# Lesson 4: End-to-End Protocols

Van-Linh Nguyen

Fall 2024

Online video link: https://youtu.be/of5CmizjSW8

國立中正大學 Cybersecurity Lab

# Outline

- Lab2: Create a network service/app in Packet Tracer

- Lab3: Create a simple Internet network

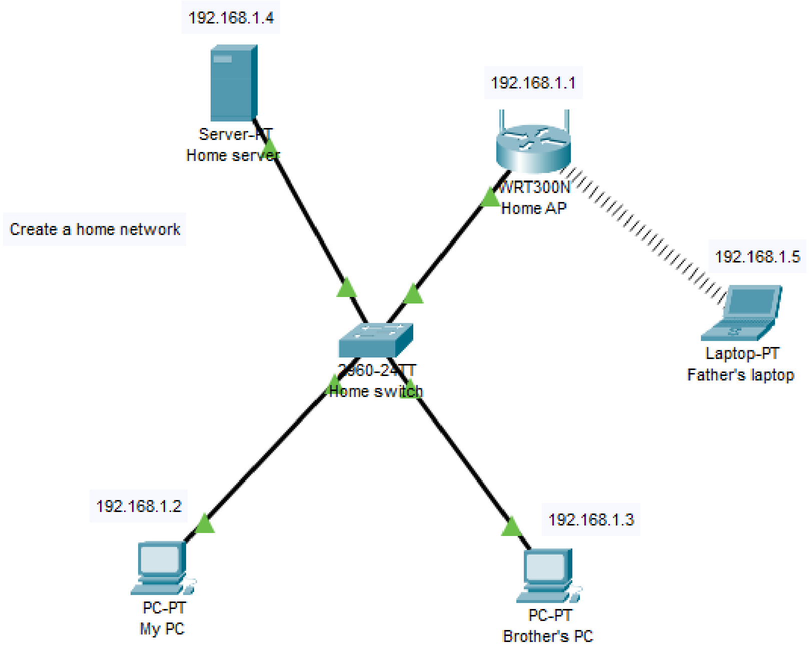
- Network protocols
  1. UDP
  2. TCP

# Lab2: Service/application

Youtube link for Lab2 reference

https://youtu.be/azCHlVFvY1o

- This lab is to recall knowledge about application and services
- Based on the first lab, please create a new DNS server with IP 192.168.1.6
- Assign the webserver at 192.168.1.4 with the domain ccu.edu.tw
- Test whether PCs can access Web on the Home server by using the domain name
- Activate FTP services on the Home server and create an account admin/admin → Test FTP connection from laptop

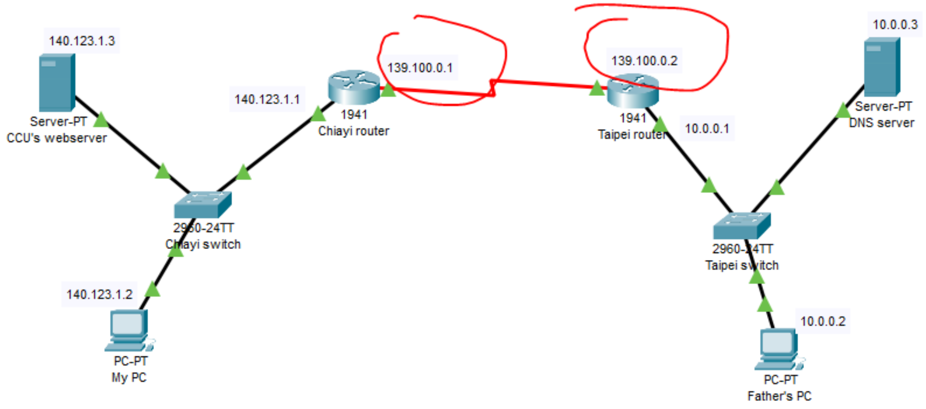


# Lab3: Create an Internet network

- Create a simple Internet between Chiayi and Taipei
- The Chiayi network starts with the network IP 140.123.1.0/24
- The Taipei network starts with the network IP 10.0.0.0/16
- Test whether PCs in the Chiayi network can access PCs at the Taipei network
- Create a Webserver at the Chiayi network (CCU's website) and a DNS server at the Taipei network
- Assign the webserver at 140.123.1.3 with the domain ccu.edu.tw
- Create a DNS record for the website at DNS server
- Test whether Father's PC can access CCU's website via domain name

Youtube link for Lab3 reference

https://youtu.be/IYusDFMPZWU

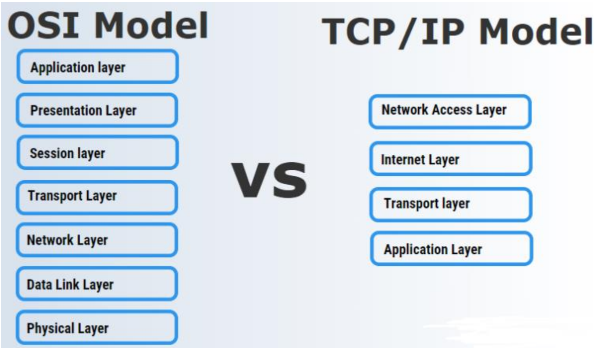


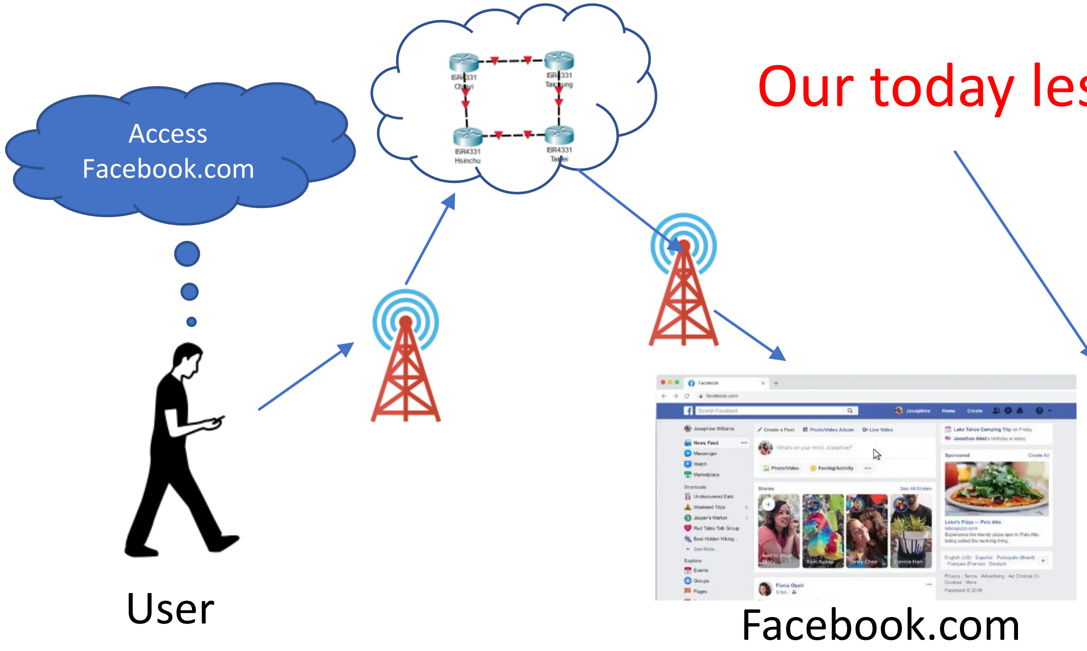
Network IP between Chiayi router and Taipei router must be different from the connected networks (140.123.1.0, 10.0.0.0)

In the sample lab, we use IP 139.100.0.0 for that network and the IPs for serial port can start with 139.100.0.1 and 139.100.0.2 as Illustrated in the above figure

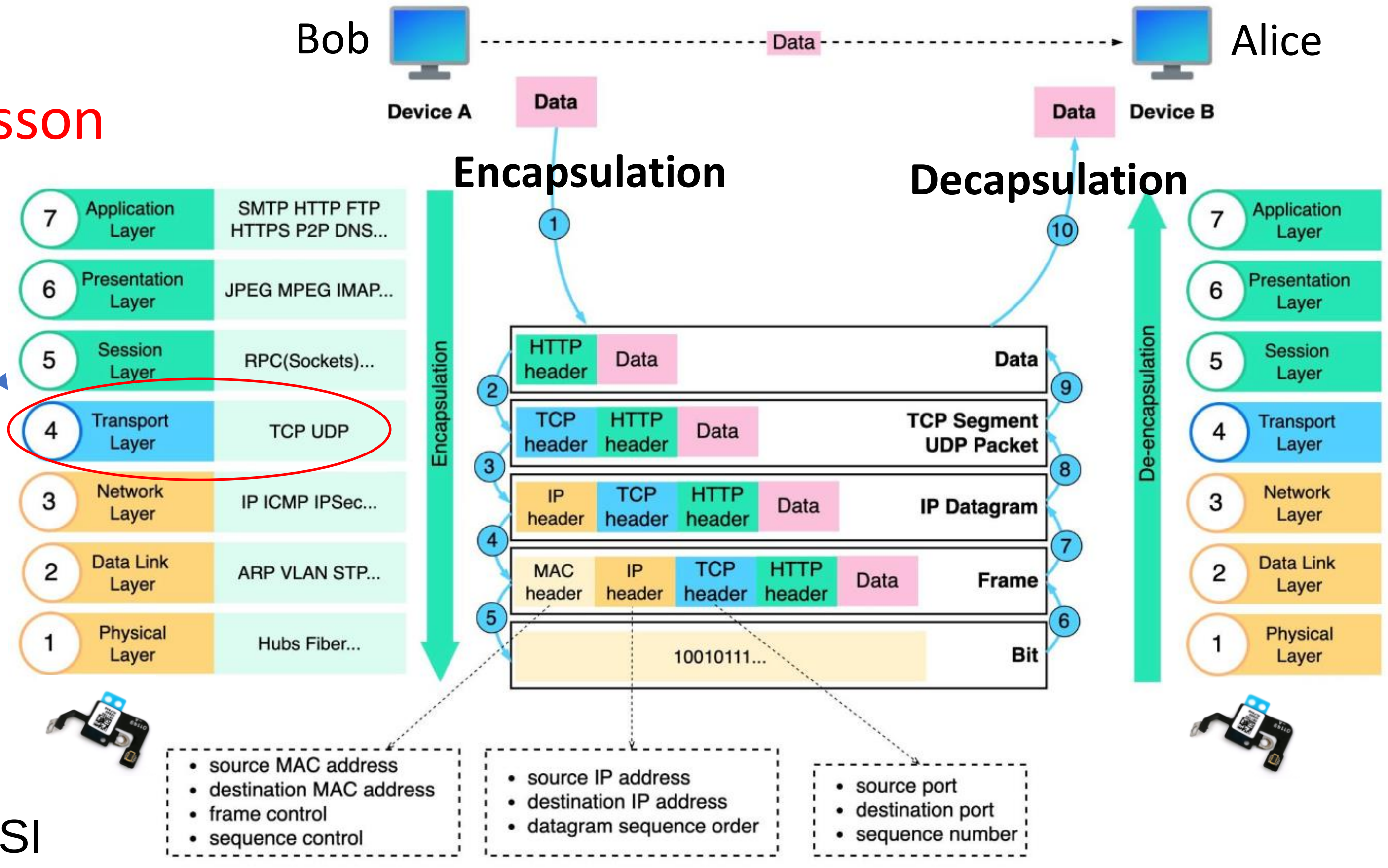
國立中正大學 Cybersecurity Lab

# Encapsulation/Decapsulation





Our today lesson



| | | |
|---|---|---|
| 7 | Application Layer | SMTP HTTP FTP HTTPS P2P DNS... |
| 6 | Presentation Layer | JPEG MPEG IMAP... |
| 5 | Session Layer | RPC(Sockets)... |
| 4 | Transport Layer | TCP UDP |
| 3 | Network Layer | IP ICMP IPSec... |
| 2 | Data Link Layer | ARP VLAN STP... |
| 1 | Physical Layer | Hubs Fiber... |

- source MAC address
- destination MAC address
- frame control
- sequence control

- source IP address
- destination IP address
- datagram sequence order

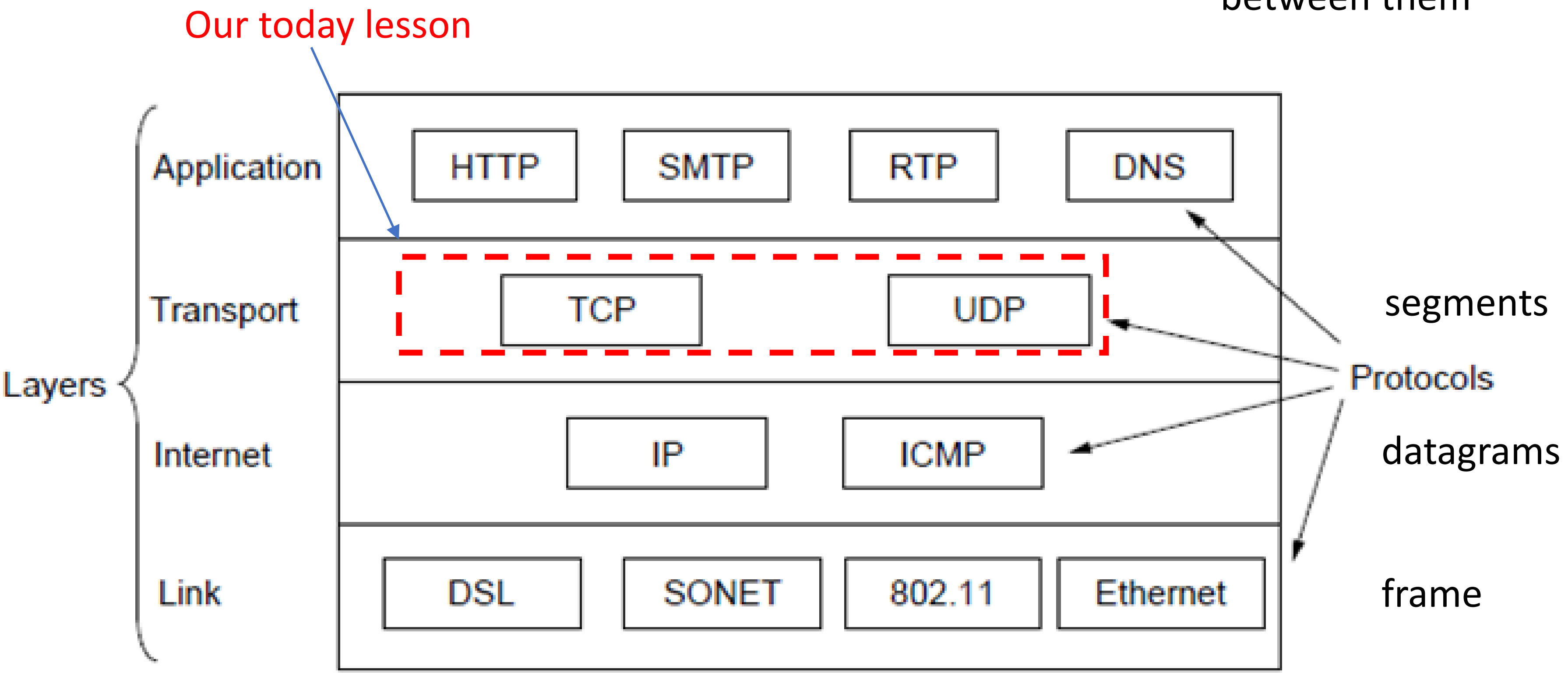
- source port
- destination port
- sequence number

https://blog.bytebytego.com/

TCP/IP Model is more popular in industry than OSI
OSI is for academic research

# End-to-end Protocols

The transport layer has responsibility for establishing a temporary communication session between two applications and delivering data between them



https://www.dcs.bbk.ac.uk/~ptw/teaching/IWT/transport-layer/notes.html

# Message encapsulation through layers

Web

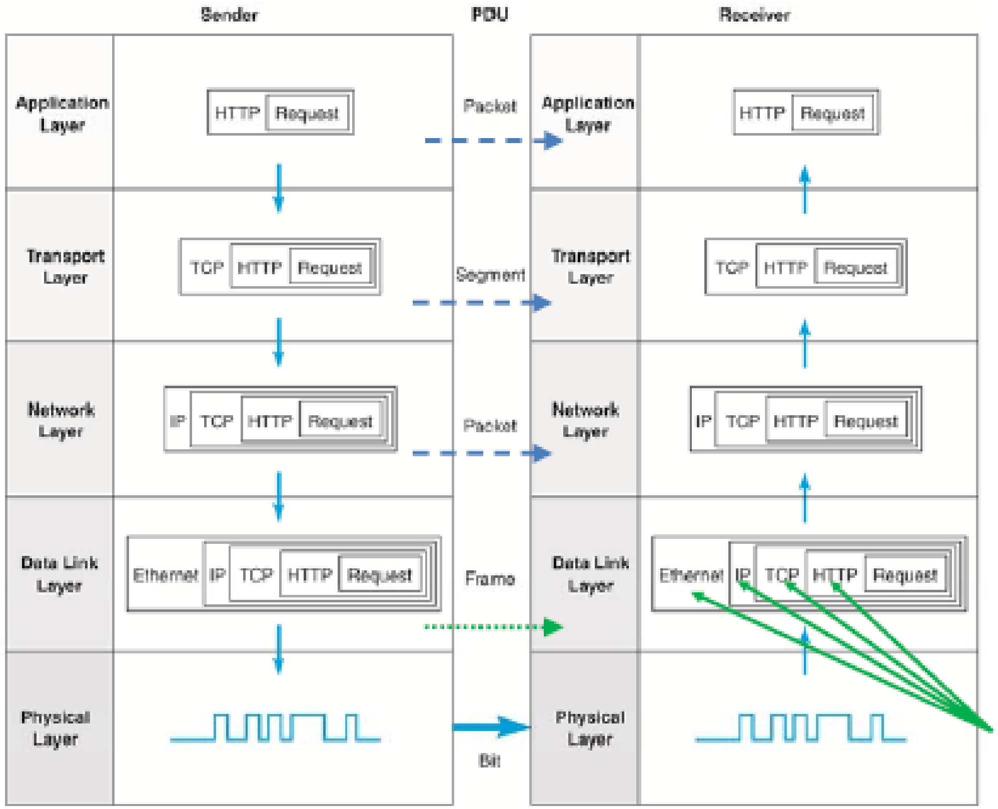


Fig. 1: Message transmission through layers.

- The application Layer uses the HTTP protocol
- The transport layer uses the TCP (Transmission Control) Protocol
- The network layer uses The IP (Internet) Protocol
- The data link layer of the LANs typically uses the Ethernet protocol
- Protocol Data Units (PDUs)

# End-to-end Protocols

- Below the presentation and session layer is transport layer

- A transport protocol can be expected to provide
  - Guarantees message delivery
  - Delivers messages in the same order they were sent
  - Delivers at most one copy of each message
  - Supports arbitrarily large messages
  - Supports synchronization between the sender and the receiver
  - Allows the receiver to apply flow control to the sender
  - Supports multiple application processes on each host

# End-to-end Protocols

- Typical limitations of the network on which transport protocol will operate
  - Drop messages
  - Reorder messages
  - Deliver duplicate copies of a given message
  - Limit messages to some finite size
  - Deliver messages after an arbitrarily long delay

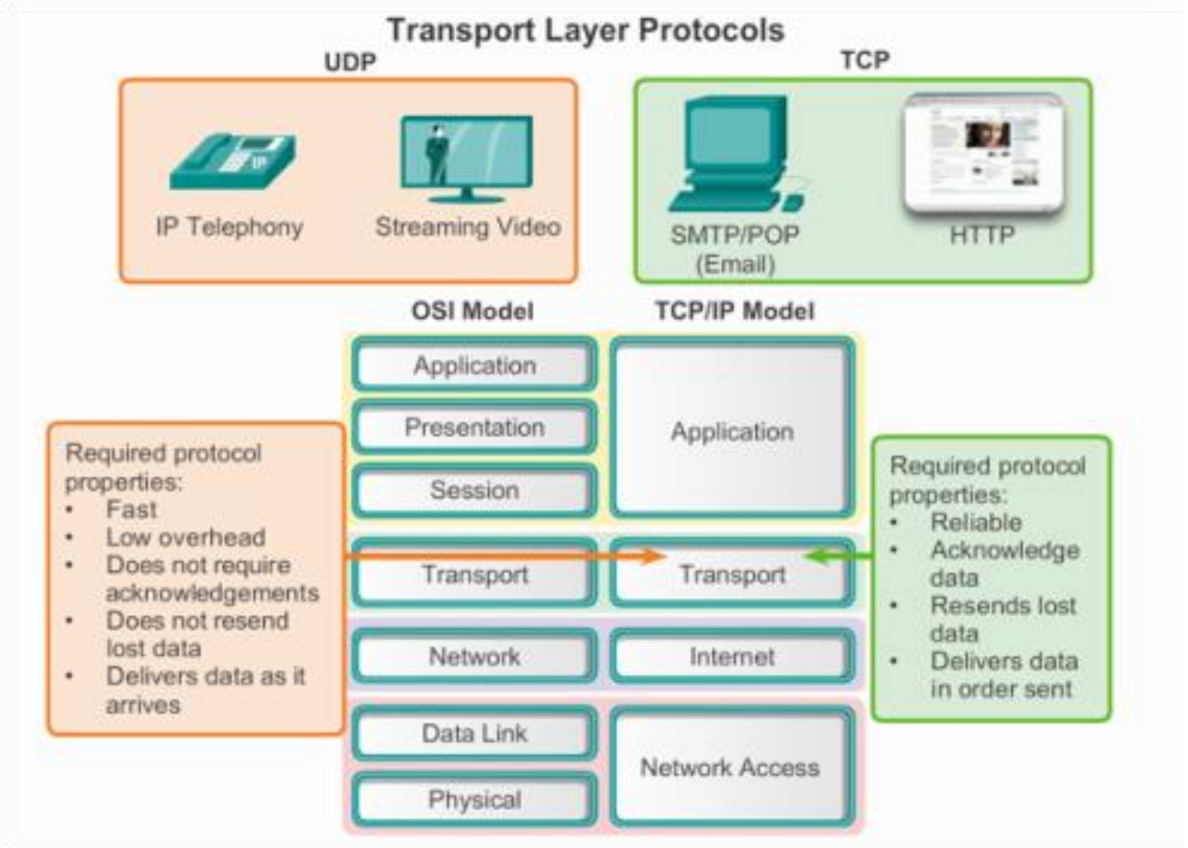
# End-to-end Protocols

- Challenge for Transport Protocols
  - Develop algorithms that turn the less-than-desirable properties of the underlying network into the high level of service required by application programs

- Connectionless transport layer
  1. Treat each packet as an individual and delivery to the destination
  2. The receiver doesn’t send acknowledgement of the packet
- Connection oriented transport layer
  1. Establish connection between the sender and the receiver before transmitting data
  2. The receiver sends acknowledgement for each received packet

# TCP vs UDP

- Two protocols are used
  1. Connectionless: User Datagram Protocol (UDP)
  2. Connection oriented: Transmission Control Protocol (TCP)



# User Datagram Protocol (UDP)

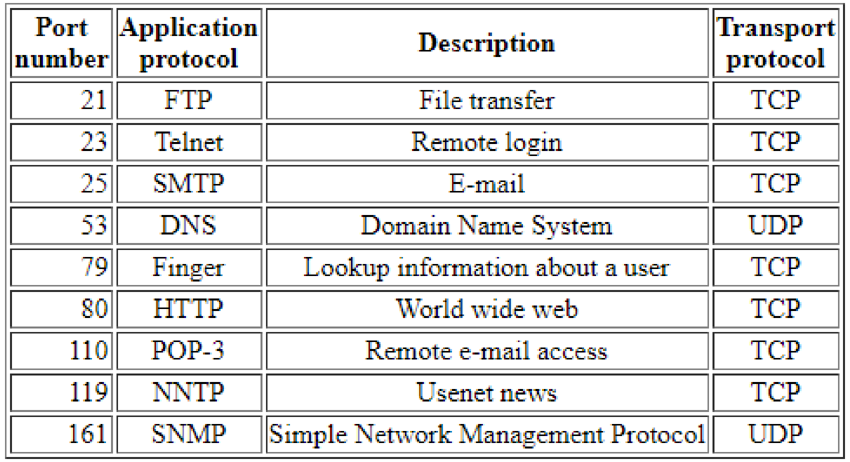
- UDP is suitable for those applications where data loss doesn’t affect the perceived quality of the service

- Multiplayer Online Games, VoIP, and Video Live Streaming

- Fast but non-guaranteed transfer. It is also called “best effort” transfer

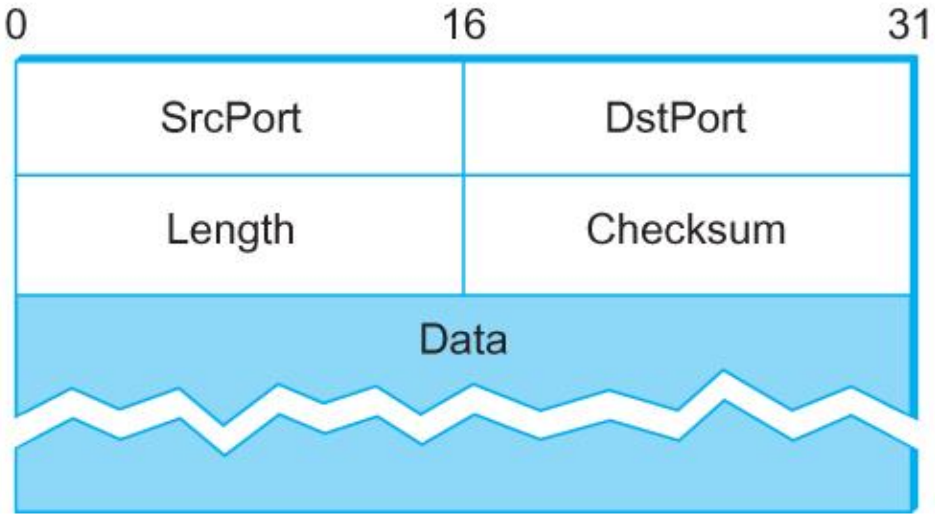
# UDP Datagram (header)

- Each UDP segment must include UDP header fields identifying the socket connection
- These header fields are the *source port number field* and the *destination port number field*
- Each port number is a 16-bit number: 0 to 65535
- Port numbers below 1024 are called *well-known ports* and are reserved for standard services



| Port number | Application protocol | Description | Transport protocol |
|---|---|---|---|
| 21 | FTP | File transfer | TCP |
| 23 | Telnet | Remote login | TCP |
| 25 | SMTP | E-mail | TCP |
| 53 | DNS | Domain Name System | UDP |
| 79 | Finger | Lookup information about a user | TCP |
| 80 | HTTP | World wide web | TCP |
| 110 | POP-3 | Remote e-mail access | TCP |
| 119 | NNTP | Usenet news | TCP |
| 161 | SNMP | Simple Network Management Protocol | UDP |

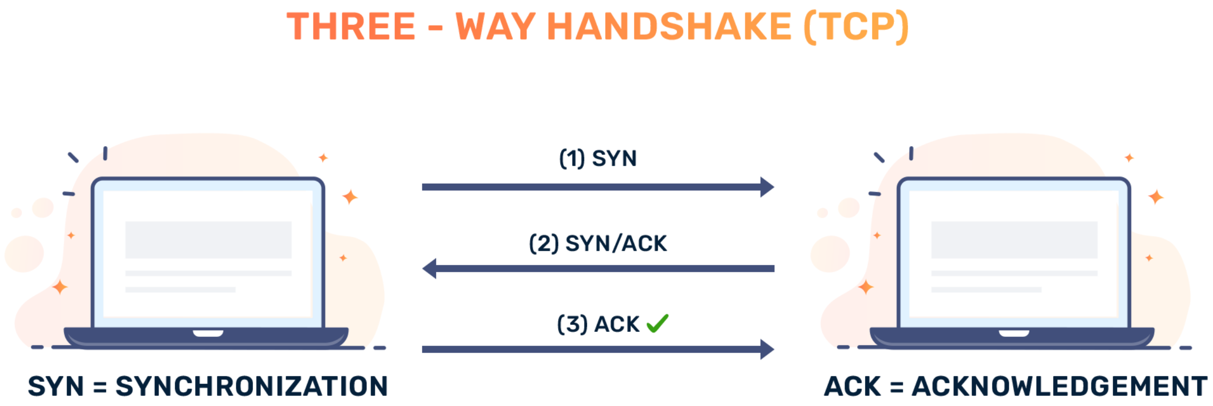
Famous ports for some services



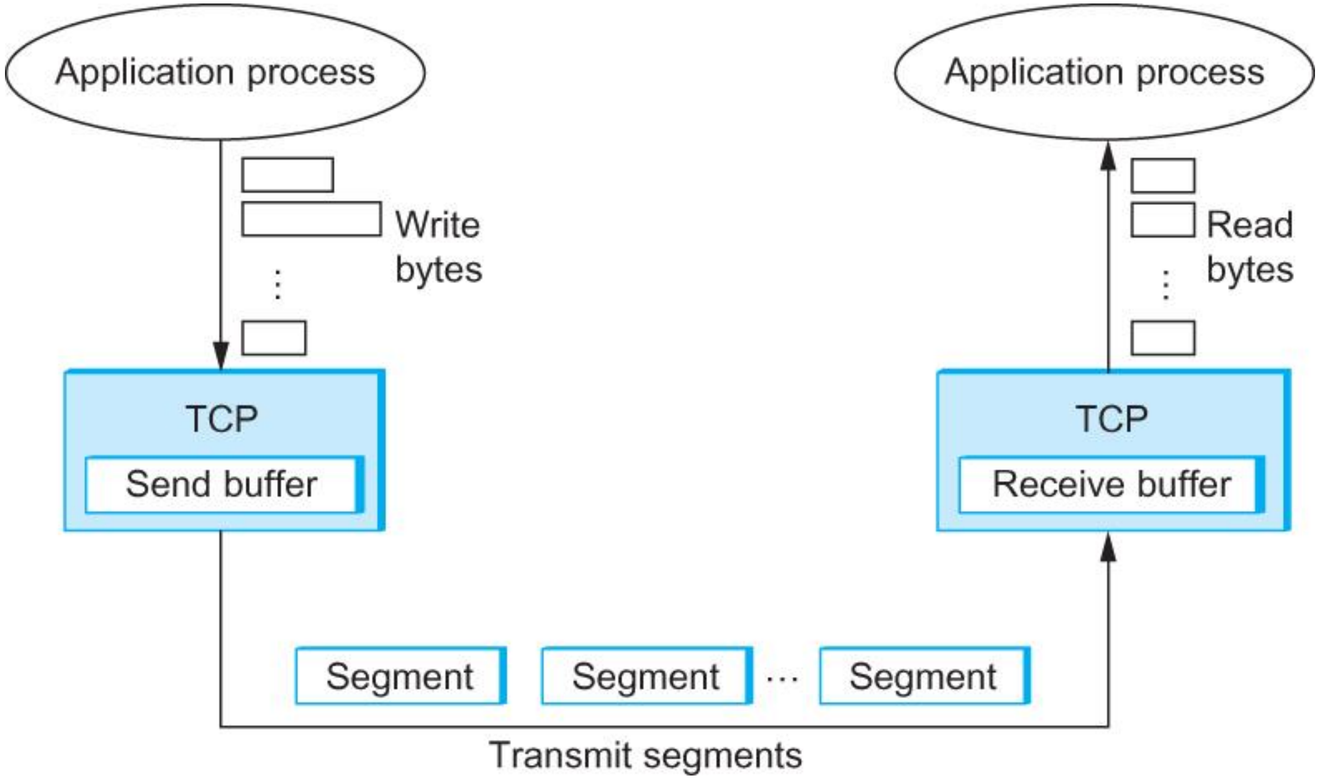
Format for UDP header

# Transmission Control Protocol (TCP)

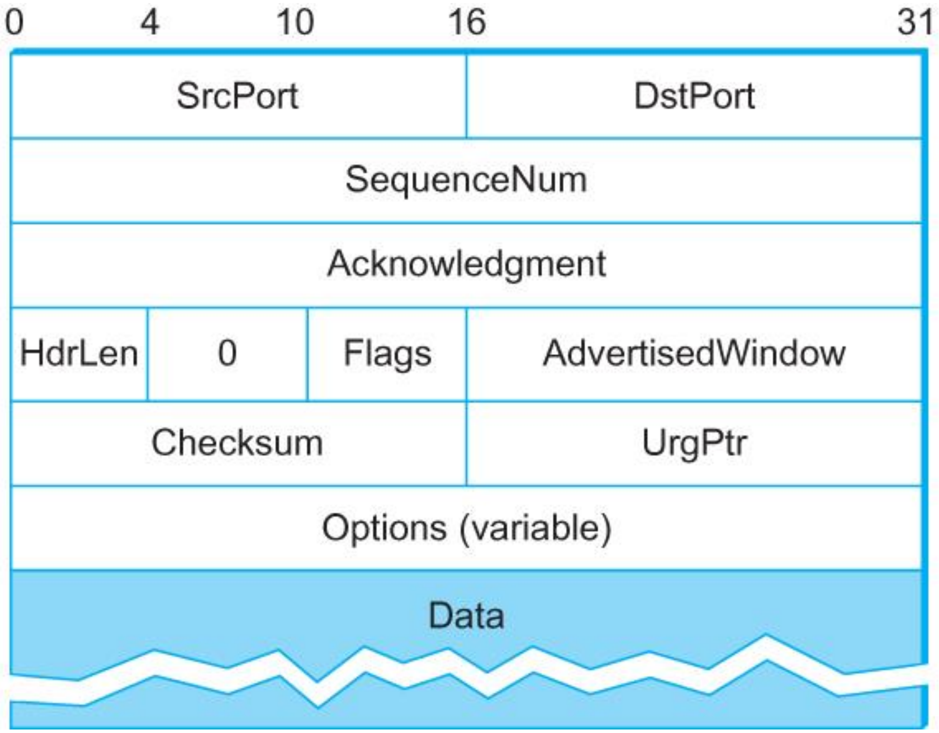
- In contrast to UDP, Transmission Control Protocol (TCP) offers the following services
  - Reliable
  - Connection oriented
  - Byte-stream service
- TCP is also known as a three-way handshake protocol



# TCP Segment & TCP Header



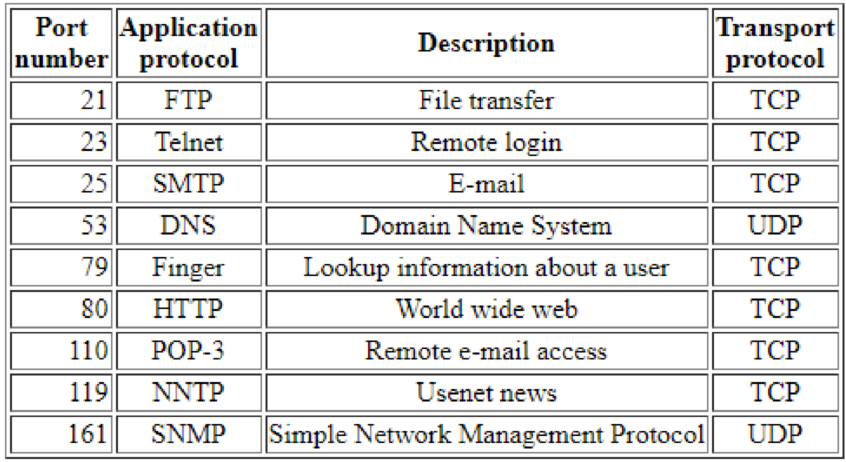
How TCP manages a byte stream.



TCP Header Format

# TCP Header

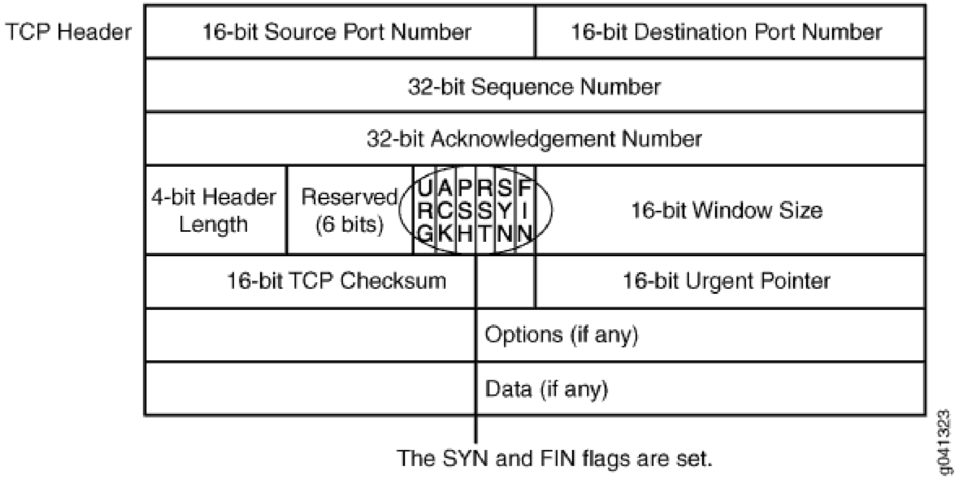
- The **SrcPort** and **DstPort**: identify the source and destination ports, respectively.
- The **Acknowledgment**, **SequenceNum**, and **AdvertisedWindow**: involved in TCP’s sliding window algorithm.
- Because TCP is a byte-oriented protocol, each byte of data has a sequence number; the **SequenceNum** field contains the sequence number for the first byte of data carried in that segment.
- The **Acknowledgment** and **AdvertisedWindow** fields carry information about the flow of data going in the other direction.



| Port number | Application protocol | Description | Transport protocol |
|---|---|---|---|
| 21 | FTP | File transfer | TCP |
| 23 | Telnet | Remote login | TCP |
| 25 | SMTP | E-mail | TCP |
| 53 | DNS | Domain Name System | UDP |
| 79 | Finger | Lookup information about a user | TCP |
| 80 | HTTP | World wide web | TCP |
| 110 | POP-3 | Remote e-mail access | TCP |
| 119 | NNTP | Usenet news | TCP |
| 161 | SNMP | Simple Network Management Protocol | UDP |

# TCP Header

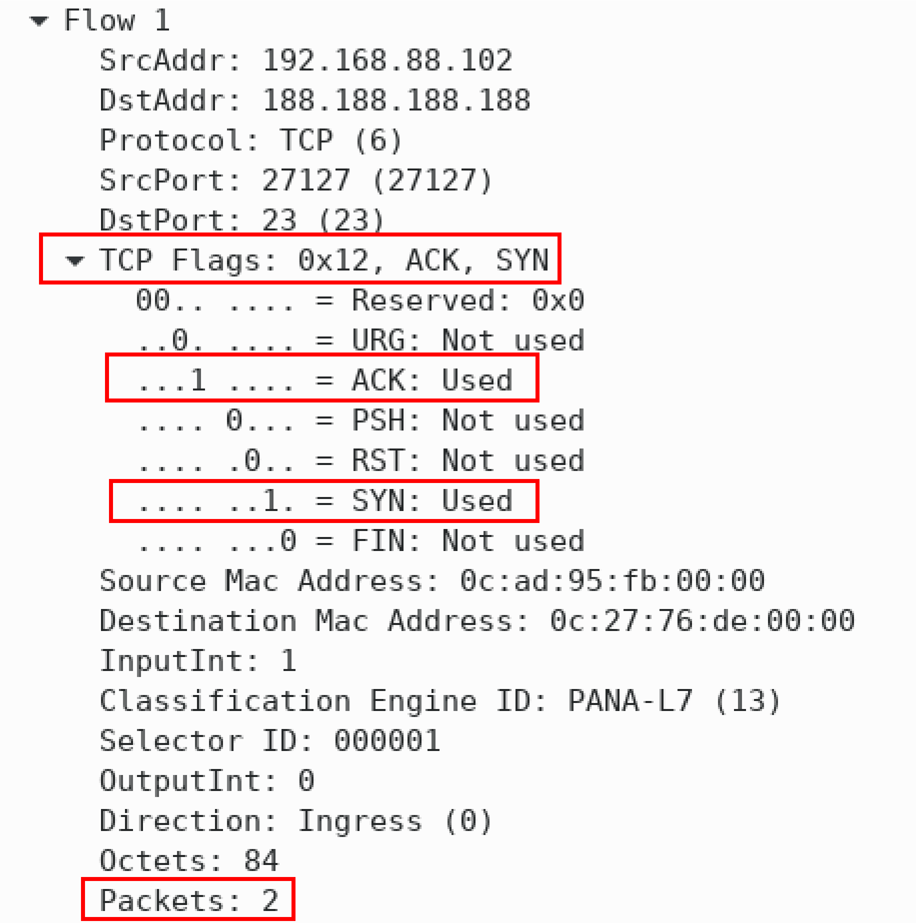
- The 6-bit Flags field is used to relay control information between TCP peers.
- The possible flags include SYN, FIN, RESET, PUSH, URG, and ACK.
- The SYN and FIN flags are used when establishing and terminating a TCP connection, respectively.
- The ACK flag is set any time the Acknowledgment field is valid, implying that the receiver should pay attention to it.



<table>
<tr><td colspan="3">16-bit Source Port Number</td><td>16-bit Destination Port Number</td></tr>
<tr><td colspan="4">32-bit Sequence Number</td></tr>
<tr><td colspan="4">32-bit Acknowledgement Number</td></tr>
<tr><td>4-bit Header Length</td><td>Reserved (6 bits)</td><td>URG<br>ACK<br>PSH<br>RST<br>SYN<br>FIN</td><td>16-bit Window Size</td></tr>
<tr><td colspan="2">16-bit TCP Checksum</td><td></td><td>16-bit Urgent Pointer</td></tr>
<tr><td colspan="2"></td><td colspan="2">Options (if any)</td></tr>
<tr><td colspan="2"></td><td colspan="2">Data (if any)</td></tr>
</table>

# TCP Header

- The **URG** flag signifies that this segment contains urgent data. When this flag is set, the UrgPtr field indicates where the nonurgent data contained in this segment begins.
-
- The urgent data is contained at the front of the segment body, up to and including a value of **UrgPtr** bytes into the segment.
- The **PUSH** flag signifies that the sender invoked the push operation, which indicates to the receiving side of TCP that it should notify the receiving process of this fact.
- Finally, the **RESET** flag signifies that the receiver has become confused

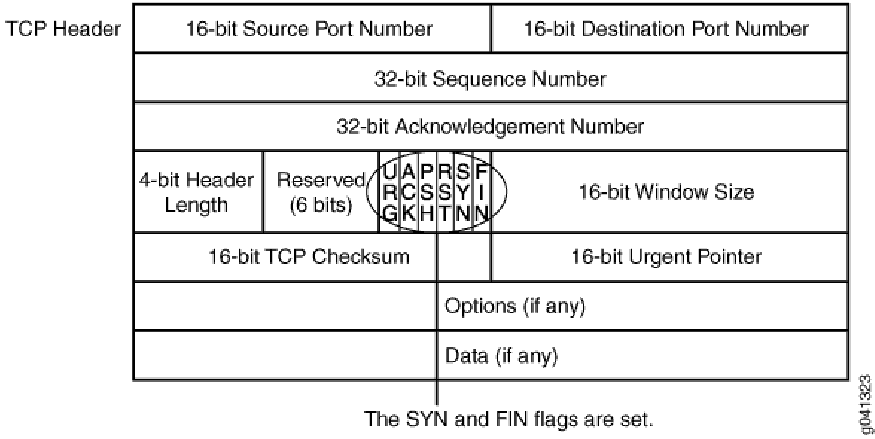


```
▾ Flow 1
    SrcAddr: 192.168.88.102
    DstAddr: 188.188.188.188
    Protocol: TCP (6)
    SrcPort: 27127 (27127)
    DstPort: 23 (23)
  ▾ TCP Flags: 0x12, ACK, SYN
      00.. .... = Reserved: 0x0
      ..0. .... = URG: Not used
      ...1 .... = ACK: Used
      .... 0... = PSH: Not used
      .... .0.. = RST: Not used
      .... ..1. = SYN: Used
      .... ...0 = FIN: Not used
    Source Mac Address: 0c:ad:95:fb:00:00
    Destination Mac Address: 0c:27:76:de:00:00
    InputInt: 1
    Classification Engine ID: PANA-L7 (13)
    Selector ID: 000001
    OutputInt: 0
    Direction: Ingress (0)
    Octets: 84
    Packets: 2
```

# TCP Header

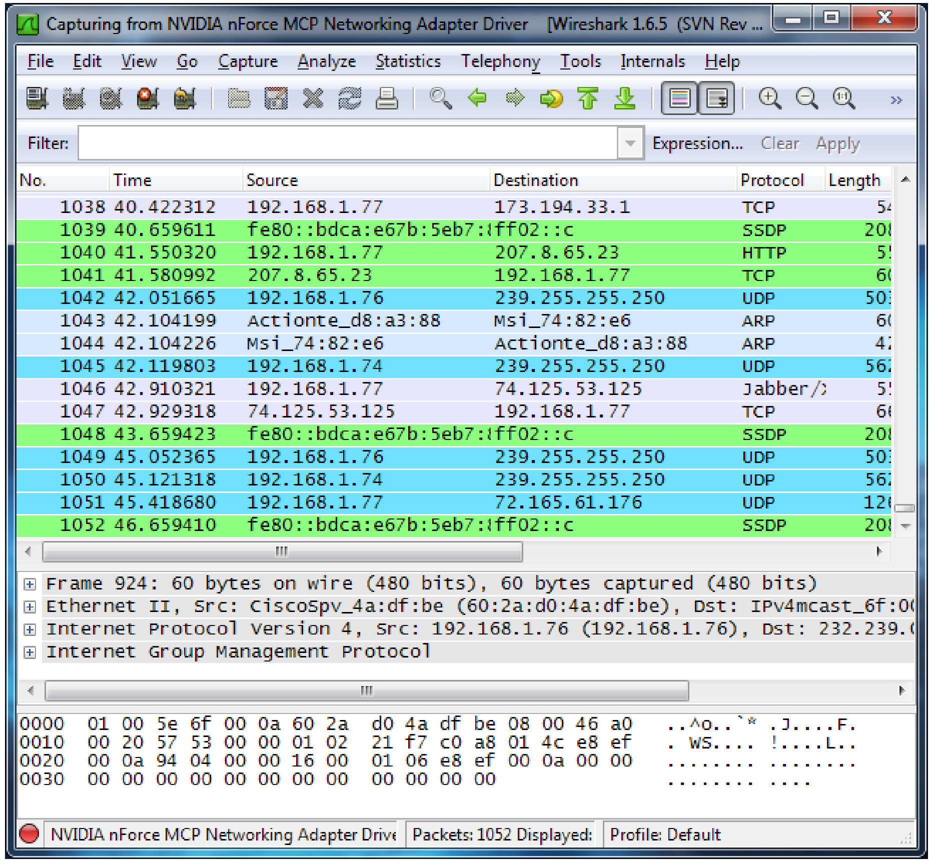
- Finally, the RESET flag signifies that the receiver has become confused, it received a segment it did not expect to receive—and so wants to abort the connection.

- Finally, the Checksum field is used in exactly the same way as for UDP—it is computed over the TCP header, the TCP data, and the pseudo header, which is made up of the source address, destination address, and length fields from the IP header → Validate the packet integrity



# Packet Analysis in Wireshark

- A free and open-source packet analyzer for network troubleshooting, analysis, software and communications protocol development, and education.



| No. | Time | Source | Destination | Protocol | Length |
|---|---|---|---|---|---|
| 1038 | 40.422312 | 192.168.1.77 | 173.194.33.1 | TCP | 54 |
| 1039 | 40.659611 | fe80::bdca:e67b:5eb7:8 | ff02::c | SSDP | 208 |
| 1040 | 41.550320 | 192.168.1.77 | 207.8.65.23 | HTTP | 5! |
| 1041 | 41.580992 | 207.8.65.23 | 192.168.1.77 | TCP | 60 |
| 1042 | 42.051665 | 192.168.1.76 | 239.255.255.250 | UDP | 50: |
| 1043 | 42.104199 | Actionte_d8:a3:88 | Msi_74:82:e6 | ARP | 60 |
| 1044 | 42.104226 | Msi_74:82:e6 | Actionte_d8:a3:88 | ARP | 4: |
| 1045 | 42.119803 | 192.168.1.74 | 239.255.255.250 | UDP | 56: |
| 1046 | 42.910321 | 192.168.1.77 | 74.125.53.125 | Jabber/ | 5! |
| 1047 | 42.929318 | 74.125.53.125 | 192.168.1.77 | TCP | 66 |
| 1048 | 43.659423 | fe80::bdca:e67b:5eb7:8 | ff02::c | SSDP | 208 |
| 1049 | 45.052365 | 192.168.1.76 | 239.255.255.250 | UDP | 50: |
| 1050 | 45.121318 | 192.168.1.74 | 239.255.255.250 | UDP | 56: |
| 1051 | 45.418680 | 192.168.1.77 | 72.165.61.176 | UDP | 126 |
| 1052 | 46.659410 | fe80::bdca:e67b:5eb7:8 | ff02::c | SSDP | 208 |

```
⊞ Frame 924: 60 bytes on wire (480 bits), 60 bytes captured (480 bits)
⊞ Ethernet II, Src: CiscoSpv_4a:df:be (60:2a:d0:4a:df:be), Dst: IPv4mcast_6f:00
⊞ Internet Protocol Version 4, Src: 192.168.1.76 (192.168.1.76), Dst: 232.239.0
⊞ Internet Group Management Protocol
```

```
0000  01 00 5e 6f 00 0a 60 2a  d0 4a df be 08 00 46 a0   ..^o..`* .J....F.
0010  00 20 57 53 00 00 01 02  21 f7 c0 a8 01 4c e8 ef   . WS.... !....L..
0020  00 0a 94 04 00 00 16 00  01 06 e8 ef 00 0a 00 00   ........ ........
0030  00 00 00 00 00 00 00 00  00 00 00 00               ........ ....
```

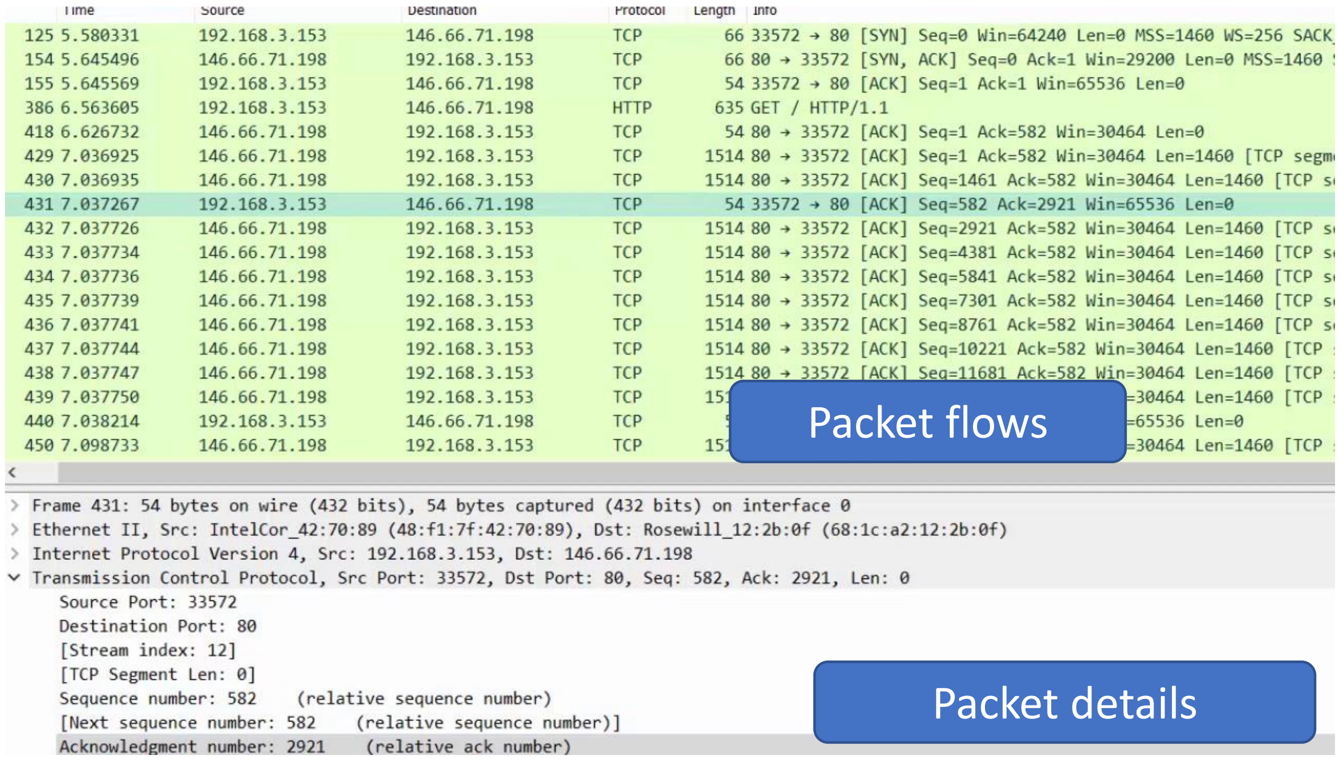
NVIDIA nForce MCP Networking Adapter Drive | Packets: 1052 Displayed: | Profile: Default

https://www.wireshark.org/

# Wireshark

- Allow us to check TCP/UDP packet in details

- Statistical data
  - ✓ Packet flow (src→ dest)
  - ✓ Endpoints
  - ✓ Packet lengths
  - ✓ I/O graphs



```
    Time        Source          Destination     Protocol  Length Info
125 5.580331    192.168.3.153   146.66.71.198   TCP         66 33572 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_
154 5.645496    146.66.71.198   192.168.3.153   TCP         66 80 → 33572 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460 :
155 5.645569    192.168.3.153   146.66.71.198   TCP         54 33572 → 80 [ACK] Seq=1 Ack=1 Win=65536 Len=0
386 6.563605    192.168.3.153   146.66.71.198   HTTP       635 GET / HTTP/1.1
418 6.626732    146.66.71.198   192.168.3.153   TCP         54 80 → 33572 [ACK] Seq=1 Ack=582 Win=30464 Len=0
429 7.036925    146.66.71.198   192.168.3.153   TCP       1514 80 → 33572 [ACK] Seq=1 Ack=582 Win=30464 Len=1460 [TCP segme
430 7.036935    146.66.71.198   192.168.3.153   TCP       1514 80 → 33572 [ACK] Seq=1461 Ack=582 Win=30464 Len=1460 [TCP se
431 7.037267    192.168.3.153   146.66.71.198   TCP         54 33572 → 80 [ACK] Seq=582 Ack=2921 Win=65536 Len=0
432 7.037726    146.66.71.198   192.168.3.153   TCP       1514 80 → 33572 [ACK] Seq=2921 Ack=582 Win=30464 Len=1460 [TCP se
433 7.037734    146.66.71.198   192.168.3.153   TCP       1514 80 → 33572 [ACK] Seq=4381 Ack=582 Win=30464 Len=1460 [TCP se
434 7.037736    146.66.71.198   192.168.3.153   TCP       1514 80 → 33572 [ACK] Seq=5841 Ack=582 Win=30464 Len=1460 [TCP se
435 7.037739    146.66.71.198   192.168.3.153   TCP       1514 80 → 33572 [ACK] Seq=7301 Ack=582 Win=30464 Len=1460 [TCP se
436 7.037741    146.66.71.198   192.168.3.153   TCP       1514 80 → 33572 [ACK] Seq=8761 Ack=582 Win=30464 Len=1460 [TCP se
437 7.037744    146.66.71.198   192.168.3.153   TCP       1514 80 → 33572 [ACK] Seq=10221 Ack=582 Win=30464 Len=1460 [TCP :
438 7.037747    146.66.71.198   192.168.3.153   TCP       1514 80 → 33572 [ACK] Seq=11681 Ack=582 Win=30464 Len=1460 [TCP :
439 7.037750    146.66.71.198   192.168.3.153   TCP       151                                  =30464 Len=1460 [TCP :
440 7.038214    192.168.3.153   146.66.71.198   TCP         5                                  =65536 Len=0
450 7.098733    146.66.71.198   192.168.3.153   TCP       151                                  =30464 Len=1460 [TCP :

> Frame 431: 54 bytes on wire (432 bits), 54 bytes captured (432 bits) on interface 0
> Ethernet II, Src: IntelCor_42:70:89 (48:f1:7f:42:70:89), Dst: Rosewill_12:2b:0f (68:1c:a2:12:2b:0f)
> Internet Protocol Version 4, Src: 192.168.3.153, Dst: 146.66.71.198
v Transmission Control Protocol, Src Port: 33572, Dst Port: 80, Seq: 582, Ack: 2921, Len: 0
    Source Port: 33572
    Destination Port: 80
    [Stream index: 12]
    [TCP Segment Len: 0]
    Sequence number: 582    (relative sequence number)
    [Next sequence number: 582    (relative sequence number)]
    Acknowledgment number: 2921    (relative ack number)
```

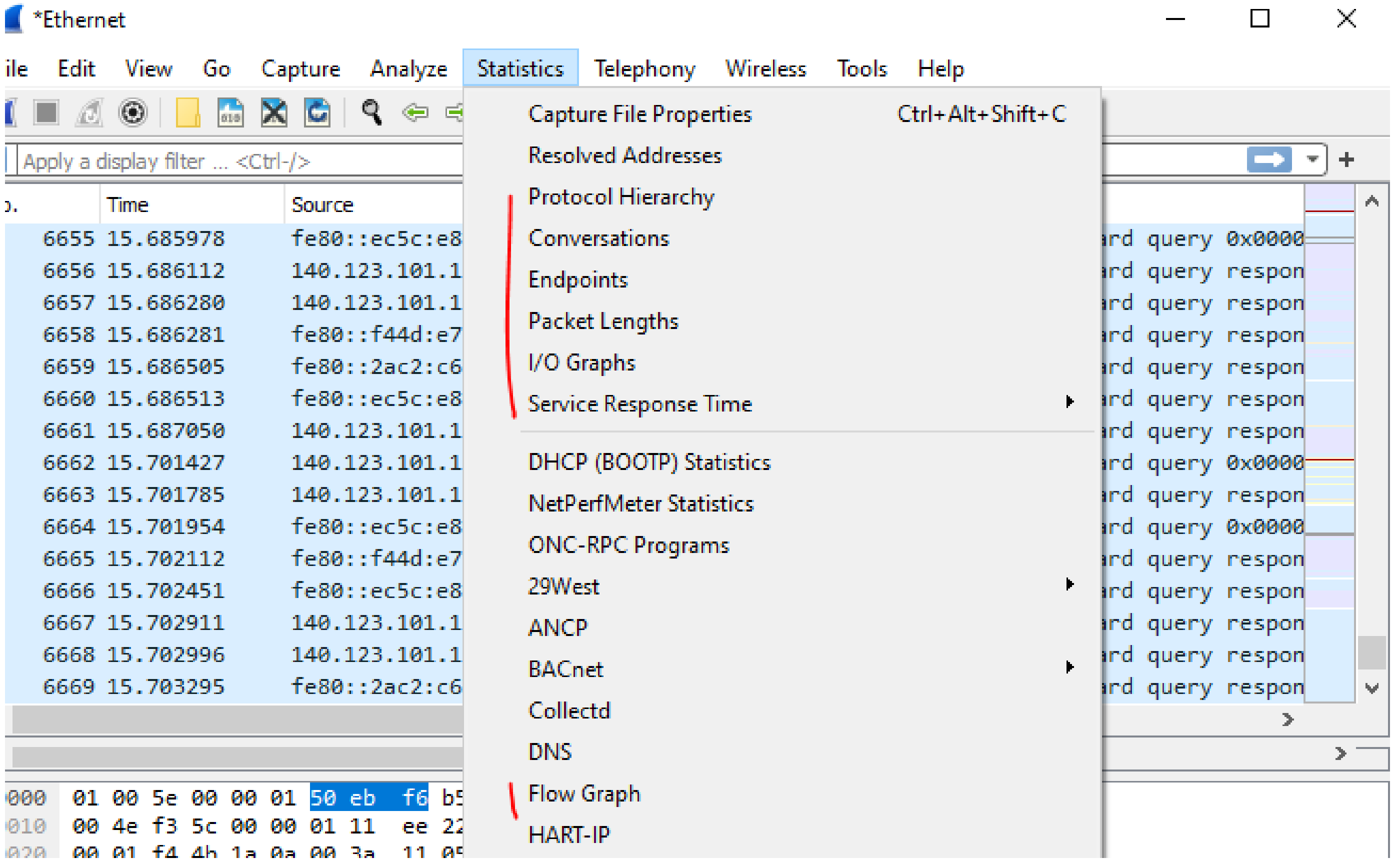
Packet flows

Packet details

# Wireshark

- Allow us to check TCP/UDP packet in details

- Statistic feature
  - ✓ Packet flow (src→ dest)
  - ✓ Endpoints
  - ✓ Packet lengths
  - ✓ I/O graphs



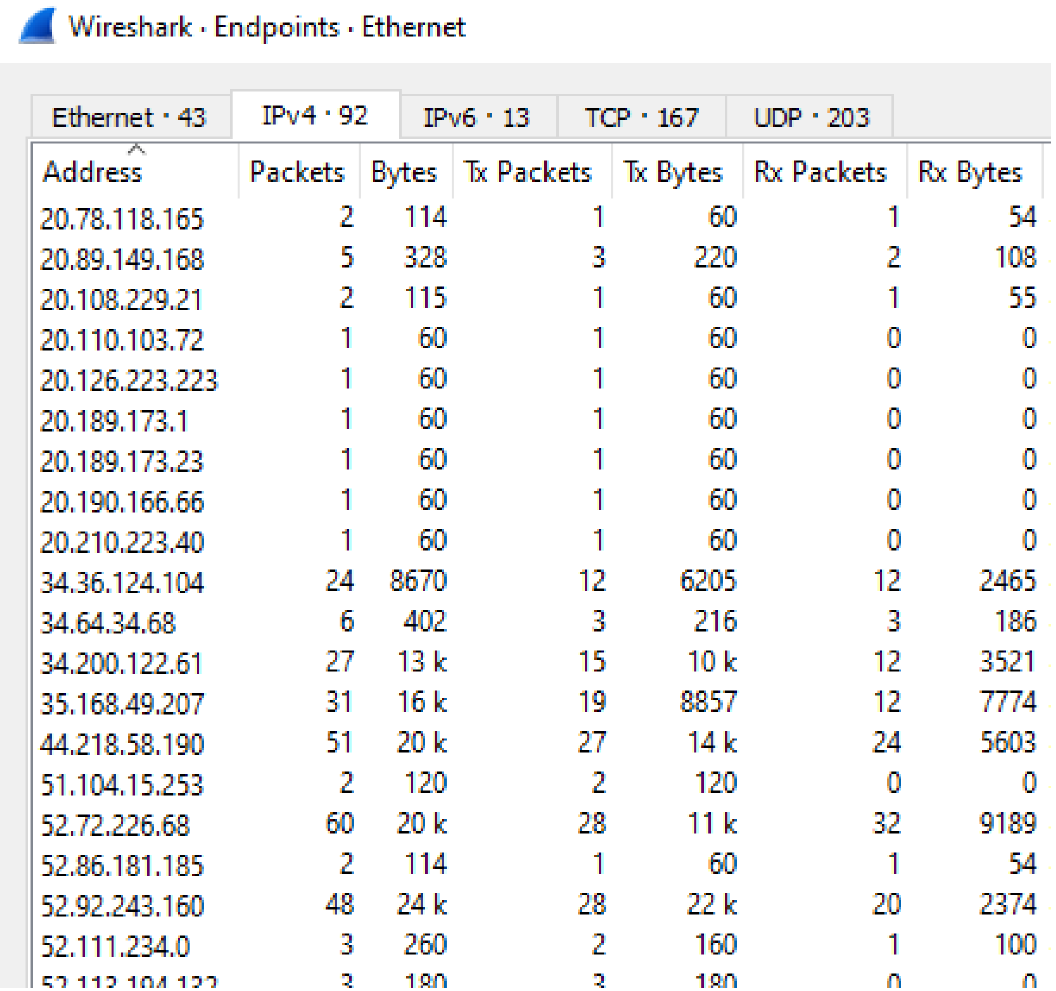
```
o.    Time       Source                 
 6655 15.685978  fe80::ec5c:e8    rd query 0x0000
 6656 15.686112  140.123.101.1    rd query respon
 6657 15.686280  140.123.101.1    rd query respon
 6658 15.686281  fe80::f44d:e7    rd query respon
 6659 15.686505  fe80::2ac2:c6    rd query respon
 6660 15.686513  fe80::ec5c:e8    rd query respon
 6661 15.687050  140.123.101.1    rd query respon
 6662 15.701427  140.123.101.1    rd query 0x0000
 6663 15.701785  140.123.101.1    rd query respon
 6664 15.701954  fe80::ec5c:e8    rd query 0x0000
 6665 15.702112  fe80::f44d:e7    rd query respon
 6666 15.702451  fe80::ec5c:e8    rd query respon
 6667 15.702911  140.123.101.1    rd query respon
 6668 15.702996  140.123.101.1    rd query respon
 6669 15.703295  fe80::2ac2:c6    rd query respon
```

```
000  01 00 5e 00 00 01 50 eb  f6 b5
010  00 4e f3 5c 00 00 01 11  ee 22
020  00 01 f4 4b 1a 0a 00 3a  11 05
```

# Wireshark

- Allow us to check TCP/UDP packet in details

- Statistic feature
  - ✓ Packet flow (src→ dest)
  - ✓ Endpoints
  - ✓ Packet lengths
  - ✓ I/O graphs

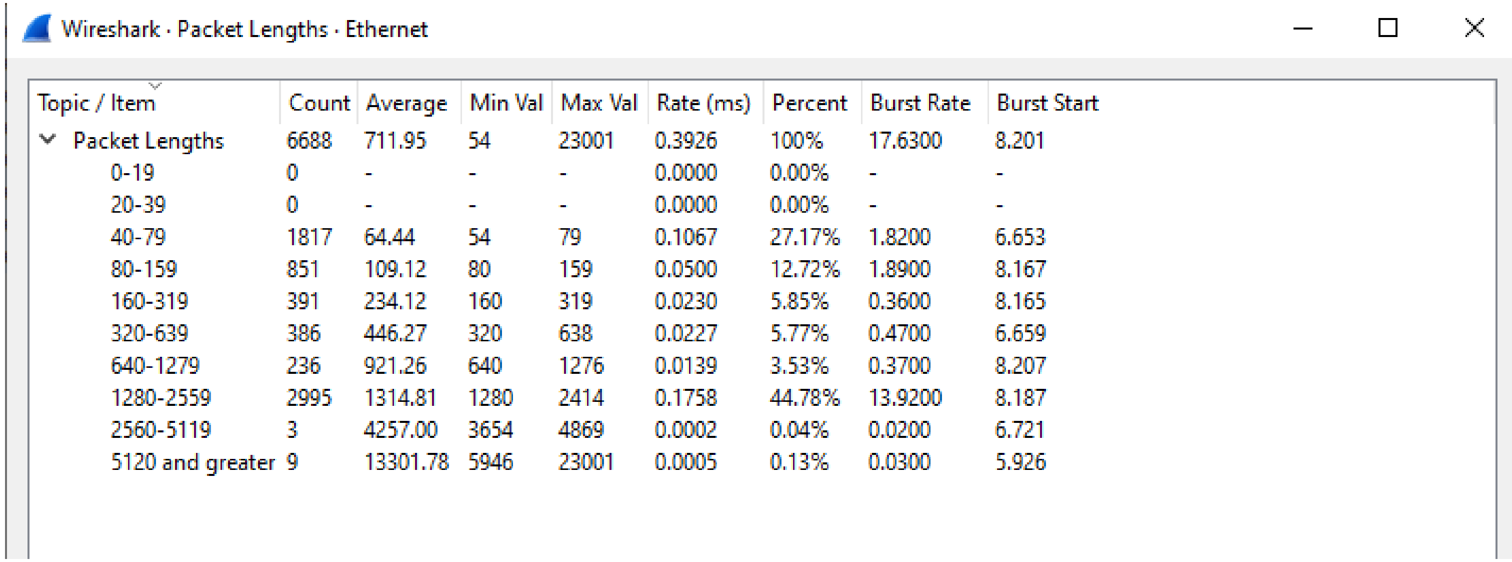


Wireshark · Endpoints · Ethernet

| Address | Packets | Bytes | Tx Packets | Tx Bytes | Rx Packets | Rx Bytes |
|---|---|---|---|---|---|---|
| 20.78.118.165 | 2 | 114 | 1 | 60 | 1 | 54 |
| 20.89.149.168 | 5 | 328 | 3 | 220 | 2 | 108 |
| 20.108.229.21 | 2 | 115 | 1 | 60 | 1 | 55 |
| 20.110.103.72 | 1 | 60 | 1 | 60 | 0 | 0 |
| 20.126.223.223 | 1 | 60 | 1 | 60 | 0 | 0 |
| 20.189.173.1 | 1 | 60 | 1 | 60 | 0 | 0 |
| 20.189.173.23 | 1 | 60 | 1 | 60 | 0 | 0 |
| 20.190.166.66 | 1 | 60 | 1 | 60 | 0 | 0 |
| 20.210.223.40 | 1 | 60 | 1 | 60 | 0 | 0 |
| 34.36.124.104 | 24 | 8670 | 12 | 6205 | 12 | 2465 |
| 34.64.34.68 | 6 | 402 | 3 | 216 | 3 | 186 |
| 34.200.122.61 | 27 | 13 k | 15 | 10 k | 12 | 3521 |
| 35.168.49.207 | 31 | 16 k | 19 | 8857 | 12 | 7774 |
| 44.218.58.190 | 51 | 20 k | 27 | 14 k | 24 | 5603 |
| 51.104.15.253 | 2 | 120 | 2 | 120 | 0 | 0 |
| 52.72.226.68 | 60 | 20 k | 28 | 11 k | 32 | 9189 |
| 52.86.181.185 | 2 | 114 | 1 | 60 | 1 | 54 |
| 52.92.243.160 | 48 | 24 k | 28 | 22 k | 20 | 2374 |
| 52.111.234.0 | 3 | 260 | 2 | 160 | 1 | 100 |
| 52.113.104.122 | 3 | 180 | 3 | 180 | 0 | 0 |

# Wireshark

- Allow us to check TCP/UDP packet in details
- Statistic feature
  - ✓ Packet flow (src→ dest)
  - ✓ Endpoints
  - ✓ Packet lengths
  - ✓ I/O graphs



Wireshark · Packet Lengths · Ethernet

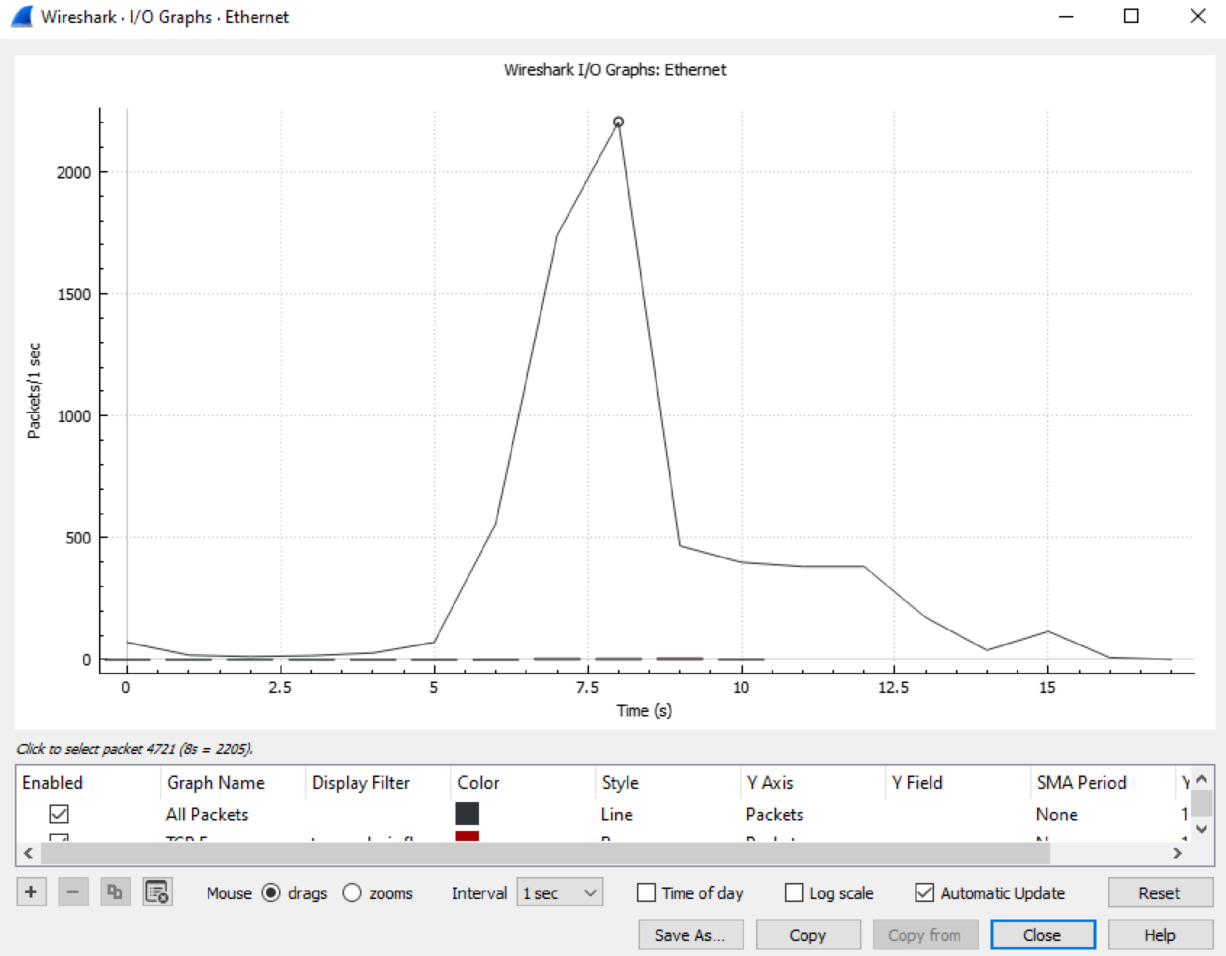
| Topic / Item | Count | Average | Min Val | Max Val | Rate (ms) | Percent | Burst Rate | Burst Start |
|---|---|---|---|---|---|---|---|---|
| Packet Lengths | 6688 | 711.95 | 54 | 23001 | 0.3926 | 100% | 17.6300 | 8.201 |
| 0-19 | 0 | - | - | - | 0.0000 | 0.00% | - | - |
| 20-39 | 0 | - | - | - | 0.0000 | 0.00% | - | - |
| 40-79 | 1817 | 64.44 | 54 | 79 | 0.1067 | 27.17% | 1.8200 | 6.653 |
| 80-159 | 851 | 109.12 | 80 | 159 | 0.0500 | 12.72% | 1.8900 | 8.167 |
| 160-319 | 391 | 234.12 | 160 | 319 | 0.0230 | 5.85% | 0.3600 | 8.165 |
| 320-639 | 386 | 446.27 | 320 | 638 | 0.0227 | 5.77% | 0.4700 | 6.659 |
| 640-1279 | 236 | 921.26 | 640 | 1276 | 0.0139 | 3.53% | 0.3700 | 8.207 |
| 1280-2559 | 2995 | 1314.81 | 1280 | 2414 | 0.1758 | 44.78% | 13.9200 | 8.187 |
| 2560-5119 | 3 | 4257.00 | 3654 | 4869 | 0.0002 | 0.04% | 0.0200 | 6.721 |
| 5120 and greater | 9 | 13301.78 | 5946 | 23001 | 0.0005 | 0.13% | 0.0300 | 5.926 |

# Wireshark

- Allow us to check TCP/UDP packet in details

- Statistic data
  - ✓ Packet flow (src→ dest)
  - ✓ Endpoints
  - ✓ Packet lengths
  - ✓ I/O graphs

I/O graphs

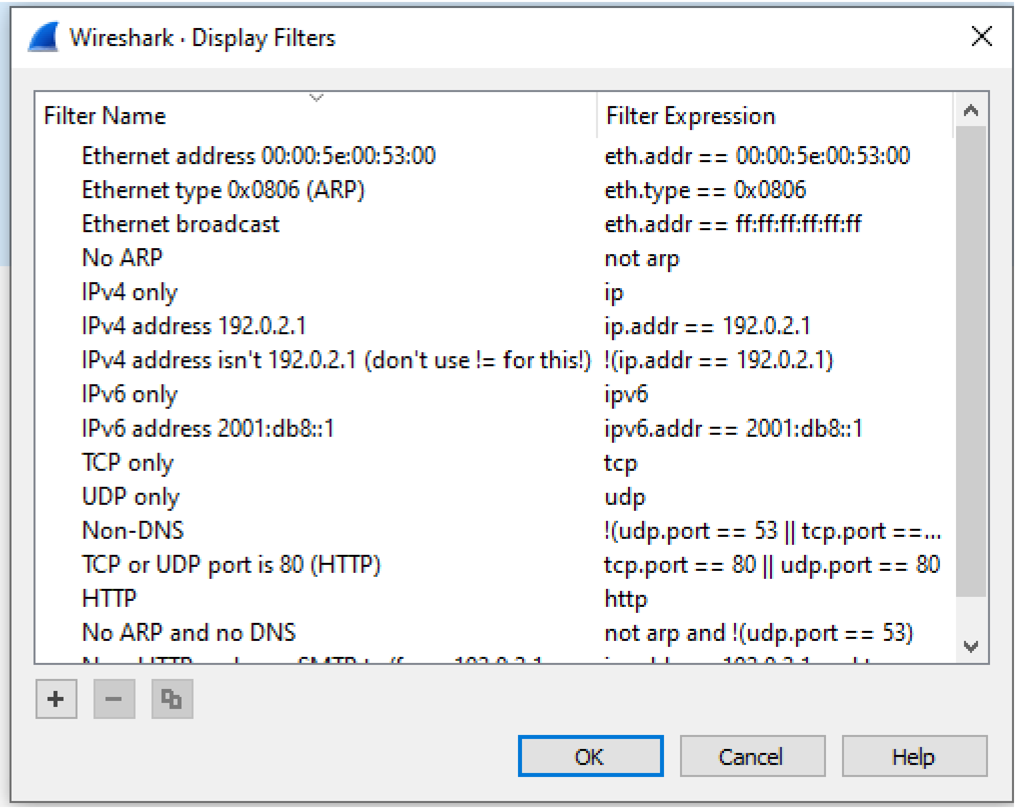

Wireshark · I/O Graphs · Ethernet
Wireshark I/O Graphs: Ethernet
Packets/1 sec
0
500
1000
1500
2000
0
2.5
5
7.5
10
12.5
15
Time (s)

*Click to select packet 4721 (8s = 2205).*

| Enabled | Graph Name | Display Filter | Color | Style | Y Axis | Y Field | SMA Period | Y |
|---|---|---|---|---|---|---|---|---|
| ☑ | All Packets | | ■ | Line | Packets | | None | 1 |

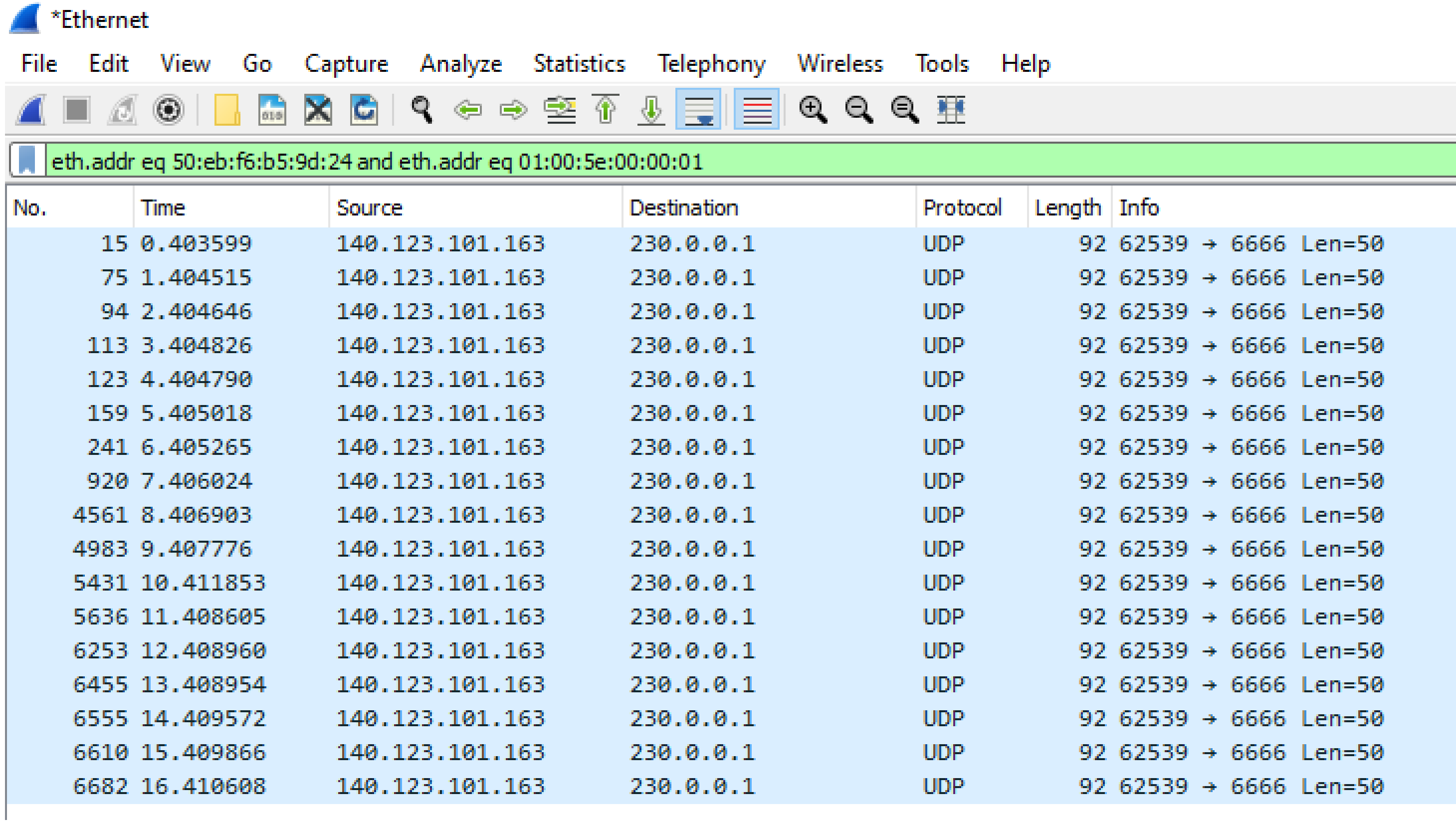
# Packet flow filter in Wireshark

- Find all packets of a flow



Wireshark · Display Filters

| Filter Name | Filter Expression |
|---|---|
| Ethernet address 00:00:5e:00:53:00 | eth.addr == 00:00:5e:00:53:00 |
| Ethernet type 0x0806 (ARP) | eth.type == 0x0806 |
| Ethernet broadcast | eth.addr == ff:ff:ff:ff:ff:ff |
| No ARP | not arp |
| IPv4 only | ip |
| IPv4 address 192.0.2.1 | ip.addr == 192.0.2.1 |
| IPv4 address isn't 192.0.2.1 (don't use != for this!) | !(ip.addr == 192.0.2.1) |
| IPv6 only | ipv6 |
| IPv6 address 2001:db8::1 | ipv6.addr == 2001:db8::1 |
| TCP only | tcp |
| UDP only | udp |
| Non-DNS | !(udp.port == 53 \|\| tcp.port ==... |
| TCP or UDP port is 80 (HTTP) | tcp.port == 80 \|\| udp.port == 80 |
| HTTP | http |
| No ARP and no DNS | not arp and !(udp.port == 53) |



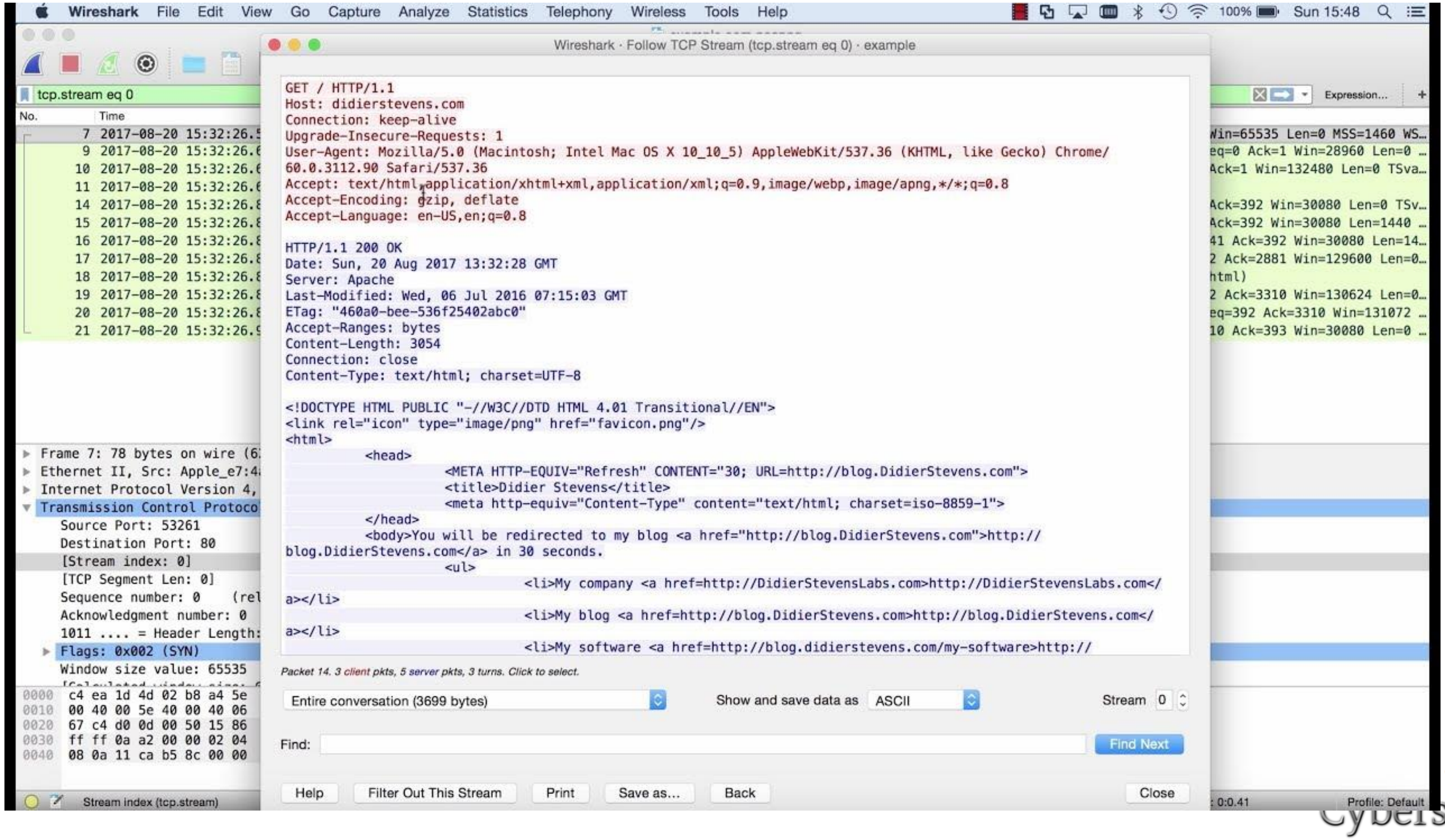
*Ethernet

```
eth.addr eq 50:eb:f6:b5:9d:24 and eth.addr eq 01:00:5e:00:00:01
```

```
No.   Time       Source           Destination  Protocol Length Info
   15 0.403599   140.123.101.163  230.0.0.1    UDP      92 62539 → 6666 Len=50
   75 1.404515   140.123.101.163  230.0.0.1    UDP      92 62539 → 6666 Len=50
   94 2.404646   140.123.101.163  230.0.0.1    UDP      92 62539 → 6666 Len=50
  113 3.404826   140.123.101.163  230.0.0.1    UDP      92 62539 → 6666 Len=50
  123 4.404790   140.123.101.163  230.0.0.1    UDP      92 62539 → 6666 Len=50
  159 5.405018   140.123.101.163  230.0.0.1    UDP      92 62539 → 6666 Len=50
  241 6.405265   140.123.101.163  230.0.0.1    UDP      92 62539 → 6666 Len=50
  920 7.406024   140.123.101.163  230.0.0.1    UDP      92 62539 → 6666 Len=50
 4561 8.406903   140.123.101.163  230.0.0.1    UDP      92 62539 → 6666 Len=50
 4983 9.407776   140.123.101.163  230.0.0.1    UDP      92 62539 → 6666 Len=50
 5431 10.411853  140.123.101.163  230.0.0.1    UDP      92 62539 → 6666 Len=50
 5636 11.408605  140.123.101.163  230.0.0.1    UDP      92 62539 → 6666 Len=50
 6253 12.408960  140.123.101.163  230.0.0.1    UDP      92 62539 → 6666 Len=50
 6455 13.408954  140.123.101.163  230.0.0.1    UDP      92 62539 → 6666 Len=50
 6555 14.409572  140.123.101.163  230.0.0.1    UDP      92 62539 → 6666 Len=50
 6610 15.409866  140.123.101.163  230.0.0.1    UDP      92 62539 → 6666 Len=50
 6682 16.410608  140.123.101.163  230.0.0.1    UDP      92 62539 → 6666 Len=50
```

# TCP stream follow in Wireshark

- Read all packets of http requests



tcp.stream eq 0

```
No.  Time
   7 2017-08-20 15:32:26.5
   9 2017-08-20 15:32:26.6
  10 2017-08-20 15:32:26.6
  11 2017-08-20 15:32:26.6
  14 2017-08-20 15:32:26.8
  15 2017-08-20 15:32:26.8
  16 2017-08-20 15:32:26.8
  17 2017-08-20 15:32:26.8
  18 2017-08-20 15:32:26.8
  19 2017-08-20 15:32:26.8
  20 2017-08-20 15:32:26.8
  21 2017-08-20 15:32:26.9
```

```
Win=65535 Len=0 MSS=1460 WS…
eq=0 Ack=1 Win=28960 Len=0 …
Ack=1 Win=132480 Len=0 TSva…

Ack=392 Win=30080 Len=0 TSv…
Ack=392 Win=30080 Len=1440 …
41 Ack=392 Win=30080 Len=14…
2 Ack=2881 Win=129600 Len=0…
html)
2 Ack=3310 Win=130624 Len=0…
eq=392 Ack=3310 Win=131072 …
10 Ack=393 Win=30080 Len=0 …
```

```
▶ Frame 7: 78 bytes on wire (6
▶ Ethernet II, Src: Apple_e7:4
▶ Internet Protocol Version 4,
▼ Transmission Control Protoco
    Source Port: 53261
    Destination Port: 80
    [Stream index: 0]
    [TCP Segment Len: 0]
    Sequence number: 0    (rel
    Acknowledgment number: 0
    1011 .... = Header Length:
  ▶ Flags: 0x002 (SYN)
    Window size value: 65535
```

```
0000  c4 ea 1d 4d 02 b8 a4 5e
0010  00 40 00 5e 40 00 40 06
0020  67 c4 d0 0d 00 50 15 86
0030  ff ff 0a a2 00 00 02 04
0040  08 0a 11 ca b5 8c 00 00
```

```
GET / HTTP/1.1
Host: didierstevens.com
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/
60.0.3112.90 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.8

HTTP/1.1 200 OK
Date: Sun, 20 Aug 2017 13:32:28 GMT
Server: Apache
Last-Modified: Wed, 06 Jul 2016 07:15:03 GMT
ETag: "460a0-bee-536f25402abc0"
Accept-Ranges: bytes
Content-Length: 3054
Connection: close
Content-Type: text/html; charset=UTF-8

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<link rel="icon" type="image/png" href="favicon.png"/>
<html>
        <head>
                <META HTTP-EQUIV="Refresh" CONTENT="30; URL=http://blog.DidierStevens.com">
                <title>Didier Stevens</title>
                <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
        </head>
        <body>You will be redirected to my blog <a href="http://blog.DidierStevens.com">http://
blog.DidierStevens.com</a> in 30 seconds.
                <ul>
                        <li>My company <a href=http://DidierStevensLabs.com>http://DidierStevensLabs.com</
a></li>
                        <li>My blog <a href=http://blog.DidierStevens.com>http://blog.DidierStevens.com</
a></li>
                        <li>My software <a href=http://blog.didierstevens.com/my-software>http://
```

*Packet 14. 3 client pkts, 5 server pkts, 3 turns. Click to select.*

Entire conversation (3699 bytes)

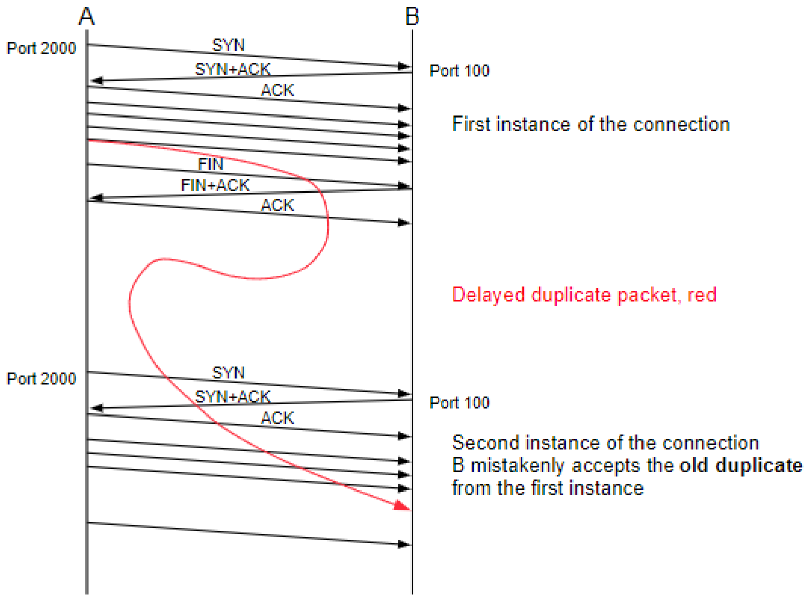
Show and save data as: ASCII

Stream: 0

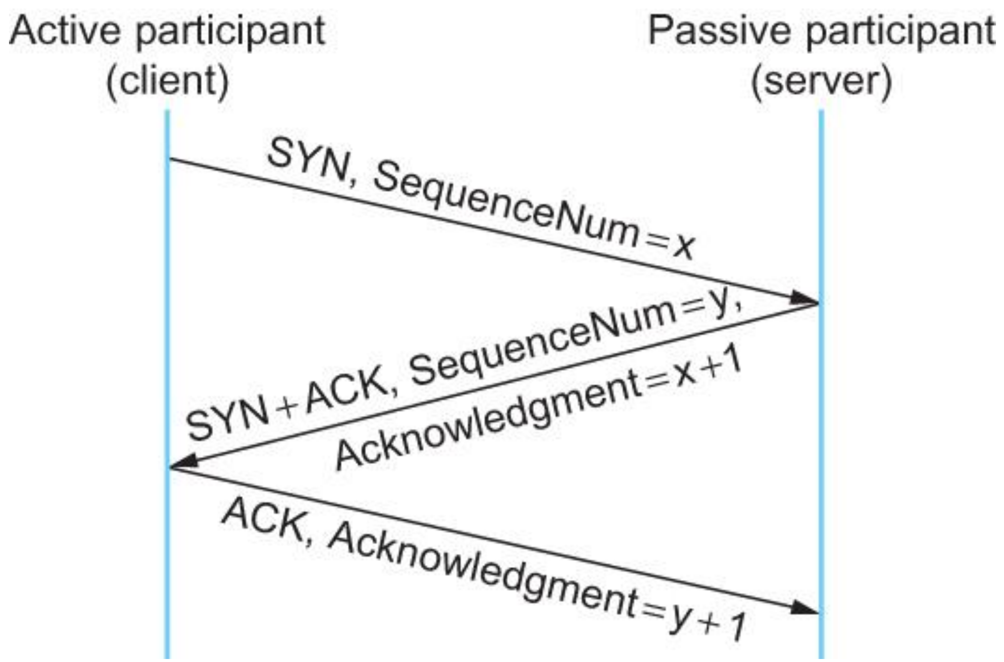
# Network issues

# TCP Issues

- At the heart of TCP is the sliding window algorithm
- As TCP runs over the Internet rather than a point-to-point link, the following issues need to be addressed by the sliding window algorithm
  - TCP supports logical connections between processes that are running on two different computers in the Internet
  - TCP connections are likely to have widely different Round Trip (RTT) times
  - Packets may get reordered in the Internet


A
B
Port 2000
SYN
SYN+ACK
ACK
Port 100
First instance of the connection
FIN
FIN+ACK
ACK
Delayed duplicate packet, red
Port 2000
SYN
SYN+ACK
ACK
Port 100
Second instance of the connection
B mistakenly accepts the old duplicate
from the first instance

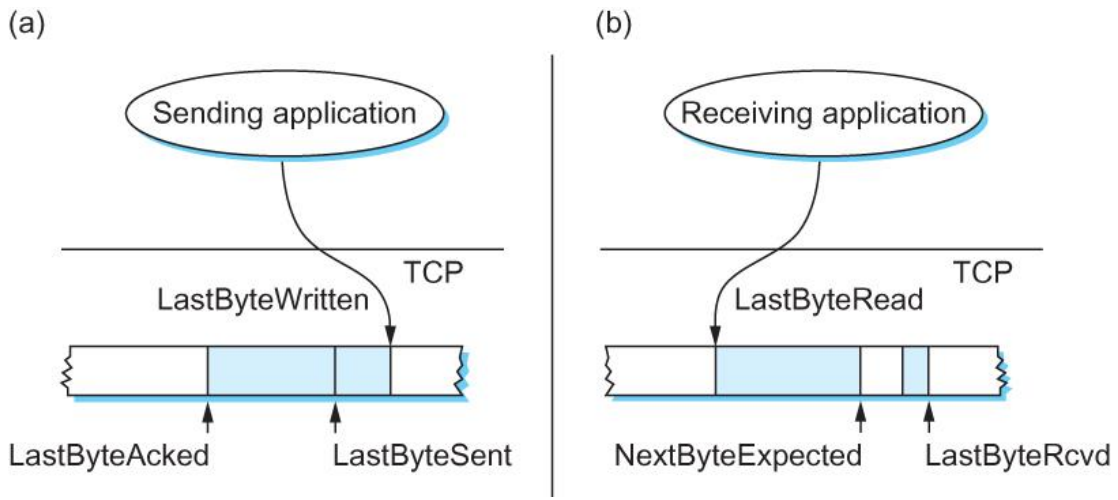
# Connection Establishment/Termination in TCP



Timeline for three-way handshake algorithm

# Sliding Window Revisited

- TCP’s variant of the sliding window algorithm, which serves several purposes:
  - (1) it guarantees the reliable delivery of data,
  - (2) it ensures that data is delivered in order, and
  - (3) it enforces flow control between the sender and the receiver.



- Sending Side
  LastByteAcked ≤ LastByteSent
  LastByteSent ≤ LastByteWritten
- Receiving Side
  LastByteRead < NextByteExpected
  NextByteExpected ≤ LastByteRcvd + 1

Relationship between TCP send buffer (a) and receive buffer (b).

# End-to-end Issues

- TCP is a byte-oriented protocol, which means that the sender writes bytes into a TCP connection and the receiver reads bytes out of the TCP connection.

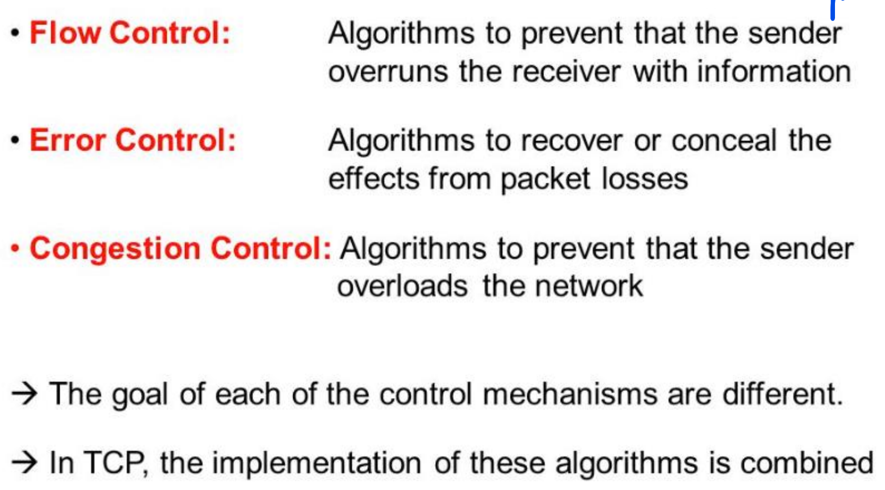
- Although "byte stream" describes the service TCP offers to application processes, TCP does not, itself, transmit individual bytes over the Internet.

- TCP needs a mechanism using which each side of a connection will learn what resources the other side is able to apply to the connection → TCP flow/congestion control

- TCP needs a mechanism using which the sending side will learn the capacity of the network

# Flow control VS Congestion control

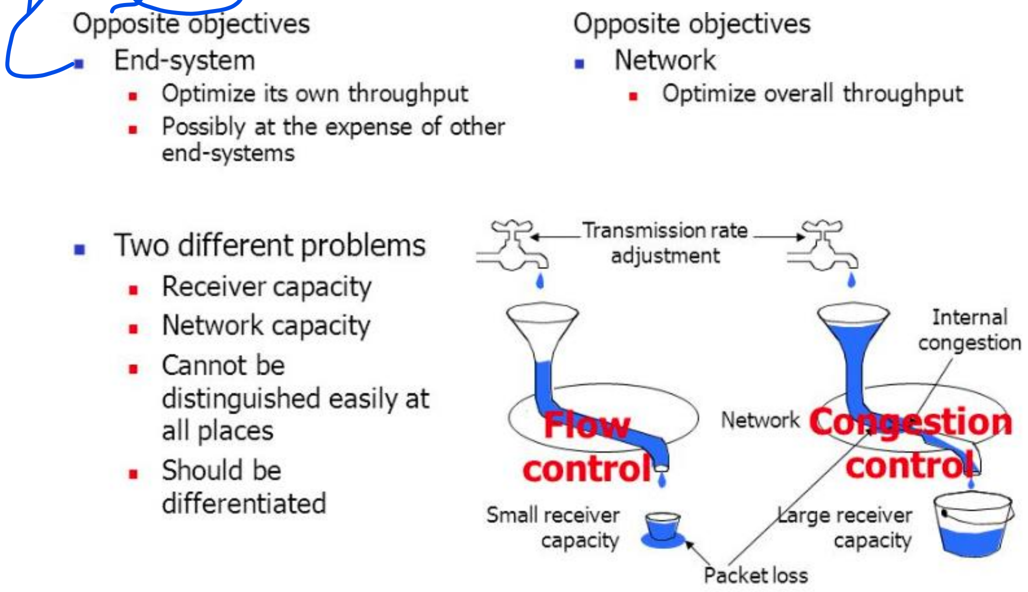
- **Flow control** involves preventing senders from overrunning the capacity of the receivers
- **Congestion control** involves preventing too much data from being injected into the network, thereby causing switches or links to become overloaded bandwidth limited



- **Flow Control:** Algorithms to prevent that the sender overruns the receiver with information
- **Error Control:** Algorithms to recover or conceal the effects from packet losses
- **Congestion Control:** Algorithms to prevent that the sender overloads the network

→ The goal of each of the control mechanisms are different.

→ In TCP, the implementation of these algorithms is combined



Opposite objectives
- End-system
  - Optimize its own throughput
  - Possibly at the expense of other end-systems

Opposite objectives
- Network
  - Optimize overall throughput

- Two different problems
  - Receiver capacity
  - Network capacity
  - Cannot be distinguished easily at all places
  - Should be differentiated

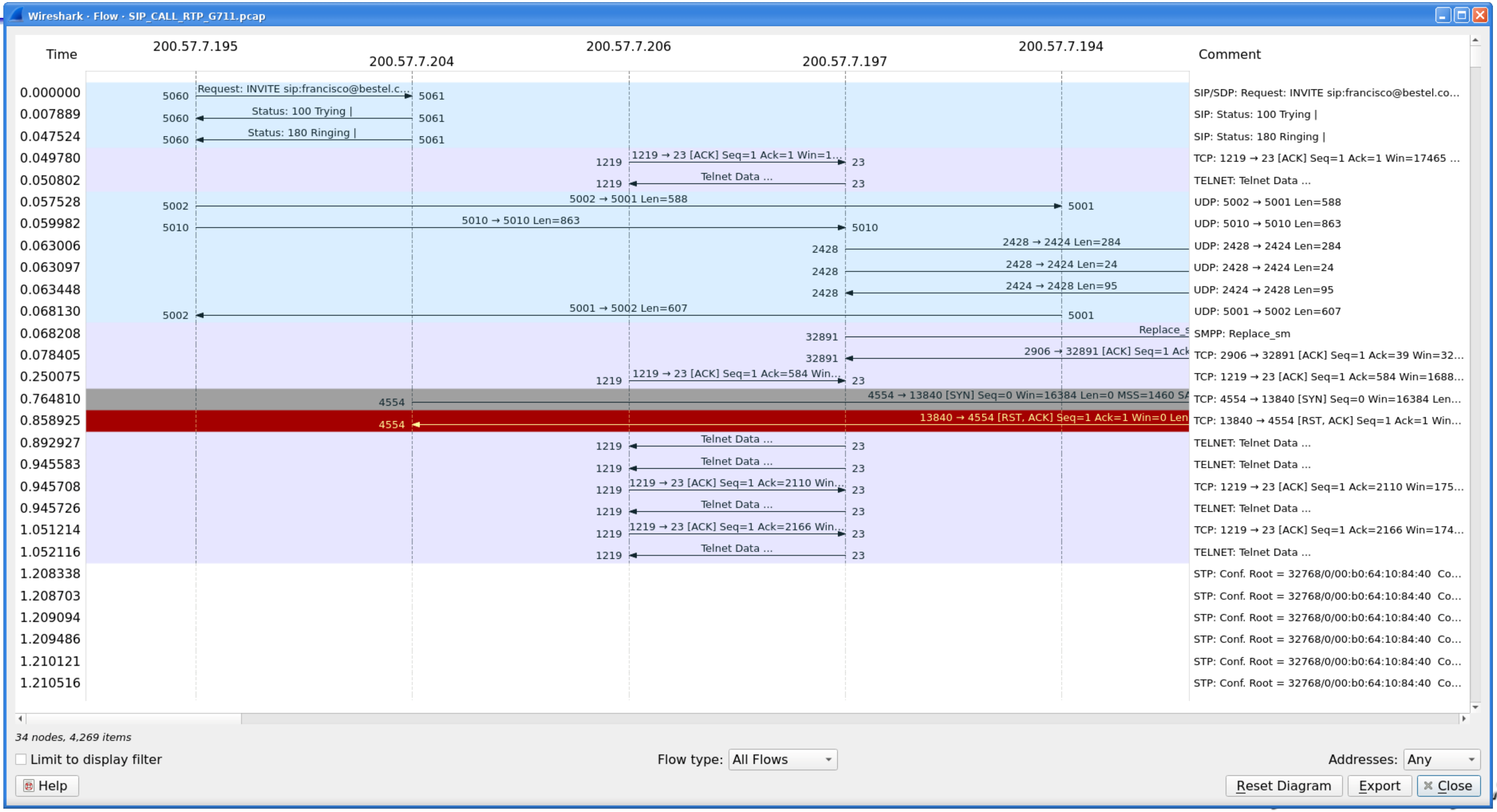
# TCP Flow Control

- $\text{LastByteRcvd} - \text{LastByteRead} \le \text{MaxRcvBuffer}$
- $\text{AdvertisedWindow} = \text{MaxRcvBuffer} - ((\text{NextByteExpected} - 1) - \text{LastByteRead})$
- $\text{LastByteSent} - \text{LastByteAcked} \le \text{AdvertisedWindow}$
- $\text{EffectiveWindow} = \text{AdvertisedWindow} - (\text{LastByteSent} - \text{LastByteAcked})$
- $\text{LastByteWritten} - \text{LastByteAcked} \le \text{MaxSendBuffer}$
- If the sending process tries to write y bytes to TCP, but

  $(\text{LastByteWritten} - \text{LastByteAcked}) + y > \text{MaxSendBuffer}$

  then TCP blocks the sending process and does not allow it to generate more data.

# TCP flow control in Wireshark


Wireshark · Flow · SIP_CALL_RTP_G711.pcap
200.57.7.195
200.57.7.204
200.57.7.206
200.57.7.197
200.57.7.194
Request: INVITE sip:francisco@bestel.c...
Status: 100 Trying |
Status: 180 Ringing |
1219 → 23 [ACK] Seq=1 Ack=1 Win=1...
Telnet Data ...
5002 → 5001 Len=588
5010 → 5010 Len=863
2428 → 2424 Len=284
2428 → 2424 Len=24
2424 → 2428 Len=95
5001 → 5002 Len=607
Replace_s
2906 → 32891 [ACK] Seq=1 Ack
1219 → 23 [ACK] Seq=1 Ack=584 Win...
4554 → 13840 [SYN] Seq=0 Win=16384 Len=0 MSS=1460 SA
13840 → 4554 [RST, ACK] Seq=1 Ack=1 Win=0 Len
1219 → 23 [ACK] Seq=1 Ack=2110 Win...
1219 → 23 [ACK] Seq=1 Ack=2166 Win...
Help
Reset Diagram
Export
Close

| Time | Comment |
|---|---|
| 0.000000 | SIP/SDP: Request: INVITE sip:francisco@bestel.co... |
| 0.007889 | SIP: Status: 100 Trying \| |
| 0.047524 | SIP: Status: 180 Ringing \| |
| 0.049780 | TCP: 1219 → 23 [ACK] Seq=1 Ack=1 Win=17465 ... |
| 0.050802 | TELNET: Telnet Data ... |
| 0.057528 | UDP: 5002 → 5001 Len=588 |
| 0.059982 | UDP: 5010 → 5010 Len=863 |
| 0.063006 | UDP: 2428 → 2424 Len=284 |
| 0.063097 | UDP: 2428 → 2424 Len=24 |
| 0.063448 | UDP: 2424 → 2428 Len=95 |
| 0.068130 | UDP: 5001 → 5002 Len=607 |
| 0.068208 | SMPP: Replace_sm |
| 0.078405 | TCP: 2906 → 32891 [ACK] Seq=1 Ack=39 Win=32... |
| 0.250075 | TCP: 1219 → 23 [ACK] Seq=1 Ack=584 Win=1688... |
| 0.764810 | TCP: 4554 → 13840 [SYN] Seq=0 Win=16384 Len... |
| 0.858925 | TCP: 13840 → 4554 [RST, ACK] Seq=1 Ack=1 Win... |
| 0.892927 | TELNET: Telnet Data ... |
| 0.945583 | TELNET: Telnet Data ... |
| 0.945708 | TCP: 1219 → 23 [ACK] Seq=1 Ack=2110 Win=175... |
| 0.945726 | TELNET: Telnet Data ... |
| 1.051214 | TCP: 1219 → 23 [ACK] Seq=1 Ack=2166 Win=174... |
| 1.052116 | TELNET: Telnet Data ... |
| 1.208338 | STP: Conf. Root = 32768/0/00:b0:64:10:84:40 Co... |
| 1.208703 | STP: Conf. Root = 32768/0/00:b0:64:10:84:40 Co... |
| 1.209094 | STP: Conf. Root = 32768/0/00:b0:64:10:84:40 Co... |
| 1.209486 | STP: Conf. Root = 32768/0/00:b0:64:10:84:40 Co... |
| 1.210121 | STP: Conf. Root = 32768/0/00:b0:64:10:84:40 Co... |
| 1.210516 | STP: Conf. Root = 32768/0/00:b0:64:10:84:40 Co... |

*34 nodes, 4,269 items*

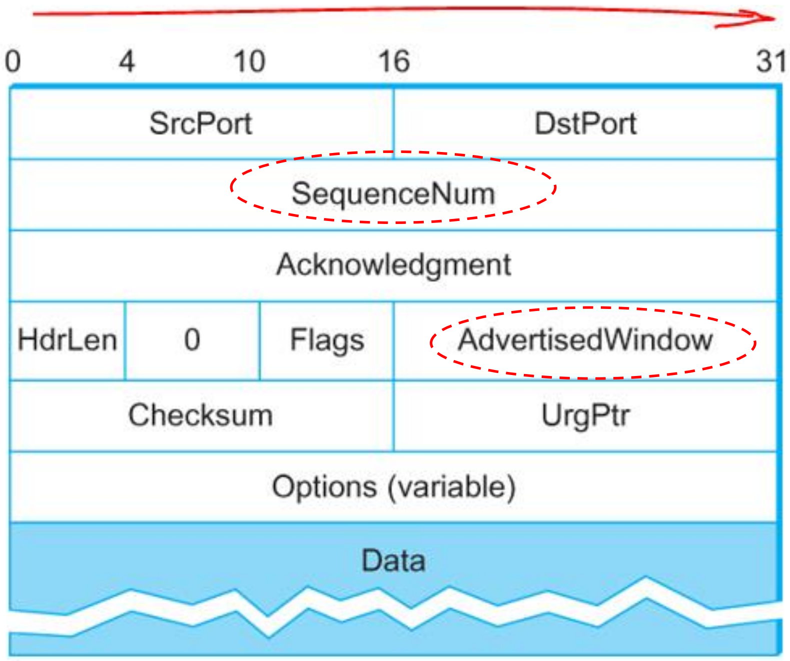
☐ Limit to display filter

Flow type: All Flows

Addresses: Any

# Consequence number’s limit: Wraparound

- SequenceNum: 32 bits longs $2^{32}$ -1 = 4GB
- Every packet's sequence number must be unique → the Sequence Number will be exhausted ( > 4GB).
- When accessible, the sequence numbers that were previously utilized can be reused as needed
- Reusing of sequence numbers is known as the Wraparound
- AdvertisedWindow: 16 bits long
  - TCP has satisfied the requirement of the sliding
  - window algorithm that is the sequence number
  - space be twice as big as the window size
  - $2^{32} >> 2 \times 2^{16}$



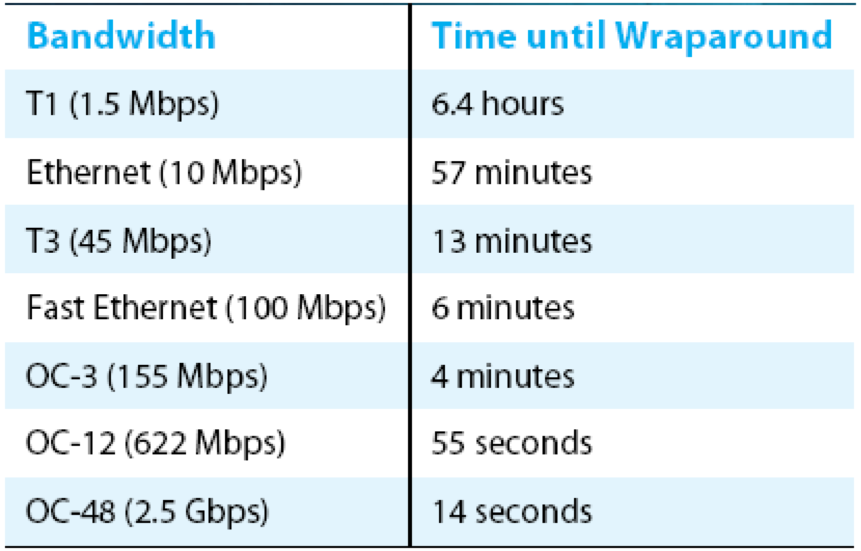
0 4 10 16 31

| SrcPort | DstPort |
| SequenceNum |
| Acknowledgment |
| HdrLen | 0 | Flags | AdvertisedWindow |
| Checksum | UrgPtr |
| Options (variable) |
| Data |

TCP Header Format

# Protecting against Wraparound

- Relevance of the 32-bit sequence number space
  - The sequence number used on a given connection might wraparound
  - A byte with sequence number *x* could be sent at one time, and then at a  later time a second byte with the same sequence number *x* could be sent
  - Packets cannot survive in the Internet for longer than the **MSS**
  - **MSS** is set to 120 sec
  - We need to make sure that the sequence number does not wrap around within a 120-second period of time
  - Depends on how fast data can be transmitted over the Internet

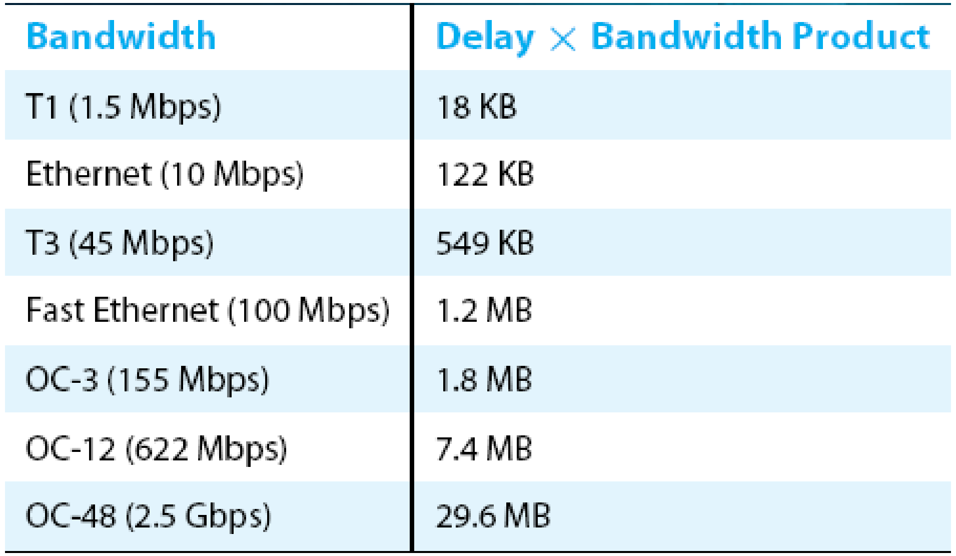


| Bandwidth | Time until Wraparound |
|---|---|
| T1 (1.5 Mbps) | 6.4 hours |
| Ethernet (10 Mbps) | 57 minutes |
| T3 (45 Mbps) | 13 minutes |
| Fast Ethernet (100 Mbps) | 6 minutes |
| OC-3 (155 Mbps) | 4 minutes |
| OC-12 (622 Mbps) | 55 seconds |
| OC-48 (2.5 Gbps) | 14 seconds |

Time until 32-bit sequence number space wraps around.

# Keeping the Pipe Full

- 16-bit AdvertisedWindow field must be big enough to allow the sender to keep the pipe full
- Clearly the receiver is free not to open the window as large as the AdvertisedWindow field allows
- If the receiver has enough buffer space
  - The window needs to be opened far enough to allow a full
  - Delay × bandwidth product’s worth of data
  - Assuming an RTT of 100 ms



| Bandwidth | Delay × Bandwidth Product |
|---|---|
| T1 (1.5 Mbps) | 18 KB |
| Ethernet (10 Mbps) | 122 KB |
| T3 (45 Mbps) | 549 KB |
| Fast Ethernet (100 Mbps) | 1.2 MB |
| OC-3 (155 Mbps) | 1.8 MB |
| OC-12 (622 Mbps) | 7.4 MB |
| OC-48 (2.5 Gbps) | 29.6 MB |

Required window size for 100-ms RTT.

# Triggering Transmission

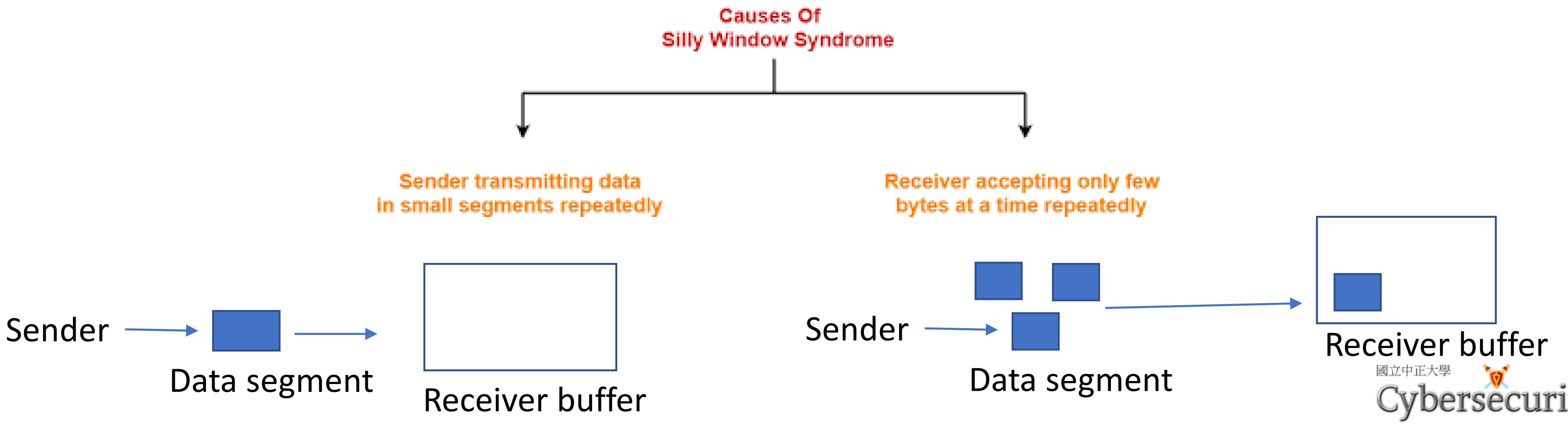
- How does TCP decide to transmit a segment?
  - TCP supports a byte stream abstraction
  - Application programs write bytes into streams
  - It is up to TCP to decide that it has enough bytes to send a segment

- What factors governs this decision
  - Ignore flow control: window is wide open, as would be the case when the connection starts
  - TCP has three mechanism to trigger the transmission of a segment
    - 1) TCP maintains a variable MSS and sends a segment as soon as it has collected MSS bytes from the sending process
      - MSS is usually set to the size of the largest segment TCP can send without causing local IP to fragment.
      - MSS: MTU of directly connected network – (TCP header + and IP header)
    - 2) Sending process has explicitly asked TCP to send it
      - TCP supports push operation
    - 3) When a timer fires
      - Resulting segment contains as many bytes as are currently buffered for transmission

# Silly Window Syndrome

- A problem in computer networking caused by poorly implemented TCP flow control

- This problem occurs when the sending application program creates data slowly, the receiving application program consumes data slowly, or both



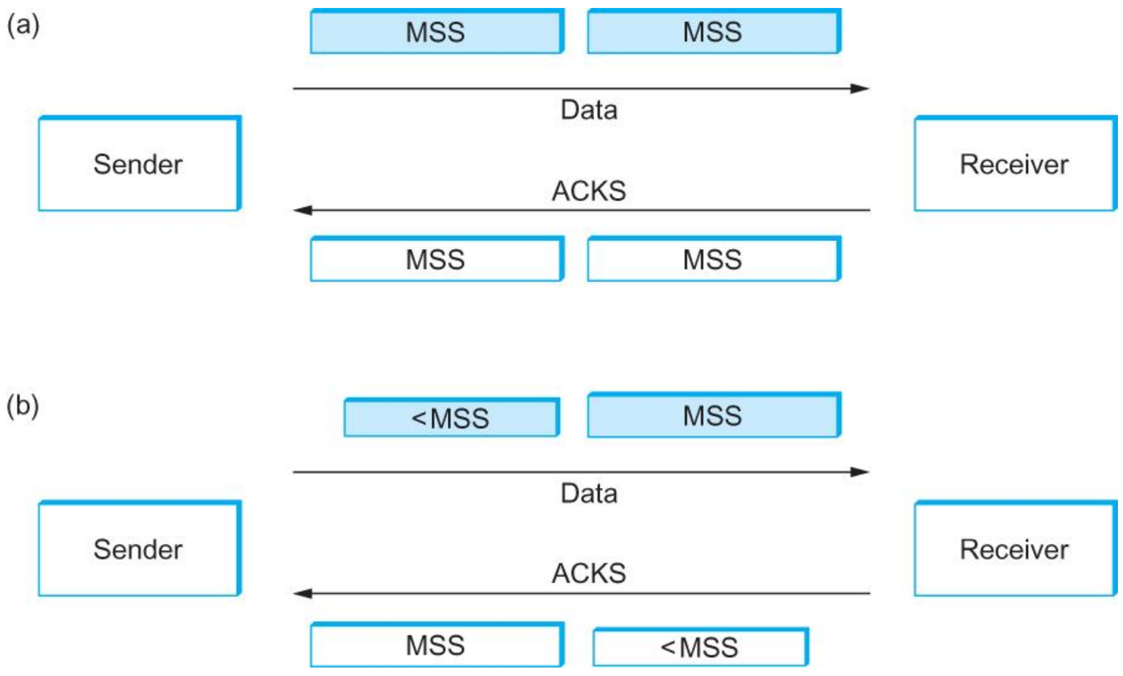
Causes Of
Silly Window Syndrome

Sender transmitting data
in small segments repeatedly

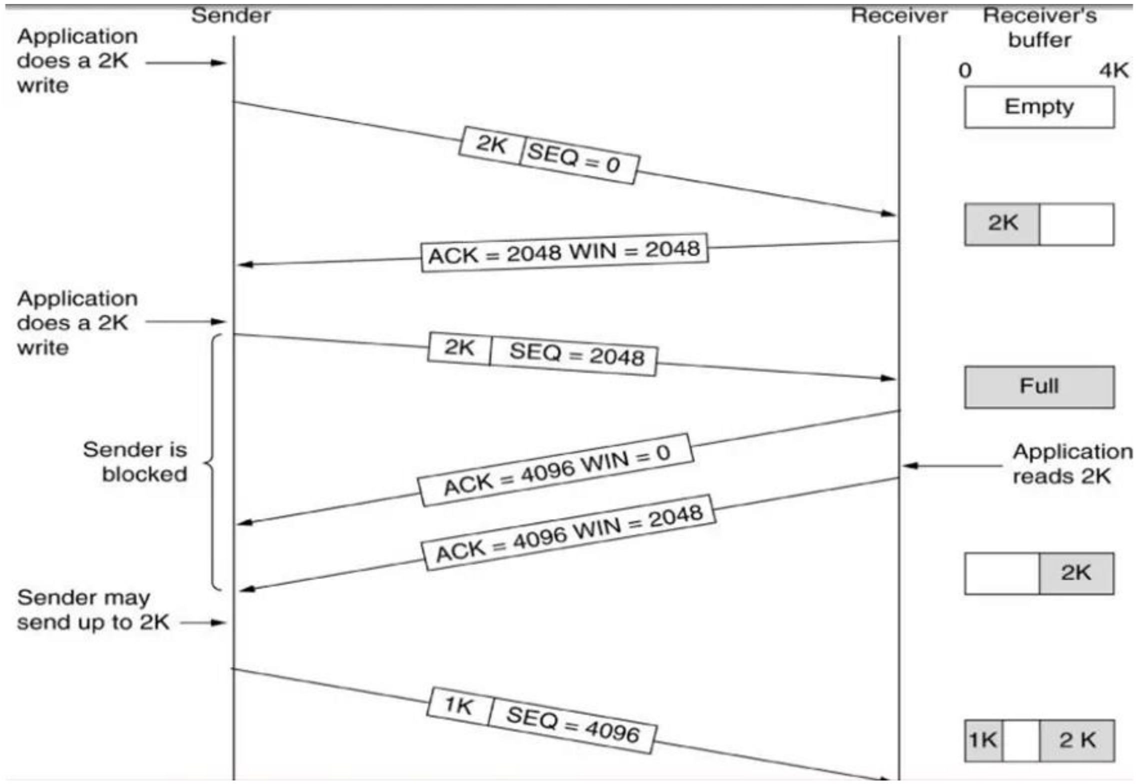
Receiver accepting only few
bytes at a time repeatedly

國立中正大學
Cybersecurity Lab

# Silly Window Syndrome



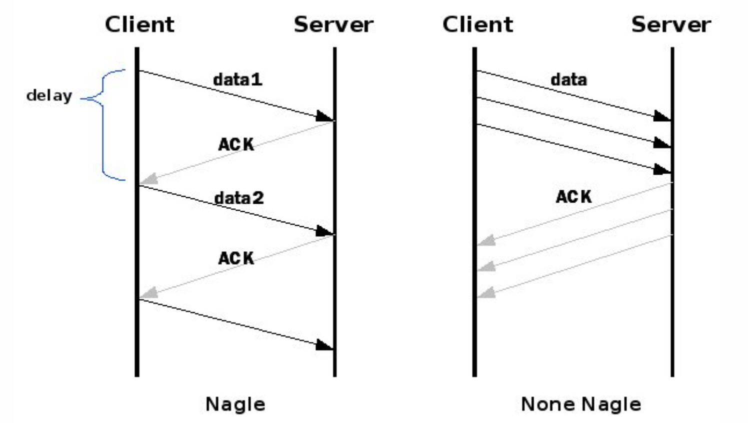
Silly Window Syndrome



Source: Gateoverflow

# Nagle’s Algorithm

- If there is data to send but the window is open less than MSS, then we may want to wait some amount of time before sending the available data
- But how long?
- If we wait too long, then we hurt interactive applications like Telnet
- If we don’t wait long enough, then we risk sending a bunch of tiny packets and falling into the *silly window* syndrome
  - The solution is to introduce a timer and to transmit when the timer expires



# Nagle's Algorithm

- We could use a clock-based timer, for example one that fires every 100 ms
- Nagle introduced an elegant self-clocking solution
- Key Idea
  - As long as TCP has any data in flight, the sender will eventually receive an ACK
  - This ACK can be treated like a timer firing, triggering the transmission of more data

# Nagle’s Algorithm

```
When the application produces data to send
  if both the available data and the window $\geq$ MSS
        send a full segment
  else
        if there is unACKed data in flight
                buffer the new data until an ACK arrives
        else
                send all the new data now
```

# Adaptive Retransmission

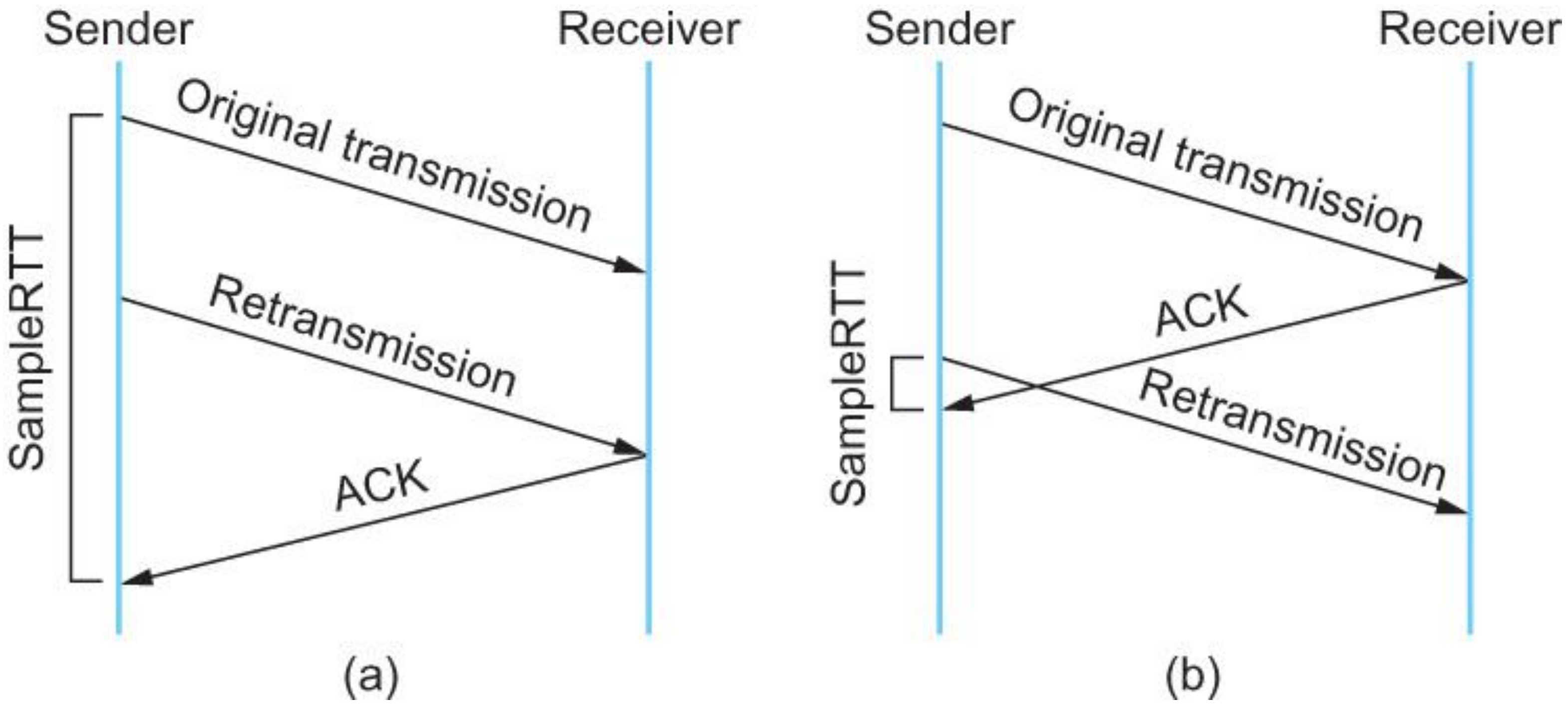
- Original Algorithm
  - Measure **`SampleRTT`** for each segment/ ACK pair
  - Compute weighted average of RTT
    - **`EstRTT`** = $\alpha$ x **`EstRTT`** + (1 - $\alpha$ )x **`SampleRTT`**

    – $\alpha$ between 0.8 and 0.9
  - Set timeout based on **`EstRTT`**
    - **`TimeOut`** = **2** x **`EstRTT`**

# Original Algorithm

- Problem
    - ACK does not really acknowledge a transmission
        - It actually acknowledges the receipt of data
    - When a segment is retransmitted and then an ACK arrives at the sender
        - It is impossible to decide if this ACK should be associated with the first or the second transmission for calculating RTTs

# Karn/Partridge Algorithm



Associating the ACK with (a) original transmission versus (b) retransmission

# Karn/Partridge Algorithm

- Do not sample RTT when retransmitting
- Double timeout after each retransmission

- Karn-Partridge algorithm was an improvement over the original approach, but it does not eliminate congestion

- We need to understand how timeout is related to congestion
  - If you timeout too soon, you may unnecessarily retransmit a segment which adds load to the network

# Karn/Partridge Algorithm

- Main problem with the original computation is that it does not take variance of Sample RTTs into consideration.
- If the variance among Sample RTTs is small
  - Then the Estimated RTT can be better trusted
  - There is no need to multiply this by 2 to compute the timeout
- On the other hand, a large variance in the samples suggest that timeout value should not be tightly coupled to the Estimated RTT
- Jacobson/Karels proposed a new scheme for TCP retransmission

# Jacobson/Karels Algorithm

- Difference = SampleRTT − EstimatedRTT
- EstimatedRTT = EstimatedRTT + ( × Difference)
- Deviation = Deviation + (|Difference| − Deviation)
- TimeOut = μ × EstimatedRTT +  × Deviation
    - where based on experience, μ is typically set to 1 and  is set to 4. Thus, when the variance is small, TimeOut is close to EstimatedRTT; a large variance causes the deviation term to dominate the calculation.

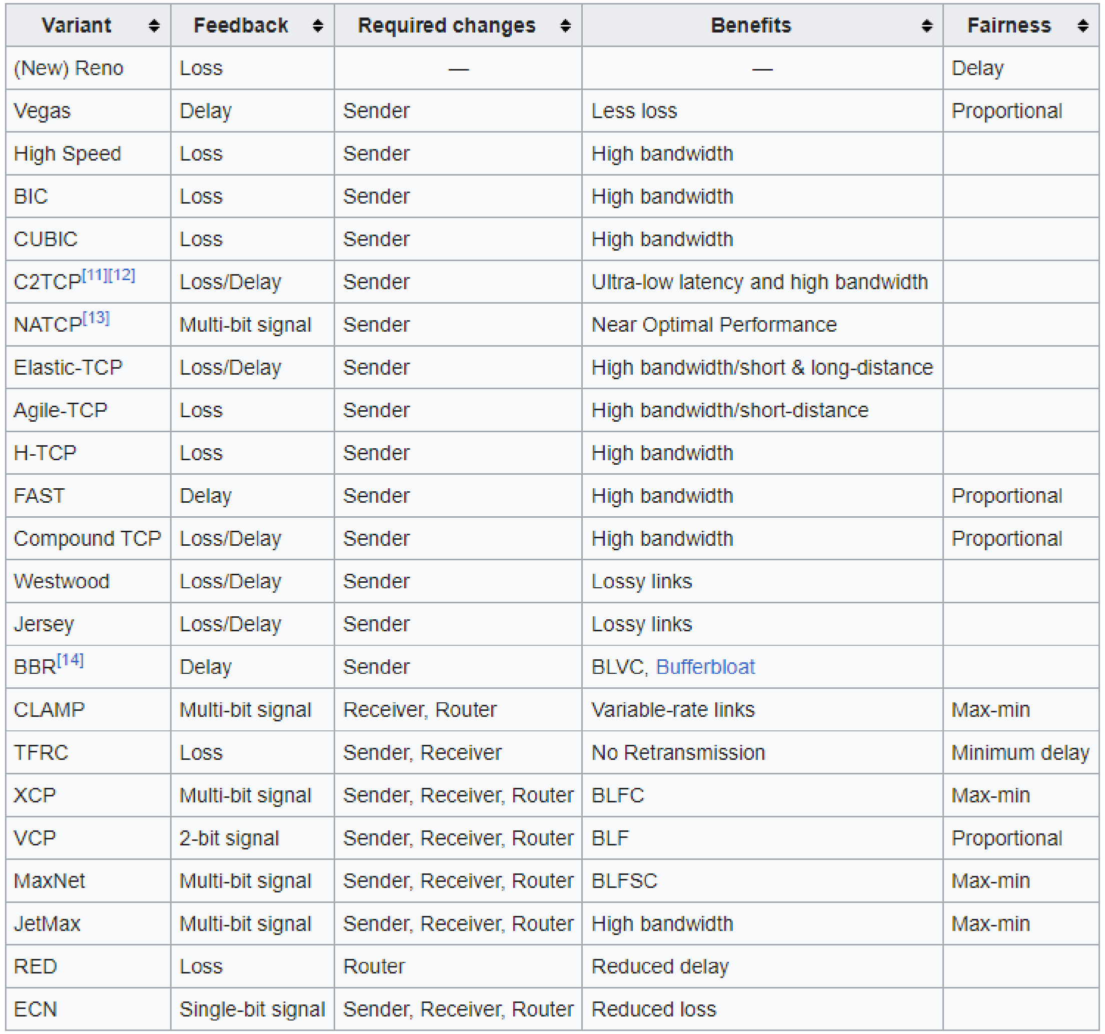
# TCP congestion control

- The mechanism that prevents congestion from happening or removes it after congestion takes place

- Congestion window state of TCP that limits the amount of data to be sent by the sender into the network even before receiving the acknowledgment

# TCP congestion control list

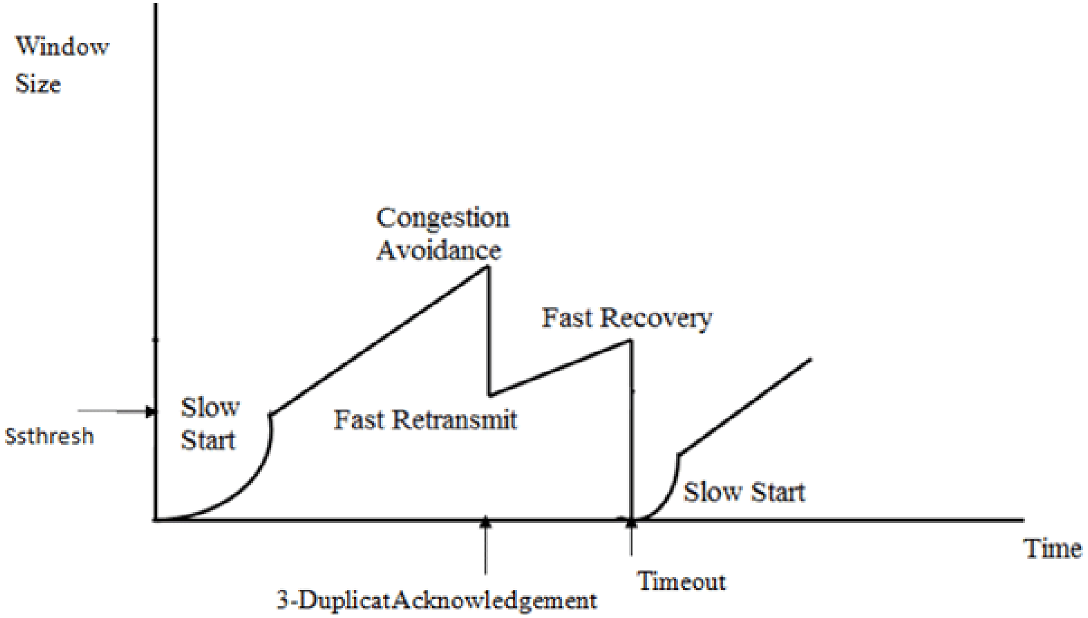
- The naming convention for congestion control algorithms (CCAs) may have originated in a 1996 paper by Kevin Fall and Sally Floyd
- There are many variants of congestion algorithms: TCP Reno/TCP Vegas….
- Efficient TCP congestion control is still a open problem for research

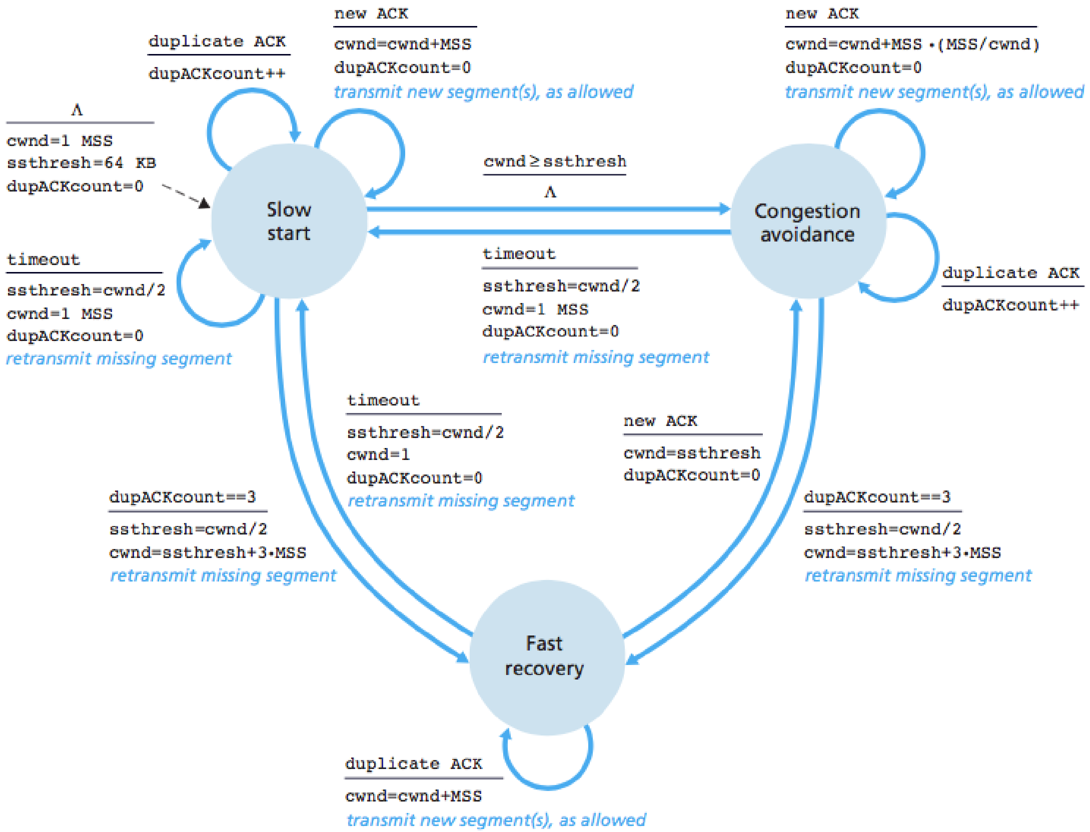


| Variant | Feedback | Required changes | Benefits | Fairness |
|---|---|---|---|---|
| (New) Reno | Loss | — | — | Delay |
| Vegas | Delay | Sender | Less loss | Proportional |
| High Speed | Loss | Sender | High bandwidth | |
| BIC | Loss | Sender | High bandwidth | |
| CUBIC | Loss | Sender | High bandwidth | |
| C2TCP[11][12] | Loss/Delay | Sender | Ultra-low latency and high bandwidth | |
| NATCP[13] | Multi-bit signal | Sender | Near Optimal Performance | |
| Elastic-TCP | Loss/Delay | Sender | High bandwidth/short & long-distance | |
| Agile-TCP | Loss | Sender | High bandwidth/short-distance | |
| H-TCP | Loss | Sender | High bandwidth | |
| FAST | Delay | Sender | High bandwidth | Proportional |
| Compound TCP | Loss/Delay | Sender | High bandwidth | Proportional |
| Westwood | Loss/Delay | Sender | Lossy links | |
| Jersey | Loss/Delay | Sender | Lossy links | |
| BBR[14] | Delay | Sender | BLVC, Bufferbloat | |
| CLAMP | Multi-bit signal | Receiver, Router | Variable-rate links | Max-min |
| TFRC | Loss | Sender, Receiver | No Retransmission | Minimum delay |
| XCP | Multi-bit signal | Sender, Receiver, Router | BLFC | Max-min |
| VCP | 2-bit signal | Sender, Receiver, Router | BLF | Proportional |
| MaxNet | Multi-bit signal | Sender, Receiver, Router | BLFSC | Max-min |
| JetMax | Multi-bit signal | Sender, Receiver, Router | High bandwidth | Max-min |
| RED | Loss | Router | Reduced delay | |
| ECN | Single-bit signal | Sender, Receiver, Router | Reduced loss | |

# TCP congestion control

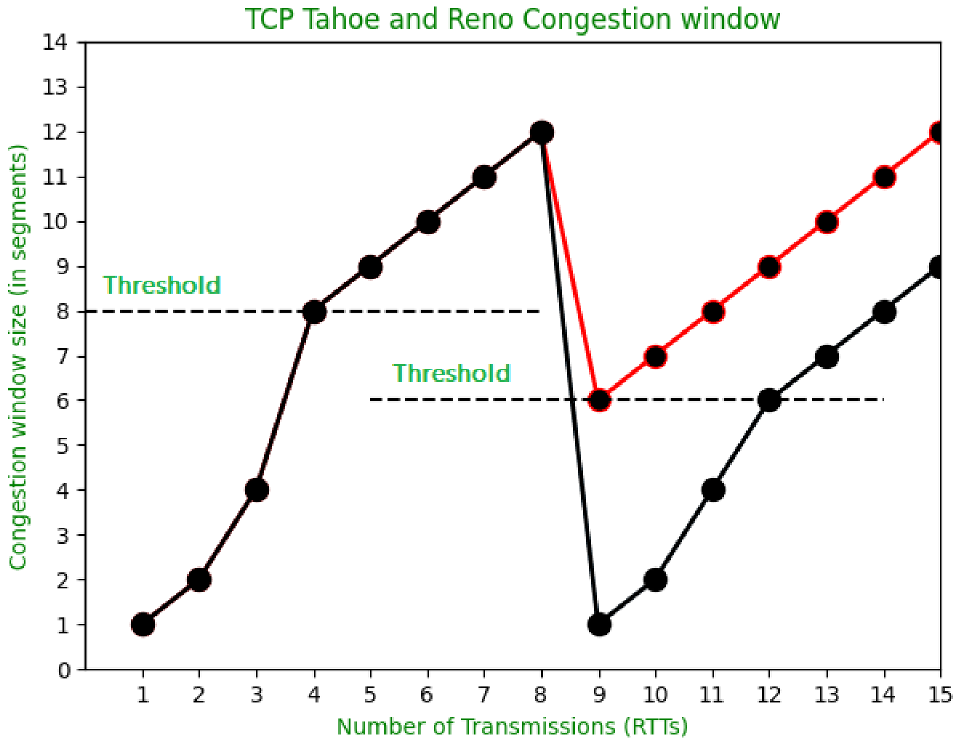
- Some famous congestion control mechanisms

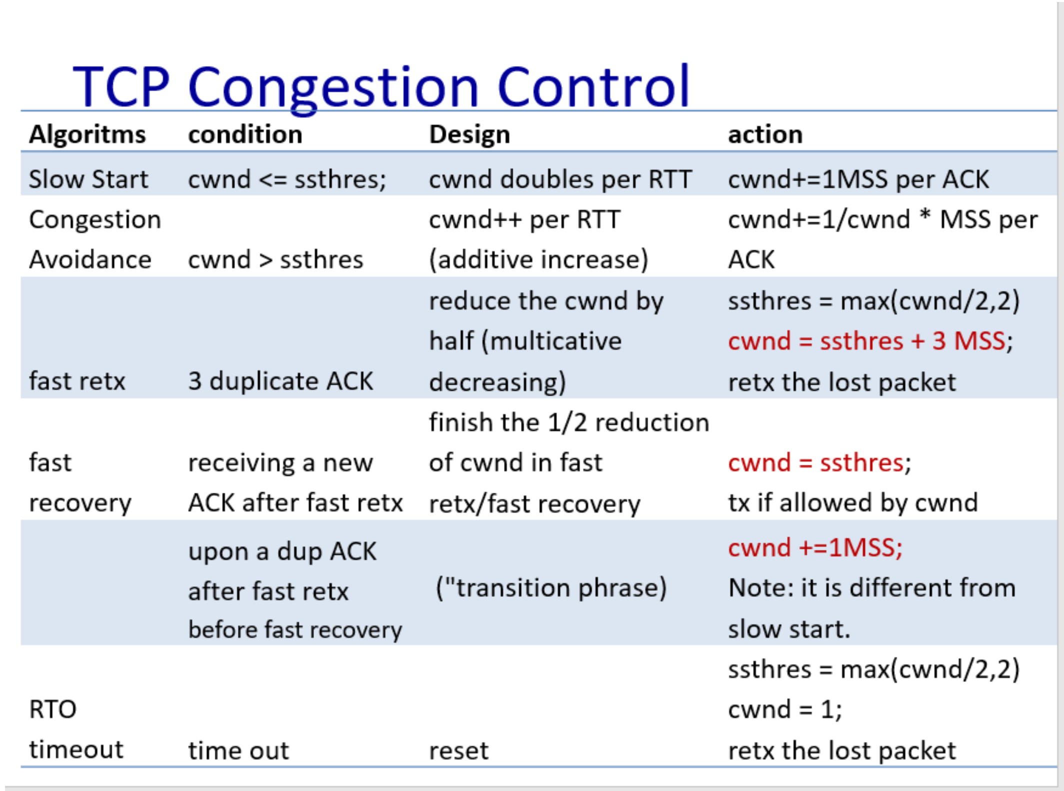




# TCP congestion control

TCP Reno may be the best choice due to its ability to adapt quickly and avoid congestion





## TCP Congestion Control

| Algoritms | condition | Design | action |
|---|---|---|---|
| Slow Start | cwnd <= ssthres; | cwnd doubles per RTT | cwnd+=1MSS per ACK |
| Congestion Avoidance | cwnd > ssthres | cwnd++ per RTT (additive increase) | cwnd+=1/cwnd * MSS per ACK |
| fast retx | 3 duplicate ACK | reduce the cwnd by half (multicative decreasing) | ssthres = max(cwnd/2,2)<br>cwnd = ssthres + 3 MSS;<br>retx the lost packet |
| fast recovery | receiving a new ACK after fast retx | finish the 1/2 reduction of cwnd in fast retx/fast recovery | cwnd = ssthres;<br>tx if allowed by cwnd |
|  | upon a dup ACK after fast retx before fast recovery | ("transition phrase) | cwnd +=1MSS;<br>Note: it is different from slow start. |
| RTO timeout | time out | reset | ssthres = max(cwnd/2,2)<br>cwnd = 1;<br>retx the lost packet |

# Summary

- We have discussed how to transmit data via transport layer.
- We have discussed UDP
- We have discussed TCP/TCP flow control/congestion control
- We have discussed to use Wireshark to analyze packets
- We have discussed several algorithms to optimize TCP transmission