The Transport Layer Protocols

COMPUTER NETWORKS A.A. 24/25

Leonardo Maccari, DAIS: Ca' Foscari University of Venice,
leonardo.maccari@unive.it

Venice, fall 2024

License

- These slides contain material whose copyright is of Olivier Bonaventure, Universite catholique de Louvain, Belgium https://inl.info.ucl.ac.be
- The slides are licensed under a Creative Commons Attribution-Share Alike 3.0 Unported License.

Introduction

- Transport protocols rely on the service provided by the network layer.
- On the Internet, the network layer provides a connectionless service. The network layer identifies each (interface of a) host by using an IP address.
- It enables hosts to transmit packets that contain up to 64 KBytes of payload to any destination reachable through the network.
- The network layer does not guarantee the delivery of information, cannot detect transmission errors and does not preserve sequence integrity.

Introduction (2)

- The two most widely deployed transport protocols on the Internet are the User Datagram Protocol (UDP) and the Transmission Control Protocol (TCP).
- The first offers a connection-less unreliable transport, the second a connection-oriented, reliable transport
- Other protocols like the Stream Control Transmission Protocol (SCTP) or the Real Time Transport Protocol (RTP) are used in niche applications and we don't cover them

Sect. 1 UDP

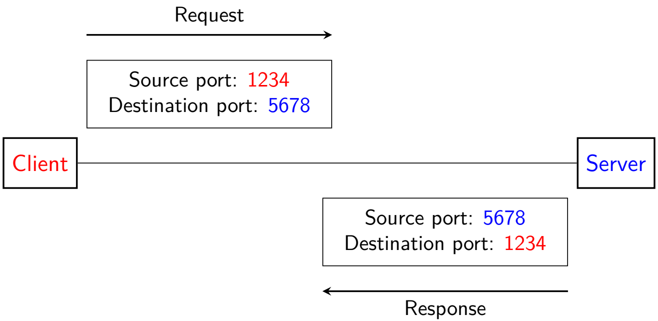
The User Datagram Protocol (UDP)

- UDP provides an unreliable connectionless transport service on top of the unreliable network layer connectionless service (RFC 768).
- The main characteristics of the UDP service are:
  - maximum SDUs size is 65467 bytes
  - does not guarantee the delivery of SDUs (losses can occur and SDUs can arrive out-of-sequence)
  - error detection

Ports and Multiplexing

- The main advantage of using the UDP service compared to using the connectionless network service is that it allows several applications running on a host to exchange SDUs with several other applications running on remote hosts.
- applications on one of the communicating hosts are identified by a port number

UDP messages and header




```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            Length             |           Checksum            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

UDP Header

he UDP header contains four fields:

- a 16 bits source port
- a 16 bits destination port
- a 16 bits length field
- a 16 bits checksum

The checksum is computed over UDP header fields, IP pseudo-header (a subset of the IP header including the addresses) and the payload.

UDP Ports

- Port numbers are encoded as a 16 bits field, so there can be up to only 65535 different server processes that are bound to a different UDP port at the same time on a given server.
- Most implementations divide the range of allowed UDP port numbers into three different ranges
  - the privileged port numbers ($1 < \text{port} < 1024$). In most Unix variants, only processes having system administrator privileges can use these ports
  - the registered port numbers (officially $1024 <= \text{port} < 49152$). Protocols that use these ports should ask IETF for an assigend number
  - the ephemeral port numbers (officially $49152 <= \text{port} <= 65535$). Anyone can use them

UDP usage

- As said several times, UDP is used for applications that do not need reliability but need to minimize latency
- Some example are:
  - Voice over IP: the RTP application-layer protocol uses UDP
  - DNS: as a simple request-response protocol it does not need a connection or state
  - QUIC: a new transport protocol focused on multimedia content, based on UDP (RFC 9000). There is a whole new version of HTTP/3 that is based on QUIC (RFC 9114)

Sect. 2 TCP

Transmission Control Protocol (TCP)

- TCP was initially defined in RFC 793, and despite many changes, an implementation that only supports RFC 793 should inter-operate with today's implementation.
- TCP provides a reliable bytestream, connection-oriented transport service on top of the unreliable connectionless network service provided by IP.
- TCP is used by a large number of applications, including web browing, e-mail etc. . . .
- The service unit of TCP is a *segment*.
- Each transmitted byte corresponds to a sequence number

Segment Header

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |       |C|E|U|A|P|R|S|F|                               |
| Offset| Res.  |W|C|R|C|S|S|Y|I|            Window             |
|       |       |R|E|G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Segment Header

The TCP header has a fixed part, and and optional extensions, so it does not necessarily have a fixed length.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |       |C|E|U|A|P|R|S|F|                               |
| Offset| Res.  |W|C|R|C|S|S|Y|I|            Window             |
|       |       |R|E|G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- 16 bit source and destination ports
- 32 bits sequence and ack numbers
- the TCP Header Length (THL) or *Data Offset* (4 bits) indicates the size of the TCP header in 32 bit words. The maximum size of the TCP header is thus 64 bytes.
- Reserved bits, now used for ECN

Segment Header

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |       |C|E|U|A|P|R|S|F|                               |
| Offset| Res.  |W|C|R|C|S|S|Y|I|            Window             |
|       |       |R|E|G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Flags (some of them):

- `SYN` is used during connection establishment
- `FIN` is used during connection release
- `RST` closes the connection in case of problems
- `ACK` indicates that the acknowledgment field is valid.
- `CWR`, `ECE` Used for Explicit Network Congestion (we do not consider them)
- `PSH`, `URG` field and Urgent Pointer, see later. . . .

Segment Header

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |       |C|E|U|A|P|R|S|F|                               |
| Offset| Res.  |W|C|R|C|S|S|Y|I|            Window             |
|       |       |R|E|G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- `window`: the size of the receiving window of the sender
- `checksum`: IP-style checksum over the whole TCP segment and some IP header fields too (IP addresses)
- After the fixed-size TCP header, optional header extensions can follow.
- We will look at 3 of them

PSH and Urgent Pointer

- The receiving part, we know, has a data buffer where it stores the received data
- It will notify the application of new data, but the sender does not control when this will happen.
- For instance, the receiver may buffer a few bytes before sending them to the application.
- There are cases in which this is detrimental to the service

Case 1: remote shell

- if you type on a remote shell, you need to see what you type.
- So when you send a single byte (a letter), this will travel to the receiver and then, it will be echoed back to you, which means the server will send it back.
- You want this to happen in real time, not in a delayed way.
- The `PSH` flag does this: when the receiver receives a segment with PSH set, it should send all the content of the buffer to the application immediately
- Note, the order of the sequence numbers is preserved, the new data does not overcome old ones. It just requires to flush the buffer at the receiver.

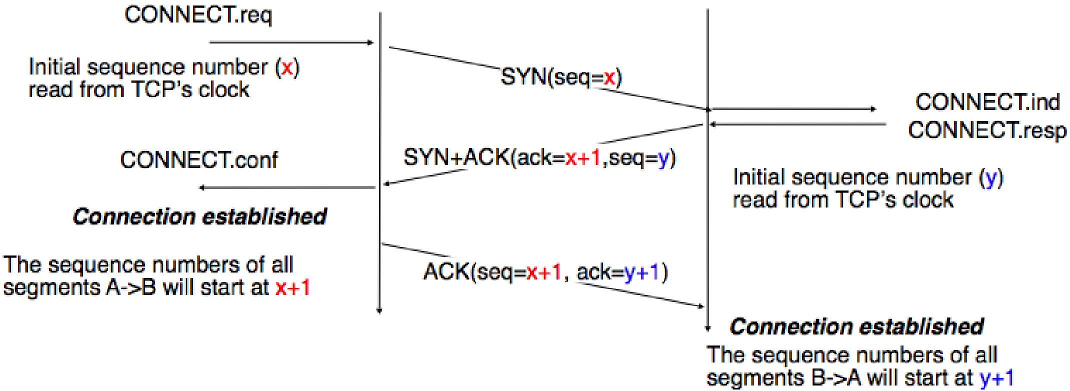
Case 2: CTRL-C

- Assume you pasted on the remote shell a long text by mistake.
- Every char will go back and forth, and you want to stop this
- You push `CTRL-C`, but this command will be queued and elaborated after all the data you already pasted will have arrived.
- The `URG` flag and `Urgent Pointer` fields served this role: the receiver should pass this data to the application, without following the order of sequence numbers.
- Actually both `URG` is deprecated and `PSH` is almost never used.

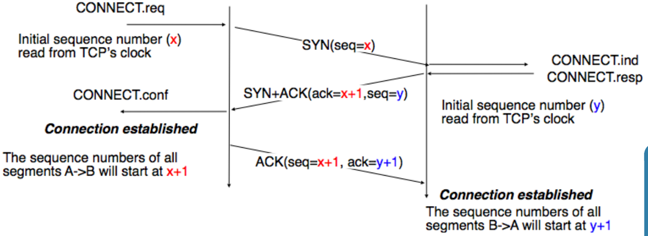
TCP

↳ 2.1 TCP FSM

Connection Set-Up



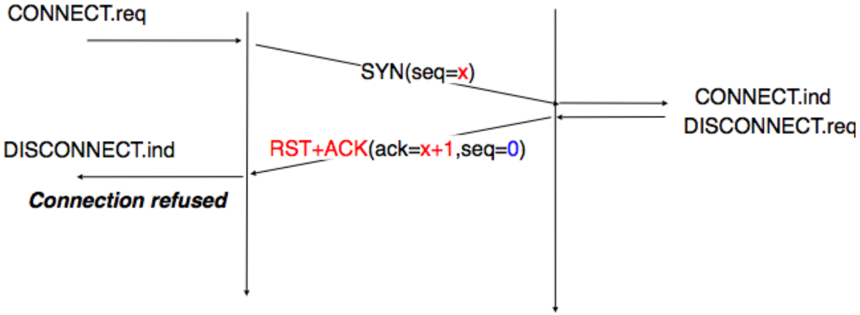
Connection Set-Up



- We have seen how a generic transport connection is established
- The only difference with TCP is the interpretation of `ACK` numbers

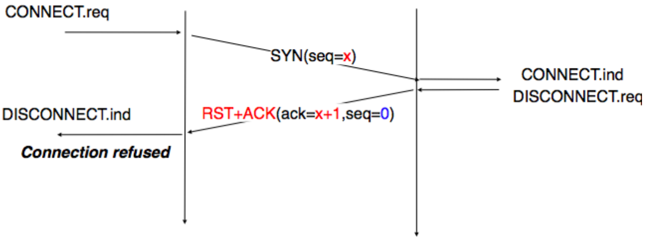
> A TCP `ACK` equal to X+1 means: I received all data up to X, I am expecting X+1

Connection Refusal



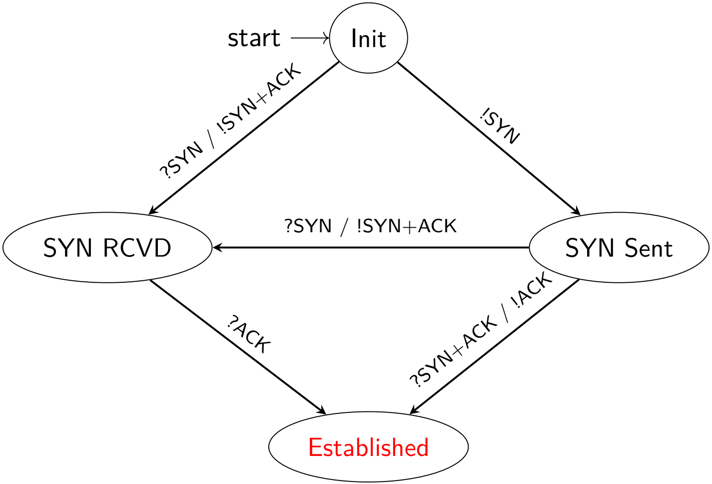
For instance, there is no application on that port.

Connection Set-Up



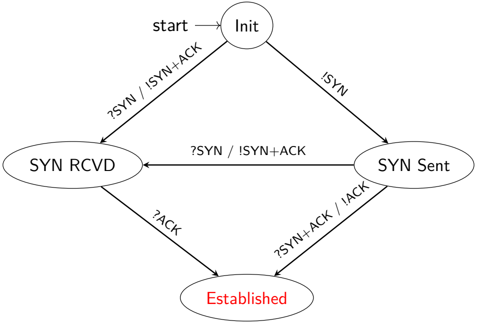
- Note that the RST flag must be sent with a valid ack number
- This is to prevent *blind RST spoofing*, that we will explain better later on.

TCP Connection Establishment FSM



Here "?" means receive and "!" means send.

TCP Connection Establishment FSM



- The only new thing here is the possibility that two hosts are both client and servers
- If they open a connection at the same time, using the same exact ports, they would stale in absence of the edge between `SYN Sent` and `SYN RCVD`. Instead, they send a `SYN+ACK`, and this of course counts as an `ACK`, and make the state move to Established.
- And of course there are retransmission timers to take into account loss of segments (not shown in the figure)

Key TCP Options 1: MSS

- The `SYN` and `SYN+ACK` packets can contain optional extensions that are used later-on
- The most important one is the `Maximum Segment Size (MSS)`, every TCP host must be able to interpret it.
- Every host sets this value to the size of the largest segment it can receive.
- `MSS` is normally set to the MTU of the network, the largest size of a data frame in the receiver network, in order to avoid fragmentation
- The minimum size is 536, but this is normally larger than that.
- `MSS` can be different, both endpoints announce their own and send it to the other.

Memory Occupation

- To implement the FSM both endpoints must maintain some state
- For each ongoing connection TCP maintains a so-called `TCB` that is `Transmission Control Block`, a set of information needed to maintain the state of the connection.
- More on its contents, later on. However, we can already say that the `TCB` contains the receiving and sending buffers of TCP

TCP Connection Release

- TCP supports two types of connection releases:
  - graceful connection release, where each TCP user can release its own direction of data transfer after having transmitted all data
  - abrupt connection release, where either one user closes both directions of data transfer or one TCP entity is forced to close the connection. For instance a non-SYN segment was received for a non-existing TCP connection, or some error in the headers.

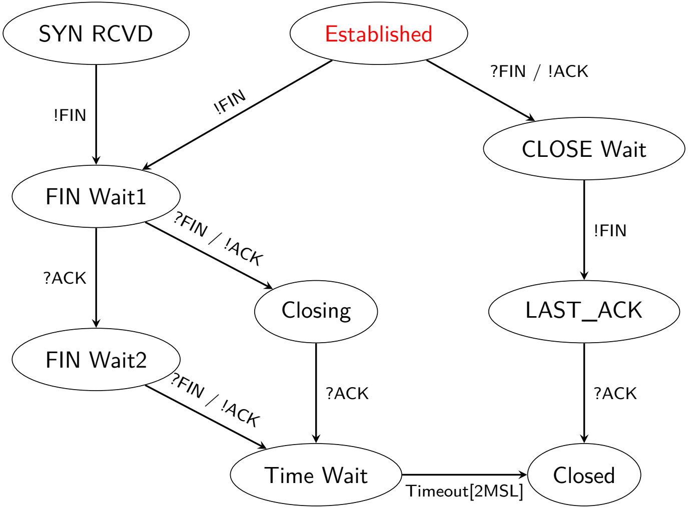
Graceful Release

- A graceful release happens with a `FIN` segment with sequence number x, that means *I am done sending you all data, last valid byte is* x
- The peer will still be able to re-send lost data before x, and waits for a `FIN` from the other side.
- `FIN` segments must be acknowledged.

Abrupt Release

- Abrupt release happens with a segment with `RST` flag, and as soon as it was sent or received, the `TCB` is erased.
- To avoid `RST` ping-pong, a TCP peer can never answer with `RST` to another `RST`

TCP Connection Release



Time Wait

- The RFC mandates a `Time Wait` state, that needs to be held for $2 \times MSL$ (4 minutes)
- This is because the `ACK` to the last `FIN` could be lost and needs to be retransmitted, so before removing the `TCB`, we must be sure that nothing needs to be retransmitted.
- However, this can lead to a large memory occupation

Exercise

- Assume 32GB memory are allocated for the TCB in a web server
- How big is a TCB? It depends, but the largest part is the size of the receive buffer. On Linux:

```
cat /proc/sys/net/ipv4/tcp_rmem
4096   131072   6291456
```

  This is the minimum/default/maximum.
- Let's assume the default value (actually the value itself is not important as we will see), then we have at most

$$32GB/131072B = 2^{35}/131072 = 262144$$

connections.

Exercise

- Let's assume a connection lasts 1 minute, that is the time to load the page, read the page, close the tab.
- However, the connection needs to be kept in memory for 1+4 minutes
- This means that only $\frac{1}{5} \times 262144 = 52428.8$ connections are active.
- The server is using 80% of its memory for connections in `time_wait` state, that are already over.
- Operating Systems reduce this to avoid memory occupation, Linux the timeout to 60s (see the source)

TCP

↳ 2.2 Reliable Transfer

Go-back-n + Selective Repeat

- TCP in its original protocol uses a go-back-n transmission with a selective repeat receiver using only cumulative acks
- There are several tweaks that are worth looking at, and some extensions that modified its behavior
- The `TCB` maintains all the variables needed to maintain the state of the connection.

TCB

The contents of the TCB that are static are[1]:

- the local and remote IP address, and ports. These never change during the lifetime of the connection
- `sending buffer`: a buffer used to store all unacknowledged data
- `receiving buffer`: a buffer to store all data received from the remote host that has not yet been delivered to the user. Data may be stored in the receiving buffer because either it was not received in sequence or because the user is too slow to process it

buffers are allocated at the beginning of the connection and filled/emptied when data arrive/depart.

[1] For the curious ones, Linux stores the TCB data in the sock struct: see this link

TCB

The contents of the TCB that change during the evolution of the connection are:

- the current state of the FSM (SYN-Sent, SYN-Received, Established. . .)
- the `maximum segment size` (MSS)
- `snd.nxt` : the sequence number of the next outgoing byte in the byte stream (the first byte of a new outgoing data segment uses this sequence number)
- `snd.una` : the earliest sequence number that has been sent but has not yet been acknowledged
- `snd.wnd` : the current size of the sending window (in bytes)
- `rcv.nxt` : the sequence number of the next byte that is expected to be received from the remote host
- `rcv.wnd` : the current size of the receive window advertised by the remote host

Sending Data

When the application puts some new data in the sending buffer the TCP layer...

- checks that the sending buffer does not contain more data than the receive window advertised by the remote host (`rcv.wnd`).
- up to MSS bytes of data are placed in the payload of a TCP segment.
- The sequence number of this segment is set to `snd.nxt` and the `snd.nxt` variable in the TCB is incremented by the length of the payload of the TCP segment.
- The acknowledgment number is set to `rcv.nxt` and the window field of the TCP segment is computed based on the free space left in the receiving buffer (not the sending buffer!).
- The data is kept in the sending buffer in case it needs to be retransmitted.

Receiving Data

When a new segment arrives, the TCP layer checks if the `ACK` bit is set, in case it...

- sets `rcv.wnd` to the value of the window field of the received segment.
- compares the acknowledgment number to `snd.una`.
- The newly acknowledged data is removed from the sending buffer and `snd.una` is updated.
- The sending buffer may or may not be empty now.

Receiving Data

If the TCP segment contains data then receiver will . . .

- compares the sequence number to `rcv.nxt`.
- If they are equal, the segment was received in sequence, data *can be* delivered to the user and `rcv.nxt` is updated (not necessarily they will be delivered now).
- The contents of the receiving buffer is checked to see whether other data already present in this buffer can be delivered in sequence to the user.
- If so, `rcv.nxt` is updated again.
- The segment's payload is placed in the receiving buffer
- send the `ACK` with the updated window size.

That's all folks...

- Ideally, all you need to know to implement TCP is what I have described so far
- However, there are many design choice that are fundamental to make TCP (and networks that use it) work. Some of these are:

Design choices

1. When should I send data?
2. How big should the window be?
3. How should I set the retransmission timers?

TCP

↳ 2.3 TCP Internals: Sender Side

1: When should I Send data?

- The application is filling the buffer with data, should the sender just send them as they arrive?
- This question seems simple, however, there are non-trivial choices to be made.
- What do you think are the two extreme approaches and what are their pros/cons?

Option A) Send ASAP

- Ideally as soon as the sending buffer is not empty, you could send a TCP segment
- In some cases (like in the remote shell mentioned before) this is a desired behaviour.
- However, a TCP packet has at least 40B of headers (20 IP + 20 TCP)
- If you send one byte per segment, $\frac{40}{41}$ of the capacity is wasted.

Option B) Wait to have at least MSS to Send

- In this case you obtain the maximum efficiency
- If your application is producing data with a very high throughput, this is probably a good choice.
- If instead your buffer takes a lot of time to be filled enough, data inside it become old before you send them.

The Nagle Algorithm

A trade-off comes from the Nagle Algorithm, that works as follows

```
if  len(data) >= MSS and rcv.wnd >= MSS:
    send one MSS-sized segment
else:
    if there are unacknowledged data:
        place data in buffer until acknowledgment has been received
    else:
        send one TCP segment containing at most rcv.wnd data
```

The Nagle Algorithm

A trade-off comes from the Nagle Algorithm, that works as follows

```
1 if  len(data) >= MSS and rcv.wnd >= MSS:
2     send one MSS-sized segment
3 else:
4     if there are unacknowledged data:
5         place data in buffer until acknowledgment has been received
6     else:
7         send one TCP segment containing at most rcv.wnd data
```

lines 1-2: Check that the receiver window allows `MSS` data, and if I have them to be sent, just do it.

↪ This allows bulk transfers: if I have many big segments to send, I will just send them.

The Nagle Algorithm

A trade-off comes from the Nagle Algorithm, that works as follows

```
1 if  len(data) >= MSS and rcv.wnd >= MSS:
2     send one MSS-sized segment
3 else:
4     if there are unacknowledged data:
5         place data in buffer until acknowledgment has been received
6     else:
7         send one TCP segment containing at most rcv.wnd data
```

lines 1-2: Check that the receiver window allows `MSS` data, and if I have them to be sent, just do it.

↪ This allows bulk transfers: if I have many big segments to send, I will just send them.
What if the receiver wants me to slow down?

The Nagle Algorithm

A trade-off comes from the Nagle Algorithm, that works as follows

```
1 if  len(data) >= MSS and rcv.wnd >= MSS:
2     send one MSS-sized segment
3 else:
4     if there are unacknowledged data:
5         place data in buffer until acknowledgment has been received
6     else:
7         send one TCP segment containing at most rcv.wnd data
```

lines 1-2: Check that the receiver window allows `MSS` data, and if I have them to be sent, just do it.

$\hookrightarrow$ This allows bulk transfers: if I have many big segments to send, I will just send them.
What if the receiver wants me to slow down?
$\hookrightarrow$ the receiver can slow this down in an `ACK` packet sending a smaller window

The Nagle Algorithm

A trade-off comes from the Nagle Algorithm, that works as follows

```
1 if  len(data) >= MSS and rcv.wnd >= MSS:
2     send one MSS-sized segment
3 else:
4     if there are unacknowledged data:
5         place data in buffer until acknowledgment has been received
6     else:
7         send one TCP segment containing at most rcv.wnd data
```

line 3: Else, the bytes I have to send are less than `MSS`, or the receiver window is small

↪ This is when I have to take a wise decision, is data big enough to be worth sending?

lines 4-5: If i am waiting for an `ACK` on old data, wait some more, so the buffer may fill up. Send data when the `ACK` is received

lines 6-7: Else, i have not been sending data so far, or they all were acked. I am allowed to send a small segment, but only once per RTT.

Nagle Consequences

- The Nagle algorithm makes it more likely that big segments are sent.
- In fact, measurements done on the Internet observed that the majority of the packets are either 1460B long, or 0B.
- 1500B is the MTU of the Ethernet link layer, the most used, so the majority of the TCP connection use it as MSS
- 0B is the size of the TCP `ACK`.
- However this makes TCP unsuitable for real time traffic, which needs Nagle's algorithm to be disabled.

2: How Big Should the Window Be?

- Assume a connection is open and the sender sends many segments, filling the sender window
- It will stop and wait for an `ACK`
- Assume the `ACK` arrives and acks all the data sent so far
- Then the sender will start sending again

In practice, you can not have more than one window of data sent per RTT, but the maximum window size is set to $2^{16} = 65536B$

TCP Maximum Speed

RTT	Maximum Throughput
1 msec	524 Mbps
10 msec	52.4 Mbps
100 msec	5.24 Mbps
500 msec	1.05 Mbps

Clearly these numbers are not up-to-date with today's capacity

Key TCP Option 2: Window Scale

- Since the TCP header can not be extended, the window size can not be enlarged
- A solution that was adopted is to add another Optional Extension that introduces a multiplier for the window
- The TCP Window Scale Option is exchanged only in `SYN` and `SYN-ACK` packets and contains a number $0 \leq S \leq 14$
- The window size number is shifted of S bits, that is, the receiving window size is $\texttt{rcv.wnd} * 2^S$

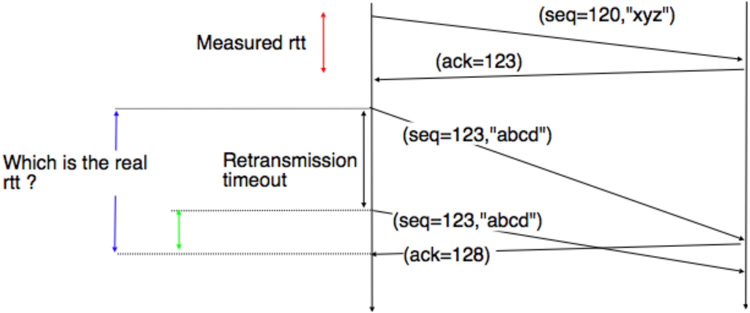
Current Speed

RTT	Maximum Throughput
1 msec	8590 Gbps
10 msec	859 Gbps
100 msec	86 Gbps
500 msec	17 Gbps

3: How Should I set the Retransmission Time Out (RTO)?

- Ideally, RTO is just slightly larger than an RTT
- If by the end of an RTT timer the ACK did not arrive, then I should retransmit.
- However, how do I know the RTT? I have to measure it.
- And this has two challenges:
  1. *Measure Error*: measuring the RTT in a lossy connection is subject to errors
  2. **Value Smoothing**: RTT just changes during a connection lifetime and we have to smooth it

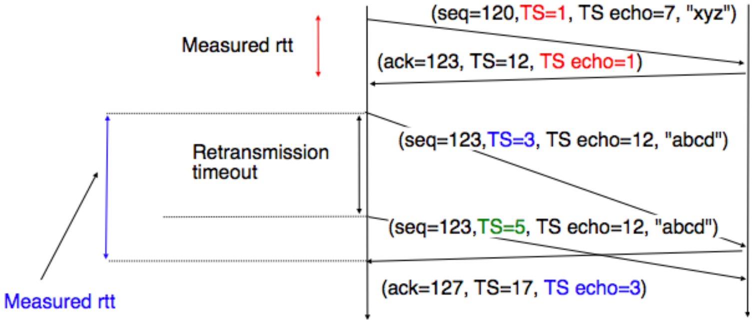
Example of Measure Errors



Key TCP Options 3: TCP Timestamps

- As a first, simple countermeasure, just ignore the RTT measured on retransmission
- However, what if the `ACK` is the one that got lost, or was duplicated?
- The second solution is another TCP Optional Extension, TCP Timestamps
- In every TCP segment, two timestamps are added, the *current* one and last one received by the other endpoint
- This allows to compute the RTT from the last received segment, no matter if a retransmitted one or not

TCP Timestamps

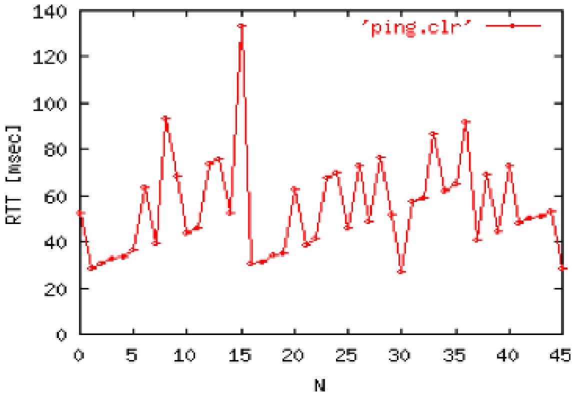


Timestamps and MSL

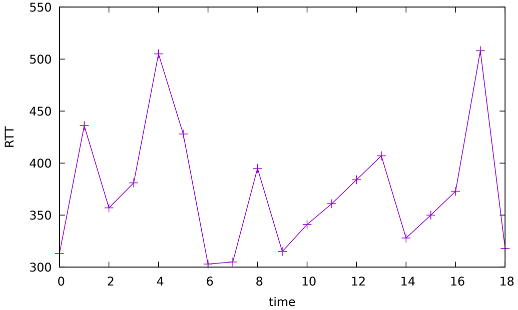
- You should remember that the sequence numbers should not wrap before MSL
- In fast networks, 32 bit sequence numbers would wrap before MSL
- TCP timestamps also solve this problem, if a segment is highly delayed, even if the sequence number wrapped in the meantime, and there could be an ambiguity, the timestamp will reveal that the segment is old.

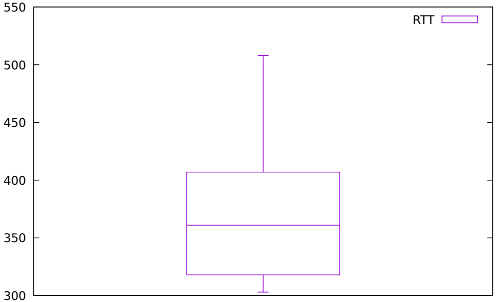
Smoothing the RTT, Setting the RTO

Assuming I can measure the RTT, how do I smooth the measurement?



Variable RTT (today): `ping 202.232.2.191`





Smoothing the RTT Estimation

- The algorithm used to smooth RTT to take into account its variability is due to Van Jacobson
- It introduces three variables:

  rtt: the last measured RTT value

  $srtt$: the smoothed RTT, initialized to the first rtt

  $rttvar$: the estimated deviation from the average of the RTT, initialized to $rtt/2$

  rto: the retransmission timer

Van Jacobson Algorithm: details

- When you have a series of samples s_i, and $0 < \alpha < 1$ the value

$$\bar{s} = (1 - \alpha)\bar{s} + \alpha s_i$$

  it's called exponential moving average (EMA) of the sequence s_i.
- the more $\alpha \simeq 1$ the more the last sample counts more than the so-far accumulated average.

Van Jacobson Algorithm

Before measuring any RTT, set RTO to a value smaller than 1s. When the first sample rtt is obtained then set:

$$srtt = rtt; \quad rttvar = rtt/2$$

At any new sample of measured RTT this is how the values are updated:

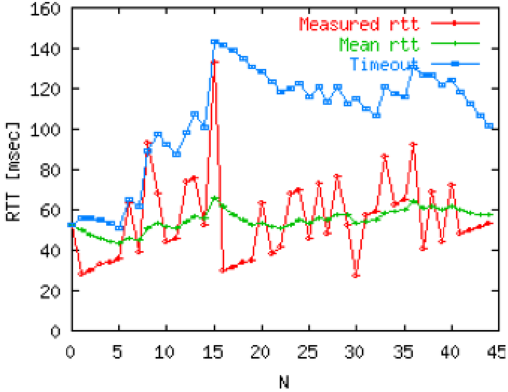
$$srtt = (1 - \alpha) \times srtt + \alpha \times rtt$$

$$rttvar = (1 - \beta) \times rttvar + \beta \times |srtt - rtt|$$

$$rto = srtt + 4 \times rttvar$$

Where $\alpha = 1/8$, $\beta = 1/4$.

RTO



- Once we have an estimation of the RTO, we need to adjust it to the network conditions
- Several modifications were done during the years to fine-tune this process
- Not all of them, work well together :-|

RTO Exponential Backoff

- RTO exponential scaling (RFC 6298): Included in the original TCP RFC, this mandates that the RTO is doubled after every failed attempt.
- After a successful attempt the `RTO` is reset as per the normal algorithm
- The rationale is: if the receiver is congested, it can not handle too many packets/s. Then the sender should slow down.

Fast Retransmit

- The sender may receive reordered acknowledgments, so for instance the sender sent up to seq=X, but it receives seq=Y<X
- It should conclude that the segment got lost and should retransmit seq=Y.
- However, it may just be that the two acknoledgments are inverted, and seq=X is on its way
- Fast retransmit mandates that retransmission before timer expiration can be done after 3 received copies of the same ACK.
- That is, even if the *rto* is not expired, but the sender receives three copies of the same ACK, probably this means that a packet got lost, and this will trigger a retransmit.

TCP

↳2.4 TCP Internals: Receiver Side. Delayed ACKs and Selective ACKs

Delayed ACKs

- Most of the times, communication is largely mono-directional
- The receiver will then often send a lot of `ACKs` that are very inefficient
- The Delayed ACKs strategy sends one ACK every second data frame received, or, after a certain timeout
- This makes it possible to acknowledge more data segments than answering at every single received segment
- This does not work well with Nagle's algorithm. Why?

Selective ACKs

- go-back-n always underperforms when there are high losses, because the window does not move
- in TCP the receiver is allowed to save out-of-sequence data, but should still acknowledge only the ones in sequence
- with the SACK option a new optional header extension is added, in which the receiver acknowledges contiguous, out of sequence data blocks it received.
- Due to space reason, not more than 3 blocks are included

Selective ACKs

- For instance, the ACK may contain something like: $[1100 - 1200], [1250 - 1500], [1800 - 1900]$
- The ACK will also contain the acknowledgment number in the fixed TCP header, for instance 850.
- This is interpreted by the sender as: *the receiver received correctly all the bytes before 850, and the intervals 1100-1200, 1250-1500, 1800-1900*.
- It is the sender's choice to decide what to do next.

Avoiding RST attacks

- An attacker may try to reset a TCP connection between two peers
- If the attacker is on-path (some router on the path between the peers), then this is impossible to stop. The attacker knows all the correct parameters to be put in the header
- If the attacker is not on-path, he may try to guess them
- To make this attack less likely, it is important that `RST` segments acknowledge the last valid sequence number, as shown in the previous image.