tcp_wmem = 4096 16384 901728
```

En estos arreglos se indica la cantidad mínima, con cuanto comenzar, y la máxima. Se puede ver que se modificó el mínimo, cuando se realizo el cambio de forma absoluta anteriormente.

```
root@n5:/tmp# sysctl net.ipv4.tcp_rmem
net.ipv4.tcp_rmem = 4131071 87380 901728
root@n5:/tmp# sysctl net.ipv4.tcp_wmem
net.ipv4.tcp_wmem = 4131071 16384 901728

root@n5:/tmp# iperf -s -w 2M
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 4.00 MByte (WARNING: requested 2.00 MByte)
------------------------------------------------------------
^C
```

Al cambiar los absolutos la opción más prolija es re-configurar los valores de los mínimos, defaults y máximos para el auto-tuning.

8. Performance TCP

9. Sesión TCP sin SACK y con SACK

```
root@n5:~# sysctl net.ipv4.tcp_window_scaling=1
net.ipv4.tcp_window_scaling = 1

root@n5:~# sysctl net.ipv4.tcp_sack=0
net.ipv4.tcp_sack=0

root@n5:~# iperf -s
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  4] local 10.0.3.10 port 5001 connected with 10.0.1.10 port 35709
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0-18.2 sec   512 KBytes   231 Kbits/sec

root@n3:~# iperf -c 10.0.3.10
------------------------------------------------------------
Client connecting to 10.0.3.10, TCP port 5001
TCP window size: 21.0 KByte (default)
------------------------------------------------------------
[  3] local 10.0.1.10 port 35709 connected with 10.0.3.10 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-17.8 sec   512 KBytes   236 Kbits/sec

root@n5:~# tcpdump -n -i eth0 tcp -s 100 -w 08-tcp-iperf-no-sack.pcap
```

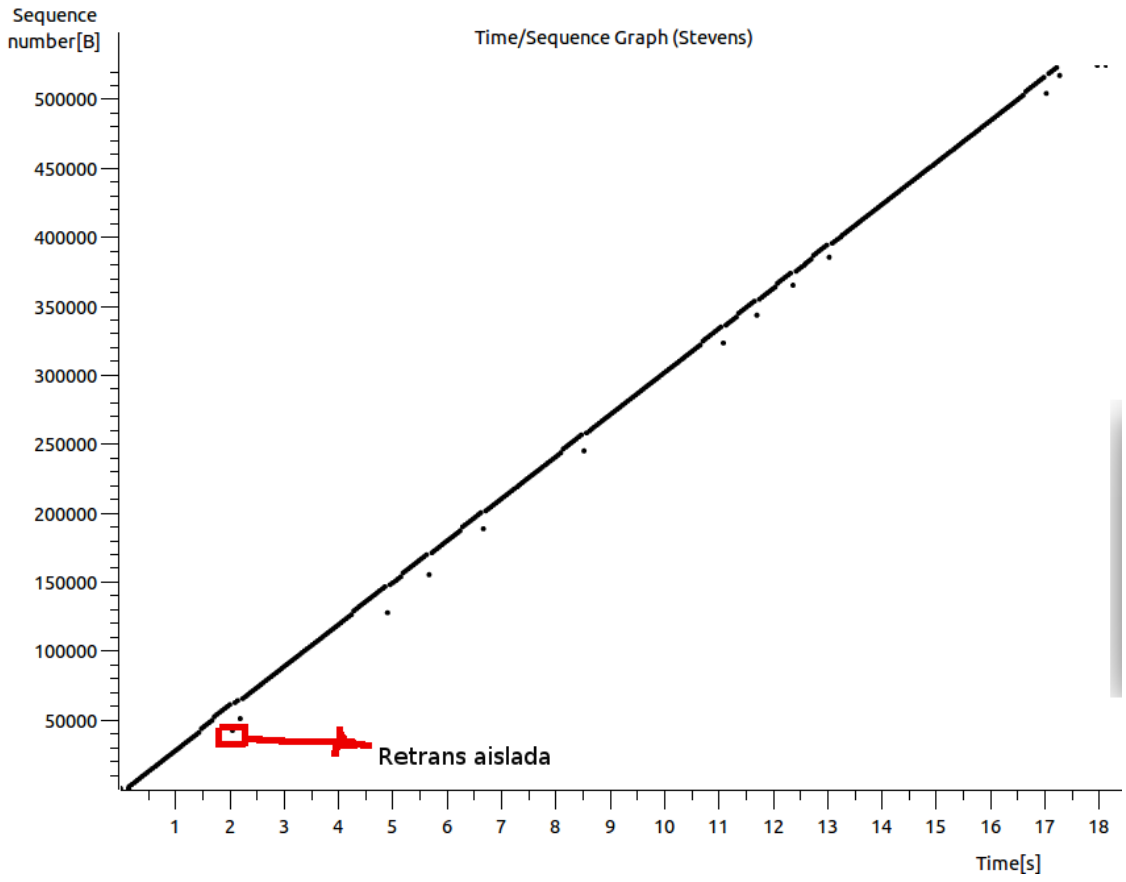

Figura 11: Diagrama de números de secuencia con SACK.

Test con SACK, opción default.

```
root@n5:~# sysctl net.ipv4.tcp_sack=1
net.ipv4.tcp_sack = 1

root@n5:~# iperf -s
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  4] local 10.0.3.10 port 5001 connected with 10.0.1.10 port 35710
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0-17.8 sec   512 KBytes   235 Kbits/sec


root@n3:~# iperf -c 10.0.3.10
```

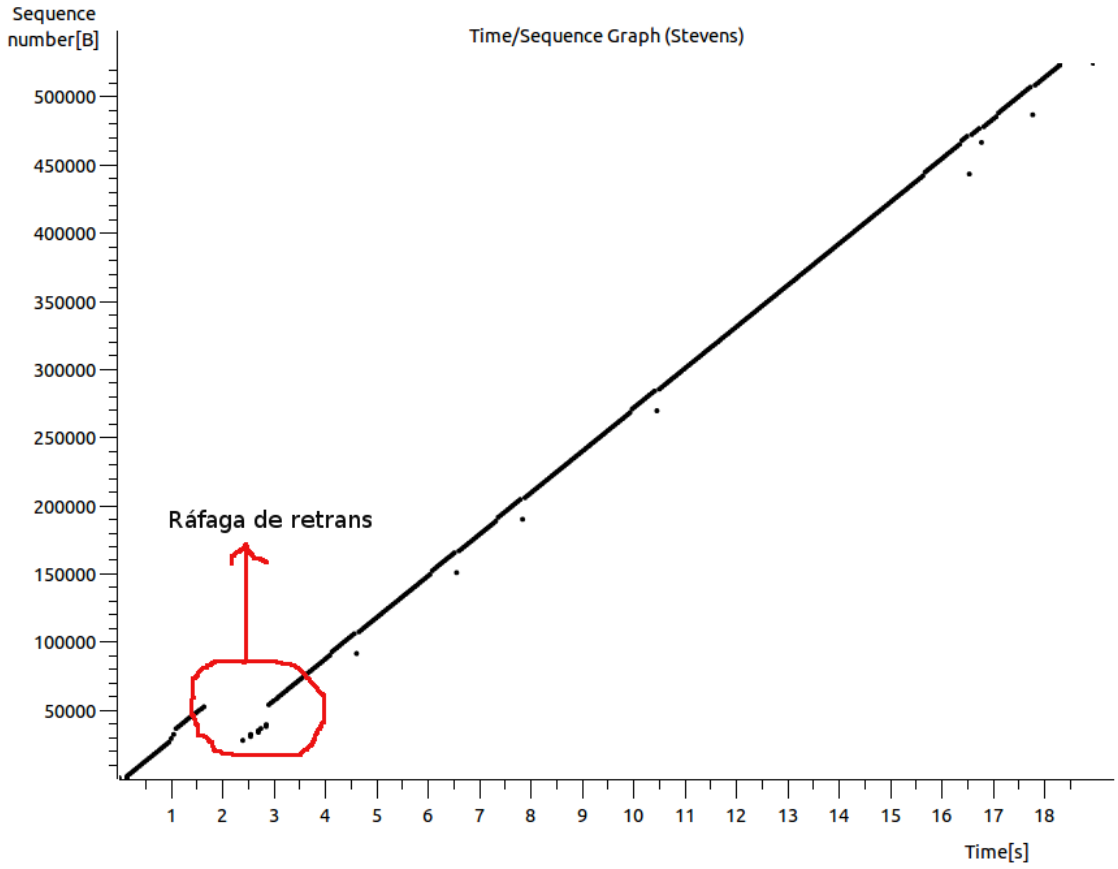

Figura 12: Diagrama de números de secuencia sin SACK.

```
------------------------------------------------------------
Client connecting to 10.0.3.10, TCP port 5001
TCP window size: 21.0 KByte (default)
------------------------------------------------------------
[  3] local 10.0.1.10 port 35710 connected with 10.0.3.10 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-16.3 sec   512 KBytes   257 Kbits/sec

root@n5:~# tcpdump -n -i eth0 tcp -s 100 -w 08-tcp-iperf-sack.pcap
```

En las figuras 11 y 12 se muestran los gráficos de números de secuencia (eje y) acorde al tiempo (eje x) de las capturas con SACK y sin SACK. Se puede observar en el mismo que las retransmisiones puntuales y por bloques.

10. Código TCP de Sockets

Analizar los siguientes códigos socket TCP. Tener en cuenta que están muy simplificados y faltos de todos chequeos.

```
? cat tcp-single-server.c

#include <unistd.h>                     /* close , read  */
#include <sys/types.h>                  /* connect */
#include <sys/socket.h>                 /* socket , connect */
#include <arpa/inet.h>                  /* ip socket , sockaddr_in */
#include <stdlib.h>                     /* exit , malloc , free */
#include <stdio.h>                      /* printf , stderr, perror*/
// #include <errno.h>                      /* errno  */
// #include <getopt.h>                     /* getopt */
// #include <string.h>                     /* strerror_r */

int main(int argc, char *argv[])
{
    int                 peer_sock, sock, len;
    struct sockaddr_in  destaddr;
    struct sockaddr_in  myaddr;
    char                c        =  '?';

    sock = socket(AF_INET, SOCK_STREAM, 0);

    myaddr.sin_family      = AF_INET;
    myaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    myaddr.sin_port        = htons((uint16_t) 18000);

    bind(sock, (struct sockaddr *)&myaddr,
         (socklen_t) sizeof(struct sockaddr_in));

    listen(sock, 5);

    while (1)
      {
        peer_sock = accept(sock, (struct sockaddr*) &destaddr,
                           (socklen_t *) &len);

        read(peer_sock, &c, (size_t) sizeof(c));
        printf("%c\n",c);
      }
}

/// EOF ///
```

```
? cat tcp-single-client.c
```

```
#include <unistd.h>                     /* close , read  */
#include <sys/types.h>                  /* connect */
#include <sys/socket.h>                 /* socket , connect */
#include <arpa/inet.h>                  /* ip socket , sockaddr_in */
#include <stdlib.h>                     /* exit , malloc , free */
#include <stdio.h>                      /* printf , stderr, perror*/
// #include <errno.h>                      /* errno  */
// #include <getopt.h>                     /* getopt */
// #include <string.h>                     /* strerror_r */

int main(int argc, char *argv[])
{
    int                 sock;
    struct sockaddr_in  destaddr;
    char                c        =  '0';
    //struct  hostent *host;

    // Check args
    if (argc < 2)
      {
        printf("usage %s destip, e.g.: %s 127.0.0.1 \n",argv[0],argv[0]);
        exit(-1);
     }

    // Resolve Dest Addr
    //host=gethostbyname(argv[1]);

    // Create Socket
    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0)
      {
        perror("ERROR: at opening tcp socket");
        fflush(stderr);
        return -1;
      }

    // Create Connect Addr
    destaddr.sin_family      = AF_INET;
    //destaddr.sin_port      = htons((uint16_t) port)
    destaddr.sin_port        = htons((uint16_t) 18000);
    destaddr.sin_addr.s_addr = inet_addr(argv[1]);

    // Try to connect
    if ( connect(sock, (struct sockaddr *) &destaddr,
                 (socklen_t) sizeof(struct sockaddr_in)) < 0)
      {
        close(sock);
        perror("ERROR: at connecting to tcp socket");
```

```
        return -1;
      }

    // Send Data (a char)
    write(sock, &c, (size_t) sizeof(c));

    close(sock);

    return (0);
}

/// EOF ///
```

Para probar se deben compilar y generar los dos ejecutables:

```
? gcc -Wall -o b tcp-single-server.c
? gcc -Wall -o a tcp-single-client.c

? b &
...

? ./a 127.0.0.1
```

Otra forma de ver las llamadas de sockets en funcionamiento sin necesidad de escribir código es usando la herramienta `strace(1)` que es un traceador de las llamadas al sistema. Por ejemplo se puede ejecutar:

```
server? strace -e trace=network nc -n -l 8000
socket(PF_INET, SOCK_STREAM, IPPROTO_TCP) = 3
setsockopt(3, SOL_SOCKET, SO_REUSEADDR, [1], 4) = 0
bind(3, {sa_family=AF_INET, sin_port=htons(8000),
     sin_addr=inet_addr("0.0.0.0")}, 16) = 0
listen(3, 1)                            = 0
accept(3,...


client?  strace -e trace=network nc -n 10.0.3.10 8000
socket(PF_INET, SOCK_STREAM, IPPROTO_TCP) = 3
connect(3, {sa_family=AF_INET, sin_port=htons(8000),
  sin_addr=inet_addr("10.0.3.10")}, 16) = -1
  EINPROGRESS (Operation now in progress)
getsockopt(3, SOL_SOCKET, SO_ERROR, [0], [4]) = 0
HOLA
CTRL+C

server?
...
accept(3, {sa_family=AF_INET, sin_port=htons(56592),
```

```
  sin_addr=inet_addr("10.0.1.10")}, [16]) = 4
HOLA
shutdown(4, 0 /* receive */)            = 0
```

Para ver cómo se envían y reciben datos se puede agregar otras opciones a la herramienta strace, como `-e trace=desc`. En este caso se verían las llamadas a read y write, como muchas otras.

```
read(0, "HOLA\n", 1024)                 = 5
write(3, "HOLA\n", 5)                   = 5
```

Referencias

[StevI] TCP/IP Illustrated, Volume 1: The Protocols, Addison-Wesley, 1994. W. Richard Stevens.

[Siever] Linux in a Nutshell, Fourth Edition June, 2003. O'Reilly. Ellen Siever, Stephen Figgins, Aaron Weber.

[RFC-793] http://www.rfc-editor.org/rfc/rfc793.txt. TRANSMISSION CONTROL PROTOCOL. (J. Postel 1981 ISI).

[RFC-2018] http://tools.ietf.org/html/rfc2018. TCP Selective Acknowledgment Options.

[RFC-7323] http://www.rfc-editor.org/rfc/rfc7323.txt. TCP Extensions for High Performance.

[TCPADM] TCP/IP Network Administration By Craig Hunt. O'Reilly.

[LINUXIP] http://www.linux-ip.net/. Guide to IP Layer Network Administrator with Linux.

[COM05] Ethereal, Wireshark. Autor original Gerald Combs, 2005.
http://www.ethereal.com/.
http://www.wireshark.org/.

[SYSCTL] Linux Kernel Net parameters. `https://www.kernel.org/doc/Documentation/networking/ip-sysctl.txt`.

UNIVERSIDAD NACIONAL DE LA PLATA

Índice

1. TCP (Transport Control Protocol) 2

2. Ejemplos Sencillos con TCP 3
2.1. Verificar los Procesos TCP . . . 3
2.2. Levantar Proceso TCP . . . 4

3. Ejemplo de TCP múltiple 8

4. Ejemplo de Análisis de Estados TCP 9
4.1. Establecimiento de Conexión . . . 9
4.2. Cierre de Conexión . . . 12
4.3. Half-Close (Cierre a medias) de Conexión . . . 19

5. Ejemplo de Análisis de Flags TCP 19

6. Opciones TCP 21
6.1. MSS (Maximum Segment Size) . . . 22
6.2. Timestamps . . . 23
6.3. SACK(Selective ACK) . . . 23
6.4. Window Scaling . . . 23

7. Control de Errores y de Flujo (Error/Flow Control) 23
7.1. TCP Flow Control y Windowing . . . 26

8. Performance TCP 29

9. Sesión TCP sin SACK y con SACK 29

10. Código TCP de Sockets 32

Índice de figuras

1. Diagrama de Segmento TCP. . . . 2
2. Topología de test uno. . . . 3
3. Topología de test dos. . . . 3
4. Segmento TCP de la captura inicial. . . . 5
5. Diagrama de flujo de la sesión TCP de la captura. . . . 7
6. Máquina de estados reducida de TCP. . . . 10
7. Flujo de segmentos de la conexión fallida. . . . 11
8. Diagrama de números de secuencia en el tiempo. . . . 24
9. Segmento retransmitido de la captura. . . . 25
10. Segmento de la captura con Win=0. . . . 25
11. Diagrama de números de secuencia con SACK. . . . 30

12. Diagrama de números de secuencia sin SACK. 31