M1 Informatique # Protocoles réseaux

TD n.7

Traces de paquets

Analyse de traces

The tcpdump **Program**

- ☑ The **tcpdump** program was written by Van Jacobson, Craig Leres, and Steven McCanne, all of Lawrence Berkeley Laboratory, University of California, Berkeley.
- ☑ **tcpdump** operates by putting the network interface card into **promiscuous mode** so that every packet going across the wire is captured. Normally interface cards for media such as Ethernet only capture link level frames addressed to the particular interface or to the broadcast address
- ☑ The output produced by **tcpdump** is "raw."
- ☑ First, it always outputs the name of the network interface on which it is listening.

```
C:\Users\lenny\Downloads\tcpdump.exe: verbose output suppressed, use -v or -vv for full protocol decode
listening on \Device\{....}, link-type EN10MB (Ethernet), capture size 262144 bytes
```

- ☑ Next, the timestamp output by **tcpdump** is of the form **09:11:22.642008** on a system with microsecond resolution, or **09:11:22.64** on a system with only 10-ms clock resolution.
- ☑ **tcpdump** always prints the name of the sending host, then a greater than sign, then the name of the destination host.

Source : TCP/IP illustrated . Volume 1. The protocols Stevens, W. Richard 1994

> man tcpdump

TCPDUMP(8) **System Manager's Manual**

2 February 2017

NAME : tcpdump - dump traffic on a network

DESCRIPTION :

- ☑ **Tcpdump prints out a description of the contents of packets** on a network interface that match the boolean expression.
- ☑ **The description is preceded by a time stamp**, printed, by default, as hours, minutes, seconds, and fractions of a second since midnight.
- ☑ It can also be run with **the -w flag**, which causes it to save the packet data to a file for later analysis, and/or with **the -r flag**, which causes it to read from a saved packet file rather than to read packets from a network interface.

- ☑ Tcpdump will, if not run with **the -c flag**, continue capturing packets until it is interrupted by a SIGINT signal (generated, for example, by typing your interrupt character, typically control-C) or a SIGTERM signal (typically generated with the kill(1) command);

- ☑ **When tcpdump finishes capturing packets**, it will report counts of:

    - packets **captured** (this is the number of packets that tcpdump has received and processed);
    - packets **received by filter** the meaning of this depends on the OS on which you're running tcpdump, and possibly on the way the OS was configured.
    - packets **dropped by kernel** (this is the number of packets that were dropped, due to a lack of buffer space, by the packet capture mechanism in the OS on which tcpdump is running, if the OS reports that information to applications; if not, it will be reported as 0).

OPTIONS :

- ☑ **-D :** Print the list of the network interfaces available on the system and on which tcpdump can capture packets. For each network interface, a number and an interface name, possibly followed by a text description of the interface, is printed. The interface name or the number can be supplied to the -i flag to specify an interface on which to capture.

- ☑ **-i interface** : Listen on interface. If unspecified, tcpdump searches the system interface list for the lowest numbered, configured up interface (excluding loopback).

- ☑ **-n** : Don't convert addresses (i.e., host addresses, port numbers, etc.) to names.

- ☑ **-r file** : Read packets from file (which was created with the -w option or by other tools that write pcap or pcapng files). Standard input is used if file is ``-''.

- ☑ **-e** : Print the link-level header on each dump line. This can be used, for example, to print MAC layer addresses for protocols such as Ethernet and IEEE 802.11.

- ☑ **-v** : When parsing and printing, produce (slightly more) verbose output. For example, the time to live, identification, total length and options in an IP packet are printed. Also enables additional packet integrity checks such as verifying the IP and ICMP header checksum.

OUTPUT FORMAT :

The output of tcpdump is protocol dependent. The following gives a brief description and examples of most of the formats.

- ☑ **Timestamps :** By default, all output lines are preceded by a timestamp. The timestamp is the current clock time in the form **hh:mm:ss.frac** and is as accurate as the kernel's clock. The timestamp reflects the time the kernel applied a time stamp to the packet. No attempt is made to account for the time lag between when the network interface finished receiving the packet from the network and when the kernel applied a time stamp to the packet; that time lag could include a delay between the time when the network interface finished receiving a packet from the network and the time when an interrupt was delivered to the kernel to get it to read the packet and a delay between the time when the kernel serviced the `new packet' interrupt and the time when it applied a time stamp to the packet.

- ☑ **Link Level Headers :** If the '-e' option is given, the link level header is printed out.

ARP/RARP Packets

☑ Arp/rarp output shows the type of request and its arguments. The format is intended to be self explanatory.

☑ Here is a short sample taken from the start of an `rlogin' from host rtsg to host csam:

```
arp who-has csam tell rtsg
arp reply csam is-at CSAM
```

☑ The first line says that rtsg sent an arp packet asking for the Ethernet address of internet host csam.

☑ Csam replies with its Ethernet address (in this example, Ethernet addresses are in caps and internet addresses in lower case).

☑ This would look less redundant if we had done tcpdump -n:

```
arp who-has 128.3.254.6 tell 128.3.254.68
arp reply 128.3.254.6 is-at 02:07:01:00:01:c4
```

☑ If we had done tcpdump -e, the fact that the first packet is broadcast and the second is point-to-point would be visible:

```
RTSG Broadcast 0806 64: arp who-has csam tell rtsg
CSAM RTSG 0806 64: arp reply csam is-at CSAM
```

☑ For the first packet this says the Ethernet source address is RTSG, the destination is the Ethernet broadcast address, the type field contained hex 0806 (type ETHER_ARP) and the total length was 64 bytes.

IPv4 Packets

☑ If the link-layer header is not being printed, for IPv4 packets, IP is printed after the time stamp.

☑ If the -v flag is specified, information from the IPv4 header is shown in parentheses after the IP or the link-layer header. The general format of this information is:

```
tos tos, ttl ttl, id id, offset offset, flags [flags], proto proto, length length, options (options)
```

- tos is the type of service field; if the ECN bits are non-zero, those are reported as ECT(1), ECT(0), or CE.
- ttl is the time-to-live; it is not reported if it is zero.
- id is the IP identification field.
- offset is the fragment offset field; it is printed whether this is part of a fragmented datagram or not.

- **flags** are the MF and DF flags; + is reported if MF is set, and DFP is reported if F is set. If neither are set, . is reported.
- **proto** is the protocol ID field.
- **length** is the total length field.
- **options** are the IP options, if any.

Next, for TCP and UDP packets, the source and destination IP addresses and TCP or UDP ports, with a dot between each IP address and its corresponding port, will be printed, with a > separating the source and destination.

- ☑ **For other protocols,** the addresses will be printed, with a > separating the source and destination.
- ☑ **Higher level protocol information**, if any, will be printed after that.
- ☑ **For fragmented IP datagrams**, the first fragment contains the higher level protocol header; fragments after the first contain no higher level protocol header.
- ☑ **Fragmentation information will be printed only with the -v flag**, in the IP header information, as described above.

TCP Packets

(N.B. : The following description assumes familiarity with the TCP protocol described in RFC-793. If you are not familiar with the protocol, this description will not be of much use to you.)

- ☑ **The general format of a TCP protocol line is:**

src > dst: Flags [tcpflags], seq data-seqno, ack ackno, win window, urg urgent, options [opts], length len

- **Src** and **dst** are the source and destination IP addresses and ports.
- **Tcpflags** are some combination of
  - S (SYN),
  - F (FIN),
  - P (PUSH),
  - R (RST),
  - U (URG),
  - W (ECN CWR),
  - E (ECN-Echo)
  - `.' (ACK)
  - `none' if no flags are set.
- **Data-seqno** describes the portion of sequence space covered by the data in this packet (see example below).
- **Ackno** is sequence number of the next data expected the other direction on this connection.

- **Window** is the number of bytes of receive buffer space available the other direction on this connection.
- **Urg** indicates there is `urgent' data in the packet.
- **Opts** are TCP options (e.g., mss 1024).
- **Len** is the length of payload data.

☑ Iptype, Src, dst, and flags are always present. The other fields depend on the contents of the packet's TCP protocol header and are output only if appropriate.

☑ Here is the opening portion of an rlogin from host rtsg to host csam.

```
IP rtsg.1023 > csam.login: Flags [S], seq 768512:768512, win 4096, opts [mss 1024]
IP csam.login > rtsg.1023: Flags [S.], seq, 947648:947648, ack 768513, win 4096, opts [mss 1024]
IP rtsg.1023 > csam.login: Flags [.], ack 1, win 4096
IP rtsg.1023 > csam.login: Flags [P.], seq 1:2, ack 1, win 4096, length 1
IP csam.login > rtsg.1023: Flags [.], ack 2, win 4096
IP rtsg.1023 > csam.login: Flags [P.], seq 2:21, ack 1, win 4096, length 19
IP csam.login > rtsg.1023: Flags [P.], seq 1:2, ack 21, win 4077, length 1
IP csam.login > rtsg.1023: Flags [P.], seq 2:3, ack 21, win 4077, urg 1, length 1
IP csam.login > rtsg.1023: Flags [P.], seq 3:4, ack 21, win 4077, urg 1, length 1
```

- The first line says that TCP port 1023 on rtsg sent a packet to port login on csam.
  - The S indicates that the SYN flag was set.
  - The packet sequence number was 768512 and it contained no data. (The notation is`first:last' which means `sequence numbers first up to but not including last.)
  - There was no piggy-backed ack, the available receive window was 4096 bytes and there was a max-segment-size option requesting an mss of 1024 bytes.

- Csam replies with a similar packet except it includes a piggy-backed ack for rtsg's SYN.
- Rtsg then acks csam's SYN. The `.' means the ACK flag was set.
  - The packet contained no data so there is no data sequence number or length.
  - Note that the ack sequence number is a small integer (1).
  - The first time tcpdump sees a TCP `conversation', it prints the sequence number from the packet.
  - On subsequent packets of the conversation,the difference between the current packet's sequence number and this initial sequence number is printed.
  - This means that sequence numbers after the first can be interpreted as relative byte positions in the conversation's data stream (with the first data byte each direction being `1').

- `-S' will override this feature, causing the original sequence numbers to be output.

- On the 6th line, rtsg sends csam 19 bytes of data (bytes 2 through 20 in the rtsg → csam side of the conversation).
  - The PUSH flag is set in the packet.

- On the 7th line, csam says it's received data sent by rtsg up to but not including byte 21.
  - Most of this data is apparently sitting in the socket buffer since csam's receive window has gotten 19 bytes smaller.
  - Csam also sends one byte of data to rtsg in this packet.
- On the 8th and 9th lines, csam sends two bytes of urgent, pushed data to rtsg.

☑ If the snapshot was small enough that tcpdump didn't capture the full TCP header, it interprets as much of the header as it can and then reports ``[|tcp]'' to indicate the remainder could not be interpreted.

☑ If the header contains a bogus option (one with a length that's either too small or beyond the end of the header), tcpdump reports it as ``[bad opt]'' and does not interpret any further options (since it's impossible to tell where they start).

☑ If the header length indicates options are present but the IP datagram length is not long enough for the options to actually be there, tcpdump reports it as ``[bad hdr length]''.

☑ **There are 8 bits in the control bits section of the TCP header:**

CWR | ECE | URG | ACK | PSH | RST | SYN | FIN

➢ **Recall that TCP uses a 3-way handshake protocol when it initializes a new connection**
  - the connection sequence with regard to the TCP control bits is

1) Caller sends SYN

2) Recipient responds with SYN, ACK

3) Caller sends ACK

UDP Packets

➢ UDP format is illustrated by this rwho packet:

actinide.who > broadcast.who: udp 84

- This says that port who on host actinide sent a udp datagram to port who on host broadcast, the Internet broadcast address.
- The packet contained 84 bytes of user data.

- Some UDP services are recognized (from the source or destination port number) and the higher level protocol information printed.
- In particular, Domain Name service requests (RFC-1034/1035) and Sun RPC calls (RFC-1050) to NFS.

UDP Name Server Requests

(N.B. : The following description assumes familiarity with the Domain Service protocol described in RFC-1035. If you are not familiar with the protocol, the following description will appear to be written in greek.)

- Name server requests are formatted as

```
src > dst: id op? flags qtype qclass name (len)
```

```
h2opolo.1538 > helios.domain: 3+ A? ucbvax.berkeley.edu. (37)
```

- Host h2opolo asked the domain server on helios for an address record (qtype=A) associated with the name ucb-vax.berkeley.edu.
- The query id was `3'.
- The `+' indicates the recursion desired flag was set.
- The query length was 37 bytes, not including the UDP and IP protocol headers.
- The query operation was the normal one, Query, so the op field was omitted.
- If the op had been anything else, it would have been printed between the `3' and the `+'.
- Similarly, the qclass was the normal one, C_IN, and omitted.
- Any other qclass would have been printed immediately after the `A'.

- A few anomalies are checked and may result in extra fields enclosed in square brackets:
  - If a query contains an answer, authority records or additional records section, ancount, nscount, or arcount are printed as `[na]', `[nn]' or `[nau]' where n is the appropriate count.
  - If any of the response bits are set (AA, RA or rcode) or any of the `must be zero' bits are set in bytes two and three, `[b2&3=x]' is printed, where x is the hex value of header bytes two and three.

UDP Name Server Responses

- ☑ Name server responses are formatted as

```
src > dst:  id op rcode flags a/n/au type class data (len)
```

```
helios.domain > h2opolo.1538: 3 3/3/7 A 128.32.137.3 (273)

helios.domain > h2opolo.1537: 2 NXDomain* 0/1/0 (97)
```

- ☑ **In the first example**, helios responds to query id 3 from h2opolo with 3 answer records, 3 name server records and 7 additional records.

- The first answer record is type A (address) and its data is internet address 128.32.137.3.
- The total size of the response was 273 bytes, excluding UDP and IP headers.
- The op (Query) and response code (NoError) were omitted, as was the class (C_IN) of the A record.

☑ **In the second example**, helios responds to query 2 with a response code of non-existent domain (NXDomain) with no answers, one name server and no authority records.

- The `*' indicates that the authoritative answer bit was set.
- Since there were no answers, no type, class or data were printed.
- Other flag characters that might appear are `-' (recursion available, RA, not set) and `|' (truncated message, TC, set).
- If the `question' section doesn't contain exactly one entry, `[nq]' is printed.

Source : man tcpdump

Deux outils de captures :

un qui s'appelle **tcpdump** en mode texte et un qui s'appelle **Wireshark**. Wireshark est un outil graphique et plus moderne.

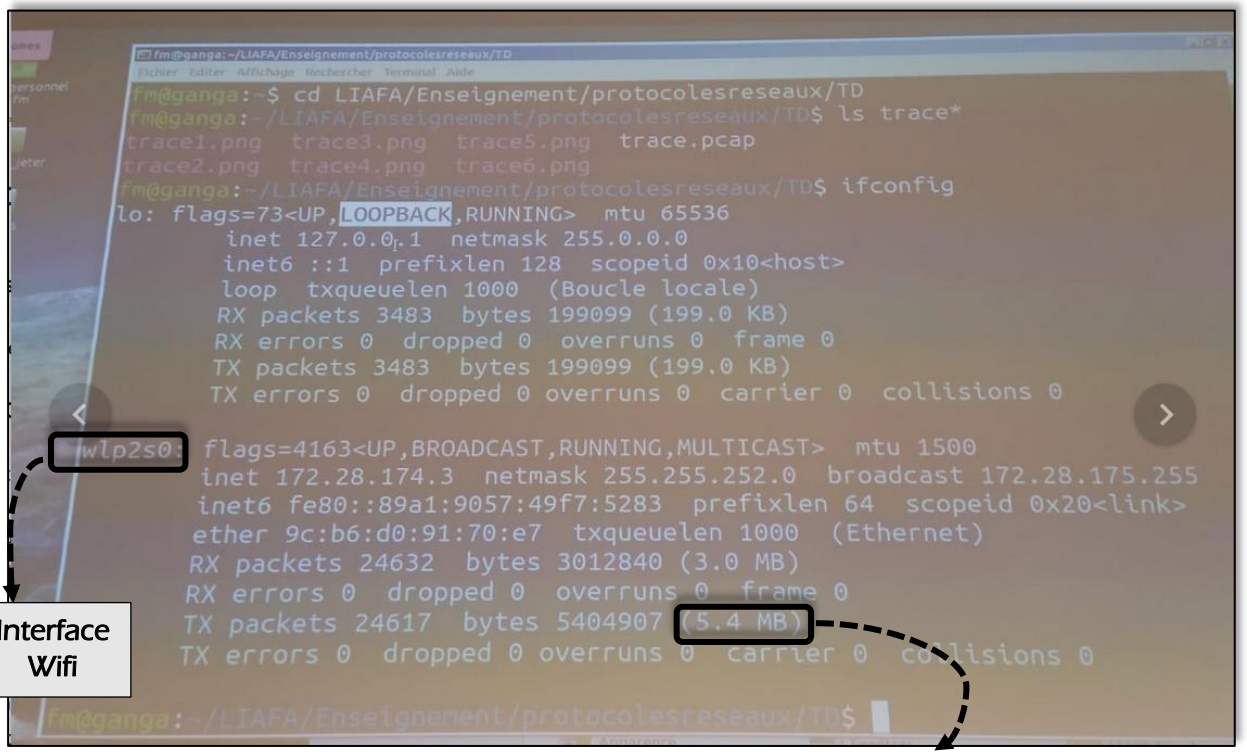
1#

Identifiez le nom de l'interface réseau par laquelle vous êtes connectés à l'Internet.

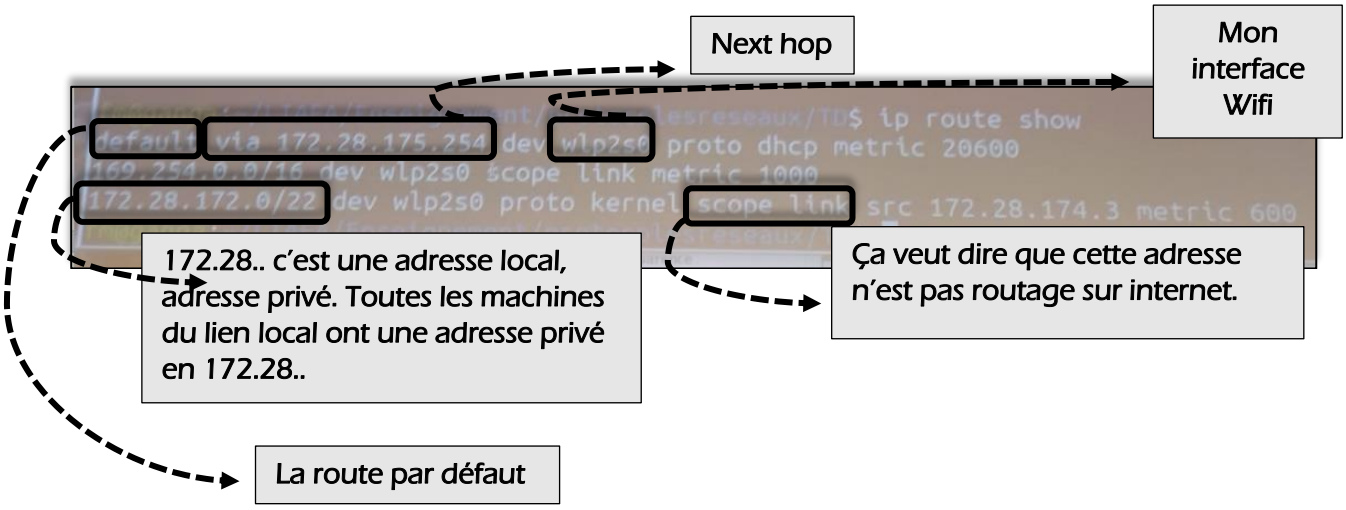
Sous Linux, vous pouvez par exemple utiliser la commande « *ip route show* » pour déterminer l'interface par la quelle passe la route par défaut.

➢ `ifconfig`

On voit les interfaces actives :



```
➢  ip route show
```

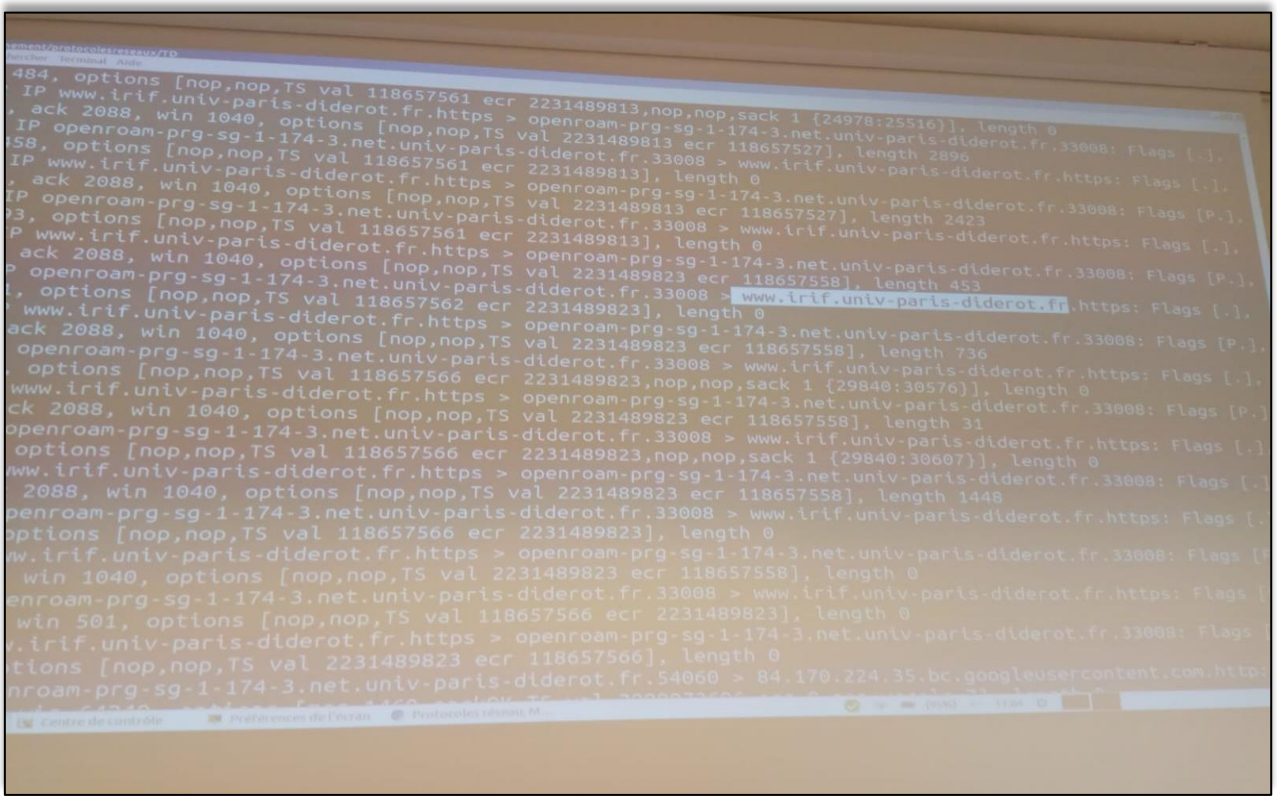



```
➢  tcpdump -n -i wlp2s0
```

- ☑ Cette commande va essayer de capturer les paquets qui sorte, cela est possible que pour l'administrateur et il faut donc ajouter `sudo` avant la commande.
- ☑ On va retirer `-n` de la commande. Maintenant, je peux voir les noms des machines
- ☑ CTRL + C : pour arrêter « tcpdump ».

La vitesse à laquelle les paques se charges augmente lorsque je vais taper une adresse sur le navigateur. (le pbm est qu'il y a un tas de paquets parasites donc faut apprendre à utiliser les filtres).

Par exemple, si on navigue vers la page Web du cours on va voir :



Wireshark est un outil graphique et plus moderne, je vois tous les interfaces de la machine, on va lancer la capture, pour l'instant il se passe pas grand-chose, mais si on recharge la page du cours...

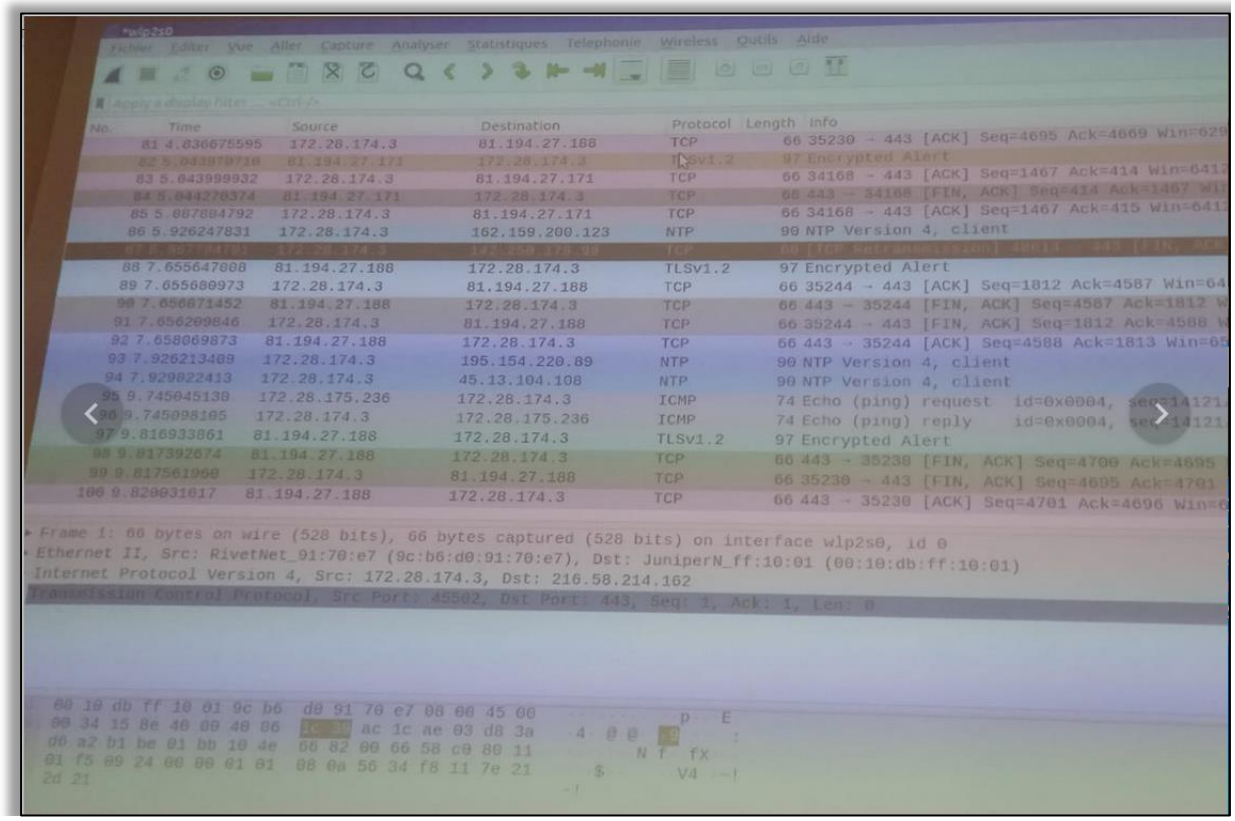
On voit tous les paquets qui sont passé, on a, par exemple, le premier paquet reçu avec le temp écouler depuis les deux derniers, le nombre de paquets capturées en total (500 paquets).

On voie également l'IP source et l'IP destination, les différents protocoles de différentes couche comme TCP de la couche 4 et TLS qui est d'une couche plus élevé.

Pour voir plus de détails on peut voir les détails du paquet couche par couche en cliquant sur le paquet.

Par ailleurs, on peut voir les adresse Mac, les informations de couche 4, on peut voir le numéro de port 123, 123 c'est le port de UDP.

➢ **wireshark**



Donc, ce premier exercice est pour présenter les outils d'analyses de protocoles réseaux.

Le fichier trace.pcapet

trace.pcap

Source : https://www.irif.fr/~jch/enseignement/reseaux/trace.pcap

```
reading from file ../trace.pcap, link-type EN10MB (Ethernet)

10:59:47.105310 IP 192.168.3.195.39965 > 138.231.176.10.80: Flags [S], seq 1793060694, win 14600, options [mss
1460,sackOK,TS val 581341 ecr 0,nop,wscale 7], length 0

10:59:47.113894 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags [S.], seq 3675933190, ack 1793060695, win 5792,
options [mss 1460,sackOK,TS val 2699823136 ecr 581341,nop,wscale 6], length 0

10:59:47.113990 IP 192.168.3.195.39965 > 138.231.176.10.80: Flags [.], ack 1, win 115, options [nop,nop,TS val 581343
ecr 2699823136], length 0

10:59:47.114230 IP 192.168.3.195.39965 > 138.231.176.10.80: Flags [P.], seq 1:118, ack 1, win 115, options [nop,nop,TS
val 581343 ecr 2699823136], length 117: HTTP: GET / HTTP/1.1

10:59:47.125640 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags [.], ack 118, win 91, options [nop,nop,TS val
2699823138 ecr 581343], length 0

10:59:47.125717 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags [.], seq 1:1449, ack 118, win 91, options [nop,nop,TS
val 2699823138 ecr 581343], length 1448: HTTP: HTTP/1.1 200 OK

10:59:47.125753 IP 192.168.3.195.39965 > 138.231.176.10.80: Flags [.], ack 1449, win 137, options [nop,nop,TS val 581346
ecr 2699823138], length 0

10:59:47.128088 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags [.], seq 1449:2897, ack 118, win 91, options
[nop,nop,TS val 2699823138 ecr 581343], length 1448: HTTP

10:59:47.128135 IP 192.168.3.195.39965 > 138.231.176.10.80: Flags [.], ack 2897, win 160, options [nop,nop,TS val 581346
ecr 2699823138], length 0

10:59:47.137330 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags [.], seq 2897:4345, ack 118, win 91, options
[nop,nop,TS val 2699823141 ecr 581346], length 1448: HTTP

10:59:47.137394 IP 192.168.3.195.39965 > 138.231.176.10.80: Flags [.], ack 4345, win 182, options [nop,nop,TS val 581349
ecr 2699823141], length 0

10:59:47.137444 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags [.], seq 4345:5793, ack 118, win 91, options
[nop,nop,TS val 2699823141 ecr 581346], length 1448: HTTP

10:59:47.137459 IP 192.168.3.195.39965 > 138.231.176.10.80: Flags [.], ack 5793, win 205, options [nop,nop,TS val 581349
ecr 2699823141], length 0

10:59:47.139685 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags [.], seq 5793:7241, ack 118, win 91, options
[nop,nop,TS val 2699823141 ecr 581346], length 1448: HTTP

10:59:47.139748 IP 192.168.3.195.39965 > 138.231.176.10.80: Flags [.], ack 7241, win 228, options [nop,nop,TS val 581349
ecr 2699823141], length 0

10:59:47.139812 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags [.], seq 7241:8689, ack 118, win 91, options
[nop,nop,TS val 2699823141 ecr 581346], length 1448: HTTP

10:59:47.139827 IP 192.168.3.195.39965 > 138.231.176.10.80: Flags [.], ack 8689, win 250, options [nop,nop,TS val 581349
ecr 2699823141], length 0

10:59:47.149209 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags [.], seq 8689:10137, ack 118, win 91, options
[nop,nop,TS val 2699823144 ecr 581349], length 1448: HTTP

10:59:47.149295 IP 192.168.3.195.39965 > 138.231.176.10.80: Flags [.], ack 10137, win 273, options [nop,nop,TS val
581352 ecr 2699823144], length 0

10:59:47.149355 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags [.], seq 10137:11585, ack 118, win 91, options
[nop,nop,TS val 2699823144 ecr 581349], length 1448: HTTP

10:59:47.149371 IP 192.168.3.195.39965 > 138.231.176.10.80: Flags [.], ack 11585, win 296, options [nop,nop,TS val
581352 ecr 2699823144], length 0
```

10:59:47.149400 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags [.], seq 11585:13033, ack 118, win 91, options [nop,nop,TS val 2699823144 ecr 581349], length 1448: HTTP

10:59:47.149437 IP 192.168.3.195.39965 > 138.231.176.10.80: Flags [.], ack 13033, win 318, options [nop,nop,TS val 581352 ecr 2699823144], length 0

10:59:47.151742 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags [.], seq 13033:14481, ack 118, win 91, options [nop,nop,TS val 2699823144 ecr 581349], length 1448: HTTP

10:59:47.151804 IP 192.168.3.195.39965 > 138.231.176.10.80: Flags [.], ack 14481, win 331, options [nop,nop,TS val 581352 ecr 2699823144], length 0

10:59:47.151862 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags [.], seq 14481:15929, ack 118, win 91, options [nop,nop,TS val 2699823144 ecr 581349], length 1448: HTTP

10:59:47.151878 IP 192.168.3.195.39965 > 138.231.176.10.80: Flags [.], ack 15929, win 320, options [nop,nop,TS val 581352 ecr 2699823144], length 0

10:59:47.154232 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags [.], seq 15929:17377, ack 118, win 91, options [nop,nop,TS val 2699823144 ecr 581349], length 1448: HTTP

10:59:47.154293 IP 192.168.3.195.39965 > 138.231.176.10.80: Flags [.], ack 17377, win 331, options [nop,nop,TS val 581353 ecr 2699823144], length 0

10:59:47.160807 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags [.], seq 17377:18825, ack 118, win 91, options [nop,nop,TS val 2699823147 ecr 581352], length 1448: HTTP

10:59:47.160864 IP 192.168.3.195.39965 > 138.231.176.10.80: Flags [.], ack 18825, win 331, options [nop,nop,TS val 581354 ecr 2699823147], length 0

10:59:47.160918 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags [.], seq 18825:20273, ack 118, win 91, options [nop,nop,TS val 2699823147 ecr 581352], length 1448: HTTP

10:59:47.160932 IP 192.168.3.195.39965 > 138.231.176.10.80: Flags [.], ack 20273, win 320, options [nop,nop,TS val 581354 ecr 2699823147], length 0

10:59:47.160960 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags [.], seq 20273:21721, ack 118, win 91, options [nop,nop,TS val 2699823147 ecr 581352], length 1448: HTTP

10:59:47.163259 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags [.], seq 21721:23169, ack 118, win 91, options [nop,nop,TS val 2699823147 ecr 581352], length 1448: HTTP

10:59:47.163317 IP 192.168.3.195.39965 > 138.231.176.10.80: Flags [.], ack 23169, win 331, options [nop,nop,TS val 581355 ecr 2699823147], length 0

10:59:47.163362 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags [.], seq 23169:24617, ack 118, win 91, options [nop,nop,TS val 2699823147 ecr 581352], length 1448: HTTP

10:59:47.163385 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags [.], seq 24617:26065, ack 118, win 91, options [nop,nop,TS val 2699823147 ecr 581352], length 1448: HTTP

10:59:47.163438 IP 192.168.3.195.39965 > 138.231.176.10.80: Flags [.], ack 26065, win 331, options [nop,nop,TS val 581355 ecr 2699823147], length 0

10:59:47.165913 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags [.], seq 26065:27513, ack 118, win 91, options [nop,nop,TS val 2699823147 ecr 581352], length 1448: HTTP

10:59:47.165960 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags [.], seq 27513:28961, ack 118, win 91, options [nop,nop,TS val 2699823147 ecr 581352], length 1448: HTTP

10:59:47.166035 IP 192.168.3.195.39965 > 138.231.176.10.80: Flags [.], ack 28961, win 331, options [nop,nop,TS val 581356 ecr 2699823147], length 0

10:59:47.166207 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags [.], seq 28961:30409, ack 118, win 91, options [nop,nop,TS val 2699823148 ecr 581353], length 1448: HTTP

10:59:47.168549 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags [.], seq 30409:31857, ack 118, win 91, options [nop,nop,TS val 2699823148 ecr 581353], length 1448: HTTP

10:59:47.168585 IP 192.168.3.195.39965 > 138.231.176.10.80: Flags [.], ack 31857, win 331, options [nop,nop,TS val 581356 ecr 2699823148], length 0

10:59:47.170831 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags [.], seq 31857:33305, ack 118, win 91, options [nop,nop,TS val 2699823150 ecr 581354], length 1448: HTTP

10:59:47.173921 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags [.], seq 33305:34753, ack 118, win 91, options [nop,nop,TS val 2699823150 ecr 581354], length 1448: HTTP

10:59:47.173987 IP 192.168.3.195.39965 > 138.231.176.10.80: Flags [.], ack 34753, win 331, options [nop,nop,TS val 581358 ecr 2699823150], length 0

10:59:47.174039 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags [.], seq 34753:36201, ack 118, win 91, options [nop,nop,TS val 2699823150 ecr 581354], length 1448: HTTP

10:59:47.174067 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags [.], seq 36201:37649, ack 118, win 91, options [nop,nop,TS val 2699823150 ecr 581354], length 1448: HTTP

10:59:47.174132 IP 192.168.3.195.39965 > 138.231.176.10.80: Flags [.], ack 37649, win 331, options [nop,nop,TS val 581358 ecr 2699823150], length 0

10:59:47.176431 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags [.], seq 37649:39097, ack 118, win 91, options [nop,nop,TS val 2699823150 ecr 581354], length 1448: HTTP

10:59:47.176506 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags [.], seq 39097:40545, ack 118, win 91, options [nop,nop,TS val 2699823150 ecr 581354], length 1448: HTTP

10:59:47.176599 IP 192.168.3.195.39965 > 138.231.176.10.80: Flags [.], ack 40545, win 331, options [nop,nop,TS val 581358 ecr 2699823150], length 0

```
10:59:47.178704 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags [.], seq 40545:41993, ack 118, win 91, options [nop,nop,TS val
2699823150 ecr 581355], length 1448: HTTP

10:59:47.178751 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags [P.], seq 41993:42339, ack 118, win 91, options [nop,nop,TS val
2699823150 ecr 581355], length 346: HTTP

10:59:47.178831 IP 192.168.3.195.39965 > 138.231.176.10.80: Flags [.], ack 42339, win 331, options [nop,nop,TS val 581359 ecr
2699823150], length 0

10:59:47.179679 IP 192.168.3.195.39965 > 138.231.176.10.80: Flags [F.], seq 118, ack 42339, win 331, options [nop,nop,TS val 581359 ecr
2699823150], length 0

10:59:47.190218 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags [F.], seq 42339, ack 119, win 91, options [nop,nop,TS val 2699823154
ecr 581359], length 0

10:59:47.190284 IP 192.168.3.195.39965 > 138.231.176.10.80: Flags [.], ack 42340, win 331, options [nop,nop,TS val 581362 ecr
2699823154], length 0
```

Exercice 2 Analyse du fichier trace.pcapet

Téléchargez le fichier

https://www.irif.fr/~jch/enseignement/reseaux/trace.pcap

et examinez-le, d'abord à l'aide de la commande « *tcpdump -n -r fichier*», ensuite à l'aide de « *tcpdump -r fichier*», enfin à l'aide de *wireshark*.

La page qui est à télécharger sur la page du cours nous présente un échange, c'est un petit fichier de 47 kilo octets. Dans ce TP on va utiliser surtout **tcpdump**.

Commandes

```
➢ tcpdump -n -r trace.pcap
```

```
➢ wireshark trace.pcap
```

1#

Quelles sont les adresses des interfaces qui communiquent ?

- ☑ **-n** : Don't convert addresses (i.e., host addresses, port numbers, etc.) to names.
- ☑ **tcpdump** always prints the name of the sending host, then a greater than sign, then the name of the destination host.

```
PS C:\Users\lenny\Downloads>  ./tcpdump -n -r trace.pcap
```

```
10:59:47.105310 IP 192.168.3.195.39965 > 138.231.176.10.80: Flags
[S], seq 1793060694, win 14600, options [mss 1460,sackOK,TS val
581341 ecr 0,nop,wscale 7], length 0

10:59:47.113894 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags
[S.], seq 3675933190, ack 1793060695, win 5792, options [mss
1460,sackOK,TS val 2699823136 ecr 581341,nop,wscale 6], length 0
```

On voit des divers échanges entre $192.168.3.195.\underbrace{39965}_{\textit{Numero de port}}$

et $138.231.176.10.\underbrace{80}_{\textit{Numero de port}}$.

Les adresse IP sont afficher au format avec 5 champ et pas 4 pour inclure les numéro de port.

Le protocole de la couche transport : TCP ou UDP ?

Réponse : TCP car on voit les numéros de séquence et les numéros d'acquittement. De plus, on voit qu'il y a des options.

```
10:59:47.137330 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags [.], seq
2897:4345, ack 118, win 91, options [nop,nop,TS val 2699823141 ecr 581346], length
1448: HTTP

10:59:47.137394 IP 192.168.3.195.39965 > 138.231.176.10.80: Flags [.], ack 4345,
win 182, options [nop,nop,TS val 581349 ecr 2699823141], length 0
```

TCP *(Transmission Control Protocol)*.

L'en-tête d'un segment TCP comporte également (entre autres) :

- Un **champ de numéro de séquence** et un **champ d'accusé de réception** (tous les deux de 32 bits), utiliser par l'expéditeur et le destinataire TCP dans la mise en œuvre du service de transfert fiable.
- Le **champ d'options** de longueur variable intervient au moment de la négociation de la taille du MSS entre l'expéditeur et le destinataire, ou en tant que facteur d'ajustement de la fenêtre au sein des réseaux a hauts débits. Il existe également une option d'horodatage.

Source : James W. Kurose, Keith W. Ross. Computer Networking, A Top-Down Approach. (Edition française)

- Les numéro de séquence et les numéro d'Ack : c'est propre à TCP.
- On a aussi les tailles de fenêtres.
- En général une entête proprement TCP fait que 20 octets car il y a pas d'options, mais la-il y a des options...
- De manière général, Il y a beaucoup plus d'information dans un paquet TCP que UDP.

TCP Packets

The general format of a TCP protocol line is:

src > dst: Flags [tcpflags], seq data-seqno, ack ackno, win window, urg urgent, options [opts], length len

UDP Packets

UDP format is illustrated by this rwho packet:

actinide.who > broadcast.who: udp 84

- This says that port who on host actinide sent a udp datagram to port who on host broadcast, the Internet broadcast address.
- The packet contained 84 bytes of user data.
- Some UDP services are recognized (from the source or destination port number) and the higher level protocol information printed.

Source : man tcpdump

3#

Pourquoi autant de paquets ont-ils une taille égale à 1448 ?

De manière général : *limite de 1 500 octets en couche 2*
(Une trame Ethernet : 1500 octets)

Si on regarde la première ligne, on voit **mss = 1460**

```
reading from file ../trace.pcap, link-type EN10MB (Ethernet)

10:59:47.105310 IP 192.168.3.195.39965 > 138.231.176.10.80: Flags
[S], seq 1793060694, win 14600, options [mss 1460,sackOK,TS val
581341 ecr 0,nop,wscale 7], length 0
```

Ça veut dire qu'il nous reste cette taille enfaite pour les données.
mss 1460 : données de la couche application.
→ **Donc, a priori on peut mètre 1460 octets de données de la couche application**

Donc si il est écrit 1460, pourquoi au final il y a des paquets de taille 1448 qui circules ?

Une trame Ethernet : 1500 octets

IP	TCP	Charge utile
20 octets	20 octets	1460 octets

il y a 1500, au niveau de la trame Ethernet. on a l'entête IP, et l'entête TCP et donc il reste 1460 pour la charge utile, charge utile = normalement les données de la couche application, mais si on rajoute des options TCP, alors ça enlève de la taille, et là il y a des options...

L'ajout d'options réduit la taille de 1460 octets :

IP	TCP	Options	Charge utile
20 octets	20 octets	———	1460 octets ———

On utilise des options qui « mange » de la place donc il reste moins de place pour la charge utile.

Le maximum qu'on peut transmettre c'est 1448 . Mais alors pourquoi il y a beaucoup de paquets qui ont une taille qui est égale au maximum ?

Les paquets ont une taille qui fait 1500. dès le début on avait dit qu'on pouvait pas envoyer des paquets plus grand que 1460 + la taille des en-têtes , donc : 1500, donc pas de fragmentations au niveau des paquets.

On voit que ce sont des paquets http, donc on connait à l'avance quelle est la limite, donc on a un algorithme glouton qui envoie le max possible à chaque fois, car on veut le moins de fragment possible.

La limite de capacité étant connu à l'avance, l'émetteur envoie des paquets de taille adapté.

Dans les traces de paquets,

- Ce ne sont pas des paquets agrèges, mais de vrais paquets. pas de paquets de la couche 4, paquets TCP.
- A la fin : paquet de 346 octets (ce qu'il a resté a envoyé après l'envoie de tous les paquets de 1448 octets)

```
10:59:47.178751 IP 138.231.176.10.80 > 192.168.3.195.39965:
Flags [P.], seq 41993:42339, ack 118, win 91, options
[nop,nop,TS val 2699823150 ecr 581355], length 346: HTTP
```

- La diff entre 2 numéro de séquence consécutifs c'est 1448, ce qui est la longueur des donnes pour la couche application.

Le protocole HTTP

Protocole HTTP (HyperText Transfer Protocol)

Le protocole de transfert utilisé pour le Word Wide Web est HTTP. Il définit les messages que les clients peuvent envoyer au serveur, et ceux que le serveur peut transmettre en réponse. Tous les clients et tous les serveurs doivent respecter les spécifications de ce protocole.

Source : Andrew S. Tanenbaum. Computer Networks.. (Edition française)

HTTP est un protocole requête-réponse : il y a une requête http et une réponse http.
On peut analyser ses paquets avec Wireshark :

- La 1er ligne ressemble a ce qu'on avait avec **tcpdump**, avec des colonnes sépares.
- On voie la requête http en cliquant sur la colonne http.
- On a divers info comme que c'est le Protocol http 1.1
- Une repose http : la réponse fait une première ligne avec le **status** qui est 200 si tout va bien.
- On peut voir également la date. A priori c'est la date de la requête.
- On voit les divers champ et après on a le fichier lui-même, un fichier html.

4#

Y a-t-il eu des pertes de paquets ?

Un paquet **ACK 40545** ça veut dire quoi ?

```
10:59:47.176599 IP 192.168.3.195.39965 > 138.231.176.10.80: Flags [.], ack 40545, win 331, options [nop,nop,TS val 581358 ecr 2699823150], length 0
```

Ça veut dire que j'ai reçu les 40545 premières octets, depuis 0 jusqu'à 40544. Prochain octet attendu : 40545.

De manière général, les pertes sont signaler avec des acquittements dupliqué.

Si on reçoit le segment **[0 – 1000 [** et après on reçois le segment **[2000 – 3000 [**, alors on sait que **[1000 – 2000 [** est perdu.

[0 – 1000 [*recu* ⟶ *ack* **1000**

[1000 – 2000 [*perdu*

[2000 – 3000 [*recu* ⟶ *ack* **1000**

[1000 – 2000 [*recu* ⟶ *ack* **3000**

ack dupliqué signale une perte

S'il n'y aurai pas de pertes : le ACK suivant aurai été un ACK de 2000.

Donc, y a-t-il eu des pertes de paquets ?

A compléter sur le pdf.

5#

Pourquoi autant de ack 118 ?

Il y a beaucoup de ack 118 , est-ce que ça signale qu'il y a beaucoup de perte de paquets?

Il est vrai que les ack dupliqué signale des pertes, mais il y a un autre mécanisme qui peut expliquer les ack dupliquer et la c'est cet autre mécanisme.

Et donc, alors pourquoi autan de ack 118 ?

Si on regarde bien on voit qu'il y a beaucoup de paquets de longueur 0. Dans un tel cas, on augmente pas le numéro d'ack.

A quoi ça sert d'envoyer un paquet de longueur 0 ?

C'est intéressent d'envoyer des TCP de longueur 0 car il contient quand même des options et il transmet aussi un numéro de ack, taille de fenêtre, numéro de séquence(*)..

Le champ win ça sert à quoi ? à régler la vitesse , ça dit combien d'octets je peux encore recevoir avant de bloquer.

TCP a donc des mécanismes de gestion de la congestion, pour que la transmission de donnes se fasse a la bonne vitesse.

(*) **tcpdump** n'affiche pas les numéros de séquences des paquets de longueur 0 tandis que **Wireshark** l'affiche.

```
10:59:47.149400 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags
[.], seq 11585:13033, ack 118, win 91, options [nop,nop,TS val
2699823144 ecr 581349], length 1448: HTTP
```

win = taille de la fenêtre du récepteur, ça règle la vitesse.

Si la valeur est 0 : STOP, demande d'arrêt d'envoi de donnes.

6#

Pourquoi les numéros de séquence des deux premiers paquets sont énormes, et ceux des suivants petits ?

Numéros de séquence

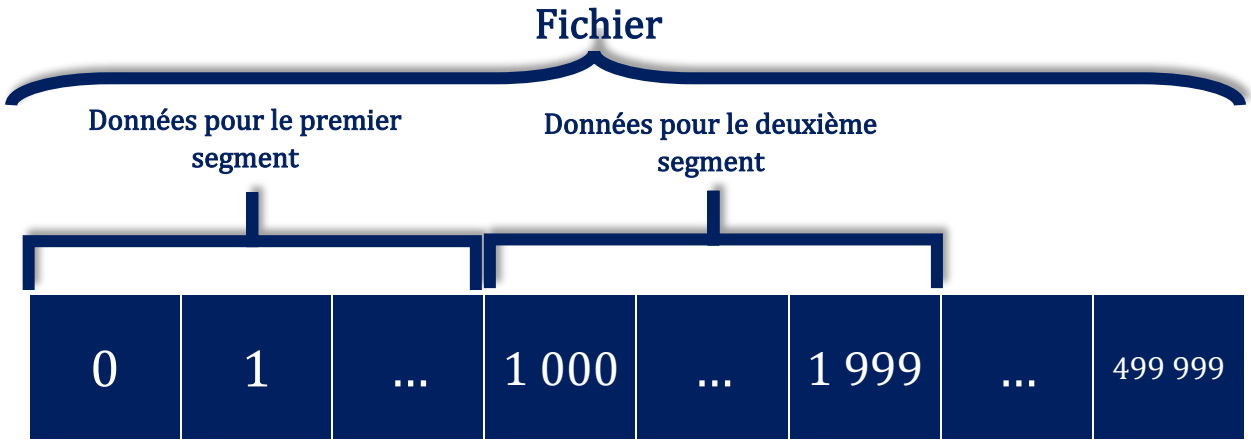
Le champ d'en-tête de **numéro de séquence** et d'**accusé de réception** des segments TCP jouent un rôle fondamental dans le service de transfert de données fiable de TCP.

Au regard de TCP, les données à transférer se présentent sous forme de flux sans structure, mais utilisant un ordre précis. Ces numéros de séquence TCP s'appliquent aux différents flux d'octets et **non** aux différents segments génères pour leur transfert.

Le **numéro de séquence d'un segment** est ainsi le numéro dans le flux d'octets du premier octet de chaque segment.

Soit un processus exploité sur le serveur A qui souhaite envoyer un flux de donnes à un processus du serveur B sur la connexion TCP.

- ✓ Le TCP du serveur A procède automatiquement à la numérotation de tous les octets du flux de données.
- ✓ Supposons que le flux de données consiste en un fichier de 500 Ko, que la taille du MMS soit fixée à 1 Ko et que le premier octet du flux de données ait le numéro séquence 0.
- ✓ TCP scindera les flux de données en 500 segments (voir graphique).
- ✓ Le premier segment aura le numéro 0, le deuxième le numéro 1000, le troisième le numéro 2000, etc.
- ✓ Chaque numéro de séquence est inséré dans le champ de numéro de séquence de l'en-tête du segment TCP approprié.



Fragmentation des données d'un fichier en segments TCP.

Source : James W. Kurose, Keith W. Ross. Computer Networking, A Top-Down Approach. (Edition française)

Exemple qui apparait sur man tcpdump

Here is the opening portion of an rlogin from host rtsg to host csam.

```
IP rtsg.1023 > csam.login: Flags [S], seq 768512:768512, win 4096, opts [mss 1024]
IP csam.login > rtsg.1023: Flags [S.], seq, 947648:947648, ack 768513, win 4096, opts [mss 1024]
IP rtsg.1023 > csam.login: Flags [.], ack 1, win 4096
IP rtsg.1023 > csam.login: Flags [P.], seq 1:2, ack 1, win 4096, length 1
IP csam.login > rtsg.1023: Flags [.], ack 2, win 4096
IP rtsg.1023 > csam.login: Flags [P.], seq 2:21, ack 1, win 4096, length 19
IP csam.login > rtsg.1023: Flags [P.], seq 1:2, ack 21, win 4077, length 1
IP csam.login > rtsg.1023: Flags [P.], seq 2:3, ack 21, win 4077, urg 1, length 1
IP csam.login > rtsg.1023: Flags [P.], seq 3:4, ack 21, win 4077, urg 1, length 1
```

☑ The first line : The packet sequence number was 768512 and it contained no data. (**line 2 with a similar packet** except it includes a piggy-backed ack..)

The notation is first:last which means sequence numbers first up to but not including last.

☑ Line 3 : The packet contained no data so there is no data sequence number or length.

☑ The first time tcpdump sees a TCP "conversation", it prints the sequence number from the packet.

- On subsequent packets (paquets suivants) of the conversation, the difference between the current packet's sequence number and this initial sequence number is printed.
- This means that sequence numbers after the first can be interpreted as relative byte positions in the conversation's data stream (with the first data byte each direction being `1').
- `-S' will override this feature, causing the original sequence numbers to be output.

The notation is **first:last** which means sequence numbers first up to but not including last.

Donc, 2 chiffres pour avoir plus de lisibilité. Par exemple :

```
10:59:47.160960 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags [.], seq 20273:21721, ack 118, win 91, options [nop,nop,TS val 2699823147 ecr 581352], length 1448: HTTP

10:59:47.163259 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags [.], seq 21721:23169, ack 118, win 91, options [nop,nop,TS val 2699823147 ecr 581352], length 1448: HTTP
```

- La diff entre 2 numéro de séquence consécutifs c'est 1448, ce qui est la longueur des donnes pour la couche application.

- Au total, Un peu avant la fin du fichier :

```
10:59:47.178751 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags [P.], seq 41993:42339, ack 118, win 91, options [nop,nop,TS val 2699823150 ecr 581355], length 346: HTTP
```

Il y a donc, selon le numéro de séquence, 42339-42340 octets qui ont été envoyé.

Retour à la question..

Pourquoi les numéros de séquence des deux premiers paquets sont énormes, et ceux des suivants petits ?

Par exemple, on a :

```
10:59:47.163259 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags [.], seq 21721:23169, ack 118, win 91, options [nop,nop,TS val 2699823147 ecr 581352], length 1448: HTTP
```

Mais, pourquoi au début les numéros de séquence des deux premiers paquets sont énormes ?

```
10:59:47.105310 IP 192.168.3.195.39965 > 138.231.176.10.80: Flags [S], seq 1793060694, win 14600, options [mss 1460,sackOK,TS val 581341 ecr 0,nop,wscale 7], length 0
```

```
10:59:47.113894 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags
[S.], seq 3675933190, ack 1793060695, win 5792, options [mss
1460,sackOK,TS val 2699823136 ecr 581341,nop,wscale 6], length 0
```

En réalité les numéros de séquence de paquets sont renuméroter par **tcpdump** pour la lisibilité.

Mais, *sauf les 2 premier :* lors de l'échange TCP initial, les premiers paquets tire des numéros de séquence au hasard pour éviter des attaques « man in the middle » car cela rend compliquer d'injecter des faux paquets. C'est un mécanisme de sécurité pour faire qu'on ne commence pas à 0. donc en réalité on commence avec un numéro de séquence élever et ensuite, pour la lisibilité, **tcpdump** diminue après les **seq**.

Commande

-n : Don't convert addresses (i.e., host addresses, port numbers, etc.) to names. ***(man tcpdump)***

```
➢    tcpdump -r trace.pcap
```

reading from file trace.pcap, link-type EN10MB (Ethernet)

11:59:47.105310 IP 192.168.3.195.39965 > spider3.ens-cachan.fr.80: Flags [S], seq 1793060694, win 14600, options [mss 1460,sackOK,TS val 581341 ecr 0,nop,wscale 7], length 0

11:59:47.113894 IP spider3.ens-cachan.fr.80 > 192.168.3.195.39965: Flags [S.], seq 3675933190, ack 1793060695, win 5792, options [mss 1460,sackOK,TS val 2699823136 ecr 581341,nop,wscale 6], length 0

11:59:47.113990 IP 192.168.3.195.39965 > spider3.ens-cachan.fr.80: Flags [.], ack 1, win 115, options [nop,nop,TS val 581343 ecr 2699823136], length 0

8#

Interpréter les flags (lettre entre crochets comme [S.])

Les flags : que en TCP. On a par exemple : [.] , [p.] , [F.].....

The general format of a TCP protocol line is:

src > dst: Flags [tcpflags], seq data-seqno, ack ackno, win window, urg urgent, options [opts], length len

Tcpflags are some combination of :

tcpdump	Nom du flag	Explication
S	SYN	Demande d'ouverture de connexion
F	FIN	Fin de connexion
P	PUSH	EOM (ou PSH) indique une fin de message (*End of Message*), les données doivent être transmises (*pushed*) à la couche supérieure.
R	RST	Demande de réinitialisation de la connexion.
U	URG	Si le champ de priorités est utilisé (pour des demandes d'interruption d'émission par exemple)
W	ECN-Echo	
.	ACK	Si la valeur du champ acquittement est significative
none	if no flags are set	

Sources :

man tcpdump

INTERNET : SERVICES ET RESEAUX Stéphane Lohier Dominique Présent DUNOD

- Lors de le connexion TCP il y a des SYN, donc le flag S c'est pour établir une connexion TCP.
- F = Finalise, fin de la transmission, paquets pour raccrocher.
  En plus, chaque un des paquets de fin doit être acquittée.
- Et en même temp on ferme la communication dans le sens serveur-client.

- PUSH – au niveau application on a fini d'envoyer les donnes. Ça peut être utilisé par la couche application en violant le mécanisme d'encapsulation.

8#

Vers la fin on dirait qu'il y a moins de ack. Pourquoi ?

L'échange serveur-client décrit dans les traces:

- Au début on a un ping-pong paquet-acquittement.
- Puis, a un certain moment, il y a 2 paquets et que 1 acquittement.. pourquoi ?

En fait TCP n'oblige pas à acquitter chaque paquet. Par exemple, si il y a plain de paquets en vols, on évite d'envoyer beaucoup d'acquittements, donc il existe le mécanisme de l'acquittement retardé (*delayed ack*) qui permet d'envoyer un acquittement pour un certain nombre de paquets. C'est possible car l'émetteur nattent pas l'acquittement pour envoyer le prochain paquet.

9#

Interprétez les champs val et ecr

Par exemple,

```
10:59:47.163385 IP 138.231.176.10.80 > 192.168.3.195.39965:
Flags [.], seq 24617:26065, ack 118, win 91, options
[nop,nop,TS val 2699823147 ecr 581352], length 1448: HTTP
```

val, ecr : Données d'horodatage, pour pouvoir calculer la latence de la ligne. En TCP, avec le mécanisme de perte et **ack** retardé, le numéro de séquence est pas suffisant pour calculer la latence de la ligne.

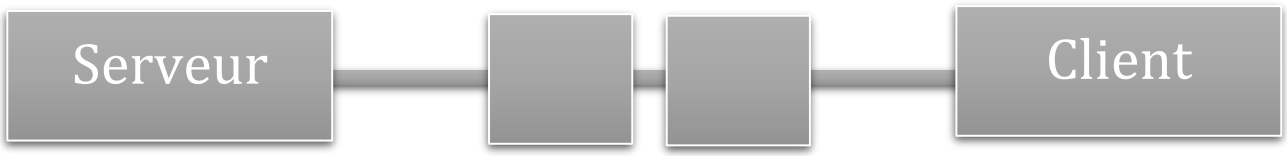
L'option **nop** - purage. Quand c'est pas un multiple de 4 on ajoute **nop** pour avoir un multiple de 4.

Il semble qu'il manque un champ de type **temp** (l'horodatage lui-même).

10#

Sur quel hôte la trace a-t-elle été capturée ? Justifier.

La trace on peut la capturer sur divers machine, on a un ordi qui est relier a un router... , ensuite on a un serveur.



Des machines au milieu

Donc, la capture était faite sur quelle machine ? le client ? une machine au milieu ? le serveur ?

D'abord y'a ce qui semble le plus évident, quand on fait une requête Web , a priori c'est plus facile de faire la capture sur la machine qui a fait la requête. De plus que de lancer **tcpdump**, a priori c'est plus facile à faire sur le client que sur le serveur ou le routeur a moins d'être l'administrateur.

Donc, ça semble logique que ça soit le client, mais comment on peut le prouver ?

Il faut regarder les horodatages, regarder au début de la trace, lorsque on envoie un accuser de réception par paquet (« lorsque on est en phase ping-pong »). Il est plus simple de le voir sur un dessin : un chronogramme des paquets.

Dans le cas du début, ou on a une série de « **Seq, Ack, Seq, ack...** », si je capture sur la machine A (dans notre cas : le serveur web), alors si j'envoie un paquet avec un numéro de séquence X, on attend un long temp avant de recevoir l'acquittement de X, et tout de suite on vas envoyer le paquet avec le numéro de séquence Y, et on va attendre longtemps avant d'avoir l'acquittement de Y..

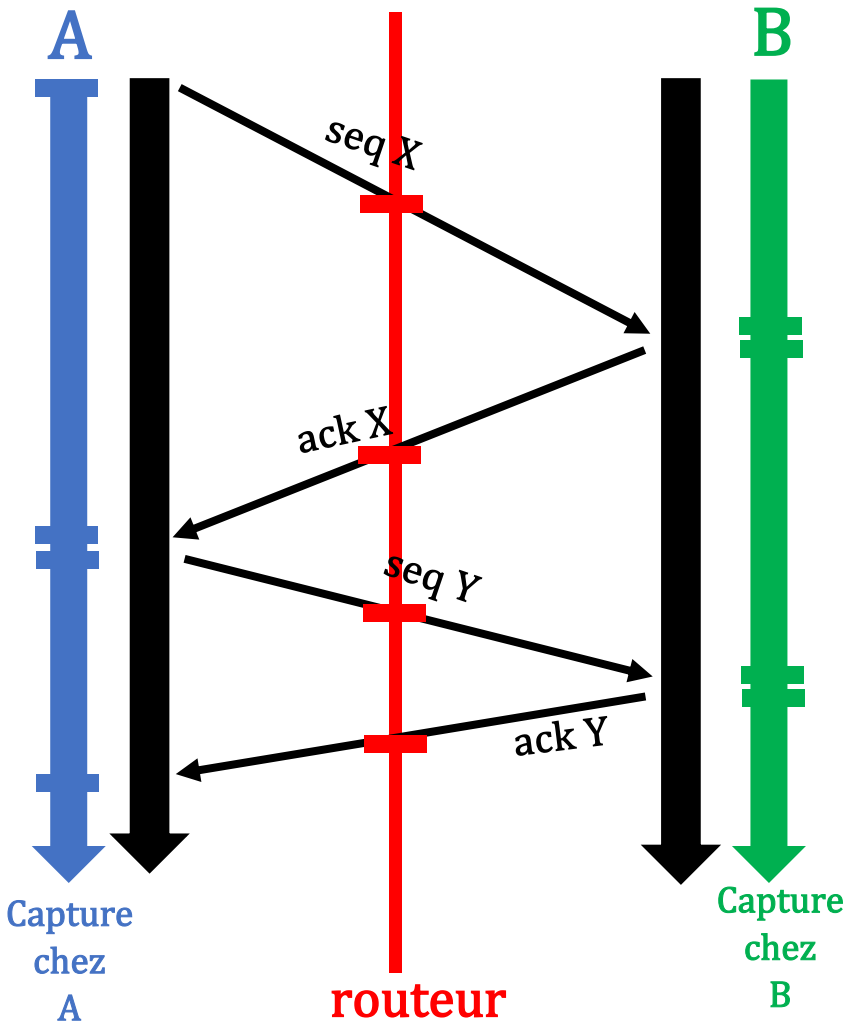
Par contre, si on fait la capture sur la machine B (dans notre cas : le client web), c'est le contraire : on reçoi un paquet avec le numéro de séquence X, on envoie tout de suite l'acquittement, et on attend beaucoup de temps avant le paquet avec le numéro de séquence suivant, et lorsque ce paquet arrive, on envoie tout de suite l'acquittement.

Et si la capture est faite sur le routeur ? dans ce cas les intervalles vont être bien réparti, entre un paquet reçue et la réaction à sa réception.

Donc en regardent les eurodatages, on peut savoir si on est plutôt du côté d'une machine A, au milieu, ou de l'autre côté.

Et ici on voie que dit qu'arrive un paquet avec un numéro de séquence X, on envoie tout de suite l'acquittement, alors que parfois lorsque on a envoyé un acquittement, on attend beaucoup de temps avant de recevoir le paquet suivant.

Donc : *dans ce cas, la capture était faite sur le client web.*



10:59:47.139685 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags [.], seq 5793:7241, ack 118, win 91, options [nop,nop,TS val 2699823141 ecr 581346], length 1448: HTTP

10:59:47.139748 IP 192.168.3.195.39965 > 138.231.176.10.80: Flags [.], ack 7241, win 228, options [nop,nop,TS val 581349 ecr 2699823141], length 0

10:59:47.139812 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags [.], seq 7241:8689, ack 118, win 91, options [nop,nop,TS val 2699823141 ecr 581346], length 1448: HTTP

10:59:47.139827 IP 192.168.3.195.39965 > 138.231.176.10.80: Flags [.], ack 8689, win 250, options [nop,nop,TS val 581349 ecr 2699823141], length 0

10:59:47.149209 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags [.], seq 8689:10137, ack 118, win 91, options [nop,nop,TS val 2699823144 ecr 581349], length 1448: HTTP

10:59:47.149295 IP 192.168.3.195.39965 > 138.231.176.10.80: Flags [.], ack 10137, win 273, options [nop,nop,TS val 581352 ecr 2699823144], length 0

10:59:47.149355 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags [.], seq 10137:11585, ack 118, win 91, options [nop,nop,TS val 2699823144 ecr 581349], length 1448: HTTP

10:59:47.149371 IP 192.168.3.195.39965 > 138.231.176.10.80: Flags [.], ack 11585, win 296, options [nop,nop,TS val 581352 ecr 2699823144], length 0

10:59:47.149400 IP 138.231.176.10.80 > 192.168.3.195.39965: Flags [.], seq 11585:13033, ack 118, win 91, options [nop,nop,TS val 2699823144 ecr 581349], length 1448: HTTP

10:59:47.149437 IP 192.168.3.195.39965 > 138.231.176.10.80: Flags [.], ack 13033, win 318, options [nop,nop,TS val 581352 ecr 2699823144], length 0

Exercice 3

#

Que se passe-t-il dans les fragments de traces suivants ? Toutes les traces ont été capturées sur la machine « A » (comment peut-on le voir ?).

- On voit partout des **win**, donc c'est 5 échanges **TCP** .
- Pas exactement la même convention d'affichage comme pour **tcpdump** : en particulier: au lieu d'avoir le format « une adresse IP, point , un port » comme 138.131.176.10.80 , on a remplacé les adresses IP par A et B, et le port 80 on la remplacer par point www (port 80 c'est normalement le port du service http, donc on remplace 80 par www), de même pour ssh. De plus, la longueur est donné entre parentaises.

1.

```
22:45:26.121149 A.32769 > B.www: S 2919412148:2919412148(0) win 5840
                                   <mss 1460,nop,wscale 1>

22:45:26.123055 B.www > A.32769: S 4144006771:4144006771(0)
                                   ack 2919412149 win 65535
                                   <mss 1460,nop,wscale 1>
22:45:26.123120 A.32769 > B.www: . ack 1 win 2920
22:45:26.124144 A.32769 > B.www: P 1:146(145) ack 1 win 2920
22:45:26.130323 B.www > A.32769: . 1:1461(1460) ack 146 win 32850
22:45:26.130402 A.32769 > B.www: . ack 1461 win 4380
22:45:26.130750 B.www > A.32769: . 1461:2921(1460) ack 146 win 32850
```

- La séquence 1, elle ressemble à quoi ? A quoi ces 7 paquets nous font penser ?
- Une question a ce poser à chaque fois : on est au début / milieu / fin de la capture ?
- Donc on est au début et là on voit que ça ressemble au traces déjà vu : serveur http qui répond à un client http.

- Le 4eme paquet : on voit que il y a 145 octets, et après 1460...
- Les 2 premiers segment de la réponses qui font 1460.

- B – serveur web.
  A – client web (au début il fait un get).
- http c'est l'exemple tipe de requête-réponse, il y a des protocoles plus bavards..

2.

```
22:45:26.195481 B.www > A.32769: . 74461:75921(1460) ack 146 win 32850
22:45:26.196216 B.www > A.32769: P 75921:77381(1460) ack 146 win 32850
22:45:26.196228 A.32769 > B.www: . ack 77381 win 64240
22:45:26.211281 B.www > A.32769: . 77381:78841(1460) ack 146 win 32850
22:45:26.211772 B.www > A.32769: P 78841:80301(1460) ack 146 win 32850
22:45:26.211783 A.32769 > B.www: . ack 80301 win 64240
```

- Il y a de l'acquittement tardé.

3.

```
22:45:26.240764 B.www > A.32769: P 127021:128387(1366) ack 146 win 32850
22:45:26.240776 A.32769 > B.www: . ack 128387 win 64240
22:45:26.249437 A.32769 > B.www: F 146:146(0) ack 128387 win 64240
22:45:26.251418 B.www > A.32769: . ack 147 win 32850
22:45:26.251840 B.www > A.32769: F 128387:128387(0) ack 147 win 32850
22:45:26.251871 A.32769 > B.www: . ack 128388 win 64240
```

- Il y a les **F** donc c'est la fin du transfert.
- A ferme la communication ; B acquitte.
- B ferme la communication ; A acquitte.

- Une socket TCP c'est une socket bi-directionnelle.

4.

```
22:49:22.739301 A.32775 > C.ssh: . 110064:111524(1460) ack 2336 win 5104
22:49:22.739312 A.32775 > C.ssh: . 111524:112984(1460) ack 2336 win 5104
22:49:22.772816 C.ssh > A.32775: . ack 96924 win 62780
22:49:22.772828 A.32775 > C.ssh: . 112984:114444(1460) ack 2336 win 5104
22:49:22.772838 A.32775 > C.ssh: . 114444:115904(1460) ack 2336 win 5104
22:49:22.773905 C.ssh > A.32775: . ack 99844 win 62780
```

- SSH c'est aussi pour ce connecté a une machine a distance, mais ajd ça peut avoir des significations plus larges, on peut transférer bcp de choses via SSH, c'est une sorte de sous-couche.

5.

```
22:53:47.759896 D.17775 > A.32782: . 75921:77381(1460) ack 1 win 17520
22:53:47.760031 D.17775 > A.32782: . 77381:78841(1460) ack 1 win 17520
22:53:47.760055 A.32782 > D.17775: . ack 78841 win 64240
22:53:47.885001 D.17775 > A.32782: . 80301:81761(1460) ack 1 win 17520
22:53:47.885072 A.32782 > D.17775: . ack 78841 win 64240
22:53:47.885816 D.17775 > A.32782: . 83221:84681(1460) ack 1 win 17520
22:53:47.885834 A.32782 > D.17775: . ack 78841 win 64240
22:53:47.885951 D.17775 > A.32782: . 84681:86141(1460) ack 1 win 17520
22:53:47.885962 A.32782 > D.17775: . ack 78841 win 64240
22:53:47.917054 D.17775 > A.32782: . 86141:87601(1460) ack 1 win 17520
22:53:47.917065 A.32782 > D.17775: . ack 78841 win 64240
22:53:48.042369 D.17775 > A.32782: . 78841:80301(1460) ack 1 win 17520
22:53:48.042419 A.32782 > D.17775: . ack 81761 win 64240
22:53:48.199537 D.17775 > A.32782: . 81761:83221(1460) ack 1 win 17520
22:53:48.199602 A.32782 > D.17775: . ack 87601 win 64240
22:53:48.199718 D.17775 > A.32782: . 89061:90521(1460) ack 1 win 17520
22:53:48.199732 A.32782 > D.17775: . ack 87601 win 64240
22:53:48.356490 D.17775 > A.32782: . 87601:89061(1460) ack 1 win 17520
22:53:48.356551 A.32782 > D.17775: . ack 90521 win 64240
22:53:48.356620 D.17775 > A.32782: . 90521:91981(1460) ack 1 win 17520
```

- Transfert uni-directionnelle.
- Il y a des pertes de paquets.
- L'ordre a lequelle les segment sont envoyée est pas croissant car faut rattraper les pertes.
- Cette échange ce fait sur une échange pas très stable.