49202 Communication Protocols

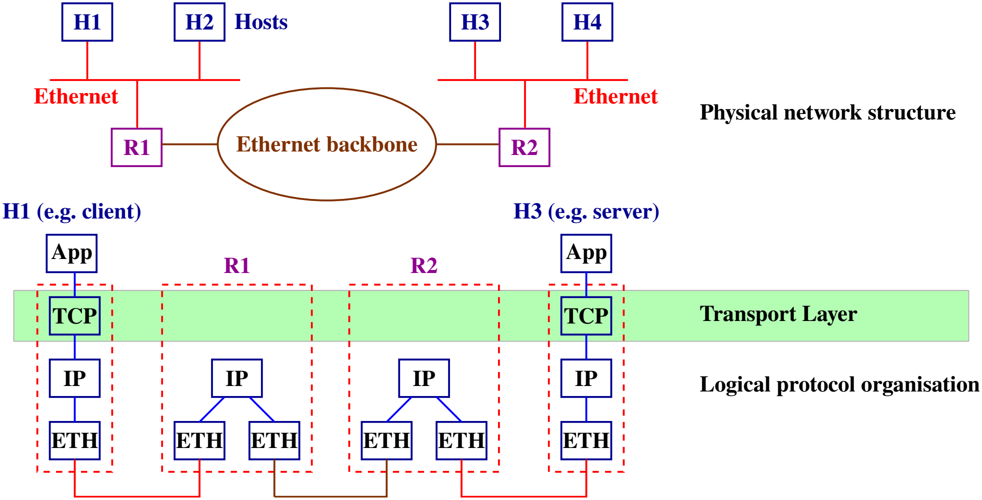
The Transport Layer

Daniel R. Franklin

Faculty of Engineering & IT

March 12, 2024

Transport layer protocols



Design issues

- The aim of the transport layer is to provide **appropriate** services to the application layer
- It builds on top of the services that are provided by the network layer - and provides important functions which are missing at this layer
- The network-layer service model is **best effort**:
  - IP packets may be lost, reordered, duplicated etc.
  - Is this a problem? Maybe, maybe not. It depends on the application.
  - Therefore: we offer a range of transport layer options to suit different sets of application requirements.
- It is remarkable that two transport-layer protocols - Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) - are sufficient for the vast majority of Internet data traffic.

Expectations

- What might we expect the transport layer to do?
- There are various functions that we might like to have - for example:
  - Directing traffic to the right application - **application multiplexing/demultiplexing**
  - Guaranteed message delivery (or at least, a mechanism to validate messages...)
  - Guaranteed delivery of messages in the same order they are sent
  - Ensuring delivery of at most one copy of the message (no duplication)
  - Transmitting data as quickly and efficiently as possible, **without** overwhelming booth the **receiver** OR the **network** - two quite different problems!
- Expectations depend on the specific requirement of the application...

Transmission Control Protocol (TCP)

- One of two principal transport layer protocols provided by the TCP/IP stack
- The other is UDP (User Datagram Protocol) - to be discussed later!
- There are others - such as Stream Transmission Control Protocol (SCTP) but they are highly specialised and rarely seen outside of specific contexts
- Occasionally, “applications” may use IP directly (such as ICMP - for example, `ping` and `traceroute` - but are these really applications?)
- Which applications use TCP?
  - TCP is good for **loss-sensitive, delay-tolerant** applications - web traffic, e-mail, file transfers, authentication...

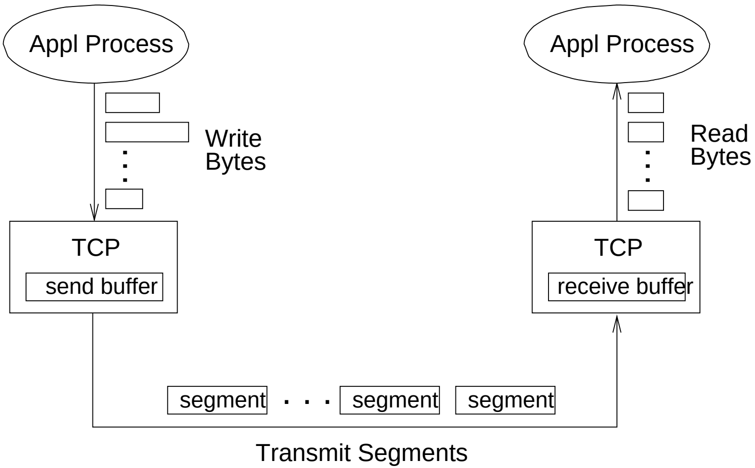
TCP service model

- The TCP service model is **connection oriented**
  - The end hosts must establish a connection before they exchange data
  - This requires that both ends maintain a consistent **state**
  - The endpoints transition between different states over the establishment, data transfer and termination phases of the connection
  - TCP is **full duplex**: data can be exchanged in both directions once the connection is established
- Importantly: TCP is **byte-stream oriented** rather than **message oriented**
  - This means that the receiver expects to receive a **contiguous stream** of bytes - you can think of the byte stream like a file being written to by the sender and read by the receiver.
  - Unlike a file, you can't rewind and go back (but the application can do whatever it wants after it has read the stream)
  - The application layer protocol can add whatever structure it wants on top of this

TCP service model

- **Reliability** means that TCP
  - Guarantees data delivery - if any part of the message is **lost** or **corrupted**, it will be **resent**
  - Delivers data in the correct order, with no gaps - the receiving application will be able to read data up to the **last byte of contiguous byte stream**
  - Out-of-order parts of the message will be buffered by the operating system but **not** delivered to the application until the missing parts arrive.
- TCP also provides two **feedback mechanisms** with the following objectives:
  - **Flow control**: Do not overload the **receiver** with too much data
  - **Congestion control**: Do not overload the **network** with too much data
- These mechanisms still try to transmit data as fast as possible while avoiding overflow or congestion.
- Congestion control in particular is not a completely solved problem.

How TCP manages a byte stream



TCP Segments

- The byte stream is not sent as a continuous stream due to limitations of the underlying network
  - There is a limit on the maximum amount of data (the **maximum transmission unit** or **MTU**) which can be transmitted at a time in every Layer 2 network protocol - e.g. Ethernet defaults to 1500 bytes of payload
  - With 20 bytes (minimum) needed for Layer 3 (IPv4) headers (40 bytes for IPv6), and a TCP header size of 20 bytes, that leaves at most 1460 bytes available for TCP's payload
- TCP must therefore divide the byte stream into chunks known as **segments** which do not exceed this size (known as the **maximum segment size** or **MSS**

TCP Segments

- TCP transmits a segment when:
  - The amount of data which has been written to its transmission buffer reaches the MSS (**Nagle's algorithm**);
  - If the socket is created with the `TCP_NODELAY` option, as soon as **anything** is written to the buffer (i.e. disabling Nagle's algorithm);
  - Explicitly triggered by the application, e.g. via the **push** operation; or
  - When a periodic timer expires (potentially even if there's no data to send!)
- The MSS is chosen to preemptively avoid the need for **fragmentation** of datagrams at Layer 3.
  - MSS = MTU - IP header size - TCP header size (where MTU = smallest MTU of any network on the entire end-to-end path)

TCP ports

- TCP uses **port numbers** to identify the client and server applications
- Many ‘**well-known**’ ports are used to identify **server applications**
  - Example 1: **HTTPS** servers listen on TCP port number 443
  - Example 2: **SMTP** mail servers listen on port number 25
- Well-known TCP port numbers are between 1 and 1023
- Have a look at the file `/etc/services` on a Linux machine (e.g. in the lab VM) to see the full list of well-known ports.
- Some protocols (such as DNS) also listen on the corresponding UDP port.

TCP ports

- The client (i.e. the initiator of a TCP connection) chooses a different, arbitrary port number as its endpoint for each TCP session
  - These ports are known **ephemeral** (i.e. short lived) ports
  - They may take on any value between 1024–65535, although the Internet Assigned Numbers Authority (IANA) **recommends** that ports in the range 49152–65535 are reserved for dynamic or private ports
  - The exact range which is used is implementation-dependent (e.g. Linux generally uses 32768–60999; Windows uses different ranges depending on the version!
- Ephemeral ports also change with each connection, even to the same service.

TCP ports

- Each TCP connection is uniquely identified by the quadruple

  (Source IP address, Source TCP port, Dest IP address, Dest TCP port)
- Example: A client with IP address **IP1** opens two simultaneous TCP connections, each from a different browser tab, to a web site with IP address **IP2**
- The operating system allocates ephemeral ports 37869 and 37870 for these two client processes
- These two TCP connections are represented by:
  1. (IP1, 37869, IP2, 80)
  2. (IP1, 37870, IP2, 80)
- The pair of (IP address, TCP port number) is also known as a **socket**

TCP header format

<table>
<tr><td>0</td><td>4</td><td>10</td><td>16</td><td>31</td></tr>
<tr><td colspan="3">Source Port</td><td colspan="2">Destination Port</td></tr>
<tr><td colspan="5">Sequence number</td></tr>
<tr><td colspan="5">Acknowledgment number</td></tr>
<tr><td>HL</td><td>Unused</td><td>Flags</td><td colspan="2">Advertised window</td></tr>
<tr><td colspan="3">Checksum</td><td colspan="2">Urgent Pointer</td></tr>
<tr><td colspan="5">Options (0 or more 32 bit word)</td></tr>
<tr><td colspan="5">Data</td></tr>
</table>

HL = Header length

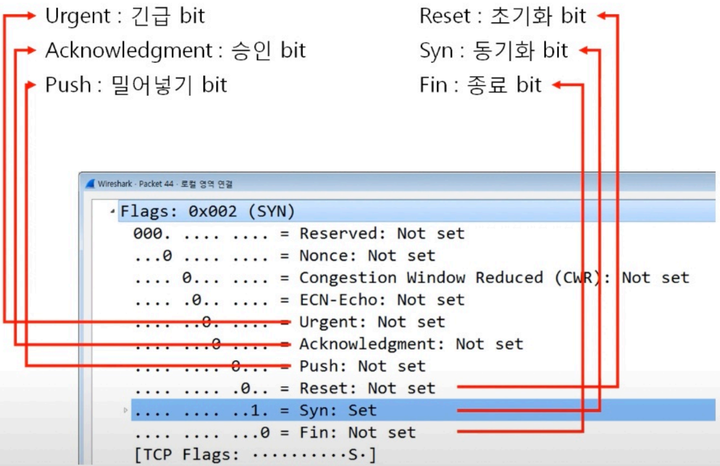
Flags = URG, ACK, PSH, RST, SYN, FIN

TCP data offset

- The TCP data offset is needed as the TCP header size is variable - if options are included in the header structure, it may be longer than 20 bytes (have a look at some `iperf3`) TCP headers)
- It indicates the position of the start of the payload, as a multiple of 32 bits (4 bytes), from the start of the packet.
- As it is 4 bits long, the maximum value of this field is 15; therefore, the maximum header length is $15 \times 4 = 60$ bytes.

TCP flags

- Another important field is the set of 1-bit flags. Flags are either on (1) or off (0); currently the following set is defined:
  - CWR: Congestion Windows Reduced (see below - means we are limiting transmit rate due to ECN)
  - ECE: Explicit Congestion Notification (ECN) - Echo (there's congestion in the network!)
  - URG: Urgent (almost never seen today, used for out-of-band urgent data)
  - ACK: Acknowledgement (data was successfully recieved)
  - PSH: Push (segment was sent early rather than waiting for a full MSS)
  - RST: Reset (various reasons such as fast close)
  - SYN: Synchronise (connection setup)
  - FIN: Finish (connection teardown)



Urgent : 긴급 bit
Acknowledgment : 승인 bit
Push : 밀어넣기 bit
Reset : 초기화 bit
Syn : 동기화 bit
Fin : 종료 bit
Wireshark · Packet 44 · 로컬 영역 연결
Flags: 0x002 (SYN)
000. .... = Reserved: Not set
...0 .... = Nonce: Not set
.... 0... .... = Congestion Window Reduced (CWR): Not set
.... .0.. .... = ECN-Echo: Not set
.... ..0. .... = Urgent: Not set
....0 .... = Acknowledgment: Not set
.... 0... = Push: Not set
.... .0.. = Reset: Not set
....1. = Syn: Set
....0 = Fin: Not set
[TCP Flags: ··········S·]

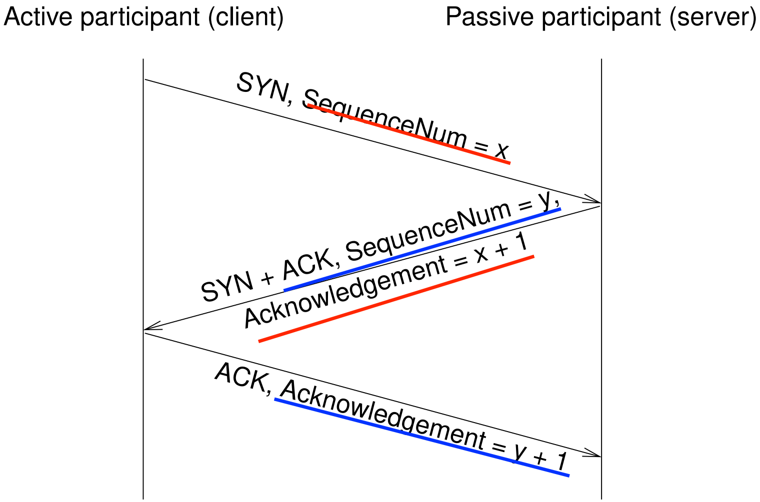
TCP header checksum

- **Checksums** are an important mechanism for validating data - they are used at Layer 2, 3 and 4 (and maybe 5!)
  - The checksum is a mathematical function of a data block of **arbitrary** length; the result has a **fixed** length
  - The algorithm is designed such that small changes to the input data block (single bit inversions for example) cause the output to be completely different
  - Hence, if a receiver computes the checksum on the same data block, and gets a different result to the one found in the header, we can conclude that either the block or the checksum was **corrupted** and can discard this segment
- The TCP checksum is computed over the TCP header, TCP data and the **pseudoheader**
- The **pseudoheader** consists of IP source and destination addresses, TCP segment length, and the protocol field from IP header

TCP connection establishment

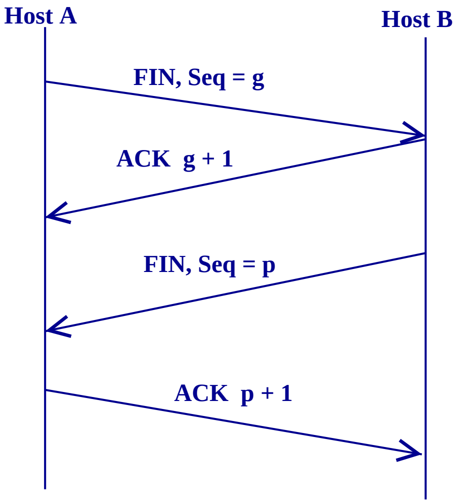
- As TCP is connection-oriented, a **connection establishment procedure** is used to set up the connection
- Based on a 3-way handshake:
  1. The client initiates the connection with an empty (no payload) TCP segment whose **SYN** (synchronisation) flag is set
     - The initial client-to-server **sequence number** is **random**
  2. If the server is listening on the destination port, it will reply to the client with an empty TCP segment with the **SYN** and **ACK** (acknowledgement) flags set.
     - The server-to-client **acknowledgement number** sent in response to a SYN packet is always one plus the sequence number on that segment
     - The segment also will have a **different, unrelated** random initial server-to-client sequence number.
  3. Finally, the client sends back an acknowledgement (empty segment, ACK flag set). The connection is now in the **ESTABLISHED** state.

TCP 3-way handshake



TCP connection termination

- Two independent handshakes
- Termination-initiating end sends a segment with the **FIN** flag set
- Response has **ACK** flag set
- A second segment is sent in the same direction with **FIN** flag set
- And the final segment has **ACK**



TCP flow control

- Flow control: sender wants to send as much data as possible without overloading the receiver
- During connection start up: receiver puts its buffer size (in bytes) in the `AdvertisedWindow` header field
- The first round: The sender can send no more than `min(availabledata, AdvertisedWindow)`
- The receiver acknowledges the packets (if they have indeed been received) and informs the sender the current available buffer size using the `AdvertisedWindow` field of the ACK packet (which may have no payload)
- The sender can now send no more than `min(available data, AdvertisedWindow - amount of unacknowledged data`

TCP를 이용한 통신과정

데이터 송수신 과정

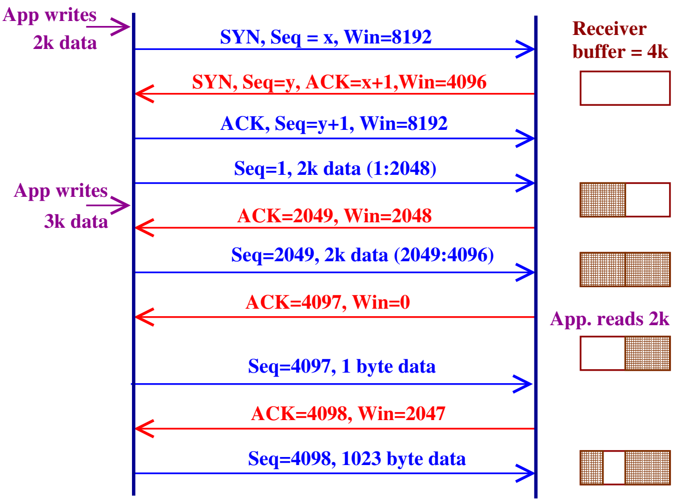
TCP를 이용한 데이터 통신을 할 때 단순히 TCP 패킷만을 캡슐화해서 통신하는 것이 아닌 페이로드를 포함한 패킷을 주고 받을 때의 일정한 규칙

1. 보낸 쪽에서 또 보낼 때는 SEQ번호와 ACK번호가 그대로다.

2. 받는 쪽에서 SEQ번호는 받은 ACK번호가 된다.

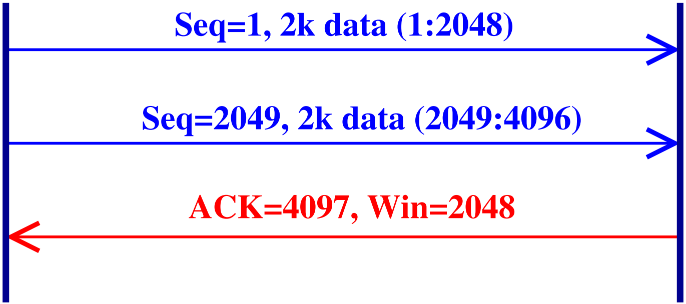
3. 받는 쪽에서 ACK번호는 받은 SEQ번호 + 데이터의 크기

TCP Flow Control - Example



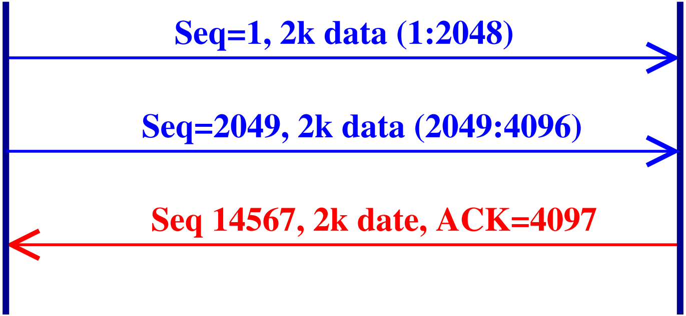
TCP data acknowledgement

- The acknowledgement contains the sequence number expected next
  - "Acknowledgement # = $x + 1$" = "I've received everything up to sequence number x"



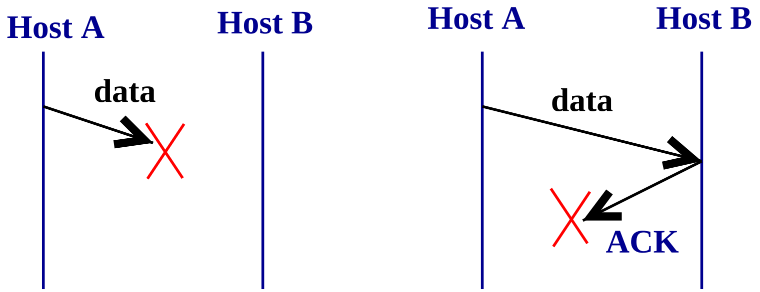
TCP ACK piggybacking

- Acknowledgement may also be included in a data packet, by setting the ACK flag in the header
- Rather than immediately sending the ACK, the acknowledging party delays until it has data to send
- This is only of benefit if data is simultaneously being streamed both ways!



TCP error control

- Basic mechanism: each end **acknowledges** data received from the other end
- However, both data segments and acknowledgements can get lost (perhaps due to congestion)



TCP error control

- When the sender transmits a segment, it sets a timeout value
  - “If I haven’t heard from the receiver and the timeout expires, I’ll retransmit the data”
- Determining a sensible timeout value for a single point-to-point link is easy:
  - Propagation delay in each direction is fixed and known beforehand
  - Bandwidth (how many bits we can transmit per second, e.g. 100 Mb/s) is known
  - Therefore, the timeout can be set to a fixed value

Determining a suitable timeout for TCP

- On anything other than a static point-to-point link, it is **not** that simple
- The two end hosts can be anywhere, with many intermediate hops over different networks
  - The propagation delay between them is not known beforehand, and may change (e.g. changing routes, LEO satellites)
  - Additional delay is introduced at congested routers - segments must wait in a queue which varies in length all the time
  - The timeout mechanism **must** work no matter where the two end hosts are located
- Conclusion: **we can’t set it to a fixed value**
- TCP uses a statistics-based **adaptive** mechanism to determine the value of timeout
- Timeout value continually adjusts during a TCP session

Adaptive retransmission

- If everything goes well, the sender expects an ACK one RTT (round trip time) after it has sent the packet (there and back again)
- We can measure each RTT by calculating the difference between time at which data is sent and the corresponding acknowledgement is received
- However, because the round trip time includes several random components due to queueing delays at routers, the actual RTT varies from segment to segment - possibly quite a bit
- We can, however, estimate some statistics that allow a sensible timeout to be determined

Adaptive retransmission

- The long-term average RTT is typically estimated via a moving-average filter: the current estimate is equal to

$$\overline{RTT_n} = (1-a) \times \overline{RTT_{n-1}} + a \times RTT_{measured}$$

where $0 \leq a \leq 1$ can be tuned according to the desired 'rate of forgetting'.

- We can also estimate the variability of the RTT using the mean deviation:

$$Mdev_n = (1-b) \times Mdev_{n-1} + b \times |\overline{RTT_n} - RTT_{measured}|$$

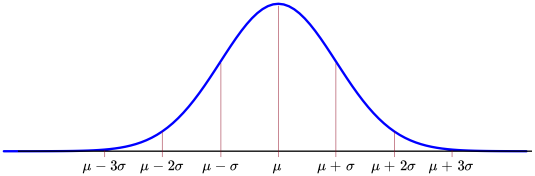
- $0 \leq a \leq 1$, $0 \leq b \leq 1$ can be tuned for the desired 'rate of forgetting' for each.

Adaptive retransmission

- The receiver timeout is then set to be

$$RTO_n = \overline{RTT_n} + 4 \times Mdev_n$$

- If the round trip time follows a Gaussian distribution (not really accurate but good enough) then the probability of a packet actually arriving **after** the calculated time-out is extremely small (of the order of 0.003%)
- Of course packet arrival statistics usually don't follow such nice distributions - but good enough to illustrate the point.



TCP congestion control

- TCP sources adjust their sending rates to avoid congesting the network
- Two questions from the sources' perspective:
  1. Am I congesting the network?
  2. If so / if not, how should I adjust my sending rate?
- **Timeouts** are used as an indication of congestion (there are others...)
  - Timeout signals that a packet was lost
  - Packets are only very rarely lost due to transmission error
  - Lost packets imply the onset of **network congestion**

Congestion control mechanism

congestion = 혼잡, 정체, 지연

- TCP maintains a variable called `CongestionWindow`
- `CongestionWindow` = Maximum number of bytes that a TCP source can send without congesting the network
  - It's simpler to think in terms of packets...
- You can think of `CongestionWindow` as being similar to `AdvertisedWindow` (used for flow control), but for the routers in between rather than the remote end-point.
- The key difference is that we cannot directly measure the state of the intermediate buffers (since TCP is an end-to-end protocol and leaves the problem of routing to IP).
- Thus we need to infer the state of these routers' queues - this is the problem of congestion control.

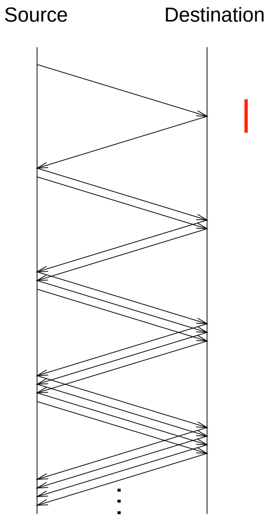
Additive increase / multiplicative decrease (AIMD)

- Additive increase / multiplicative decrease is used once the connection is established. The fundamental idea is to:
  - Slowly increase `CongestionWindow` (allow more data to be 'in flight' at any time, unacknowledged) if there is no congestion
  - Rapidly decrease `CongestionWindow` (reduce the amount of unacknolwedged data permitted) if we think there is congestion
- Objective: adjust the packet transmission rate according to changes in available bandwidth - increasing cautiously and decreasing aggressively
- In general this is done by
  - **Incrementing** `CongestionWindow` by **one segment per RTT** (additive increase)
  - **Dividing** `CongestionWindow` **by two** whenever a timeout occurs (multiplicative decrease)

Example of multiplicative decrease

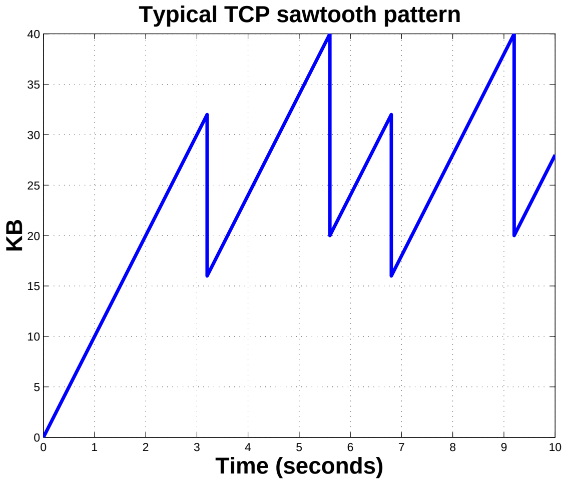
- A timeout occurs when the `CongestionWindow` is 16
  - `CongestionWindow` reduces to 8
- If another timeout occurs, then `CongestionWindow` reduces to 4
- If the next packet is successfully ACK-ed, increase by 1

Additive increase



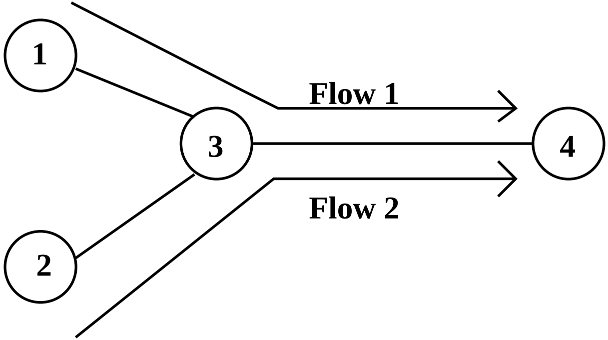
- In this example the number of packets that we allow to be ‘in flight’ (unacknowledged) starts at 1
- When this is acknowledged, we increase the limit to 2
- When these are acknowledged, we increase to 3 and so on
- In reality we use ’number of bytes’ not ’number of packets’ as the metric

Sawtooth behaviour



Why multiplicative decrease?

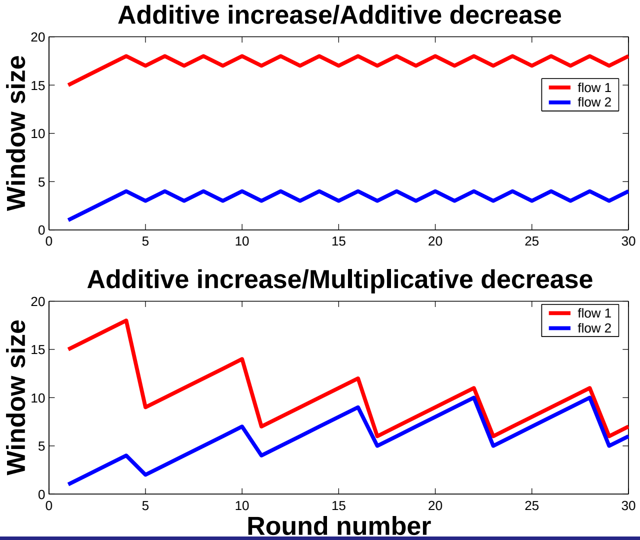
- The network is shared by many users
- Multiplicative decrease ensures fairness between users
- Fairness: Not easy to define
  - For the situation below, it means equal share of link bandwidth



Fairness Comparison

- Theoretical modelling is difficult, and gets harder as networks become more complex. Instead, we can test different schemes via simulation:
  - Additive increase/additive decrease (AIAD)
  - Additive increase/multiplicative decrease (AIMD)
  - Multiplicative increase/multiplicative decrease (MIMD)
  - Multiplicative increase/additive decrease (MIAD)
- Result: AIMD is fair, but not AIAD, MIMD or MIAD

AIAD versus AIMD



Summary: flow control and congestion control

- Flow control:
  - Aim: avoid overloading the receiver
  - Uses `AdvertisedWindow` from receiver
- Congestion control:
  - Aim: not overloading the **network** (i.e. overflowing the buffers of routers in between the two ends of the network)
  - Uses `CongestionWindow` maintained by the sender
  - `CongestionWindow` is adjusted by AIMD and other mechanisms to be introduced...

Flow control + congestion control working together

- Flow and congestion control algorithms do **not** work in isolation. So, instead:
  - Define `MaxWindow = min(CongestionWindow, AdvertisedWindow)`
- The sender can send no more than difference between `MaxWindow` and the amount of unacknowledged data.
- This is the basic mechanism of combined flow and congestion control - avoid overloading EITHER the receiver OR the network.

Summary

- TCP
  - Service model
  - Flow control
  - Adaptive retransmission
  - Congestion control / avoidance
- Next: improvements! Smarter congestion control! Also, UDP