10 OSI Layer 4 – Transport Layer

10.1 Goal of this section

The goal of this section is for readers to understand the transport layer and what services and functions it provides – and to understand the two most ubiquitous transport layer protocols which are User Datagram Protocol (UDP) and Transmission Control Protocol (TCP). By comparing and contrasting these, the reader should also understand some of the trade-offs that occur at this layer, and be able by comparison to pick up and understand other transport protocols.

10.2 Purpose of the transport layer

The transport layer provides a service link between two applications running on hosts, rather than just between two network endpoints, with well-defined transport guarantees on the link.

10.3 Interfaces up and down

The interface to the network layer below is a packet of data along with the network addressing information required to deliver that packet to its destination.

The interface to the application layer above depends on the protocol in use. The generic sockets interface is the clearest example of this interface, which was covered in section 6 as well as in the first lab practical.

10.4 Services

In addition to application disambiguation to allow multiple links ending on the same endpoint, the transport layer offers a set of transport guarantees that can be useful to applications – with different transport protocols offering different services. Typically, these extra services come at an additional cost of resources (additional latency and/or processing) and so applications will choose a protocol to use dependent on the features they need. Typical services include the following:

10.4.1 Handling and error-correcting of corruption/short-scale network failures

Lower level layers can detect corruption in their data or can lose packets of data themselves. In these cases, he transport layer handles retransmitting the data to ensure a clean version of it arrives.

10.4.2 Guaranteed ordering

Lower layers do not guarantee the order that packets are received in. For example, if application A sends packets 1, 2, 3, they might be received in a different order (say 1, 3, 2). For clarity – the ordering is *often* maintained much of the time by lower layers, but it isn't guaranteed. The transport layer can provide a guarantee that if it is given packets in a particular order, they will arrive out of the transportation layer at the far end in that order.

10.4.3 Proof of delivery

While the transport layer cannot guarantee delivery in all circumstances (Someone unplugging a cable can prevent that.), it can provide proof of delivery, so a source application knows that its message has got through.

10.4.4 Flow and congestion control

The transport layer can rate-limit the flow of data over links, which can be used to achieve two things. **Flow control** is when a sender and receiver rate-limit the flow of data to prevent the receiver becoming overloaded. **Congestion control** is when a sender and receiver rate-limit the flow of data to prevent the underlying network resources becoming overloaded.

10.5 IP Addresses and Ports

Transport protocols deliver data between source and destination applications. While IP addresses are sufficient to identify a particular network node, we need additional disambiguation to reach a destination application. The transport layer provides this with the concept of a **port**. A port is an identifier that on an endpoint uniquely defines a particular connection. In implementation, a port is simply a 16-bit number.

Port numbers 0-49151 are **reserved** ports. These have well-defined uses assigned by IANA, and split into two sets:

- 0-1023 are **well known** (also called **system**) ports. Historically, these had extra restrictions, such as only being used by applications with root privileges, etc. For example, Port 22 is used for Secure Shell (ssh)
- 1024-49151 are **registered** (also called **user**) ports. For example, Session Initiation Protocol (used for setting up IP phone calls) runs on port 5060. In reality, above port number 12k or so, this space is sparsely populated – and there are many port numbers that are "squatted on" by companies in this space and are de-facto used for a particular purpose without official IANA sanctioning. As examples, Skype uses 23399 and Steam uses several in the range 27000-27040.

Port numbers 49152-65535 are **ephemeral** ports (also called **dynamic** ports). These may be assigned and/or reassigned by the OS to any application.

Humans write IP address and port combinations with the ports in decimal after a colon. For example: 172.19.8.52:5060.

10.6 Transport Protocols

There are two ubiquitous transport layer protocols, User Datagram Protocol (UDP) and Transmission Control Protocol (TCP). There are several other transport protocols out there (for example SCTP and QUIC), that are typically built on the ideas found in one or other of UDP/TCP to provide either additional functionality or new compromises between cost and features. We'll cover UDP and TCP in depth in this section.

10.7 User Datagram Protocol (UDP) (RFC 768)

10.7.1 Abstract

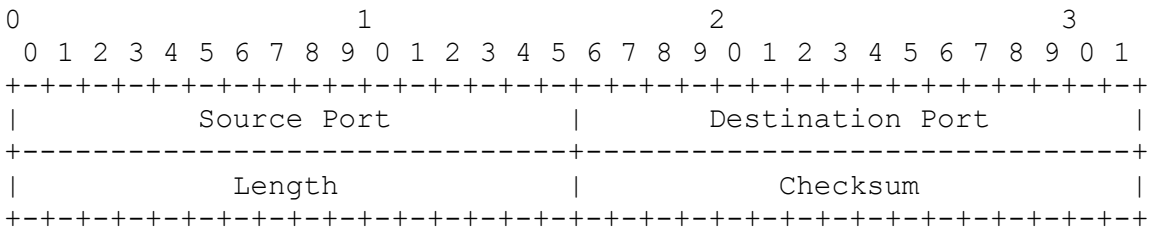
UDP provides a fast "datagram" service between endpoints with minimum possible overheads. It is a connectionless, unidirectional service where endpoints can efficiently send potentially large chunks of data (datagrams) to each other. The goal of UDP is fast transport, and so it provides minimal additional service on top of the IP layer (no error recovery, flow control, guaranteed delivery, etc.) in order to provide minimum overhead. All it provides over the raw IP below is the following:

- Source and Destination Port identification. This is to allow data to be delivered directly to the destination application, and to allow responses coming back from the destination to be correctly returned to the source.
- Datagram integrity verification. UDP provides a checksum that verifies the entire UDP datagram is consistent. You'll remember that the IPv4 IP header already contains a checksum, and so for UDP the checksum is optional and may be set to 0 to further reduce processing overheads. For IPv6 (which does not contain a checksum) the UDP checksum is mandatory.

UDP Joke

At this point in the course I would tell you my favourite UDP joke, but I wouldn't know if you got it.

10.7.2 UDP Header



The UDP header consists of the following fields:

- **Source Port** (2 bytes). The port of the source application.
- **Destination Port** (2 bytes). The port destination application.
- **Length** (2 bytes). The length in bytes of the UDP datagram (including the UDP header).
- **Checksum** (2 bytes). The data integrity checksum, as calculated below.

10.7.3 UDP Checksum calculation

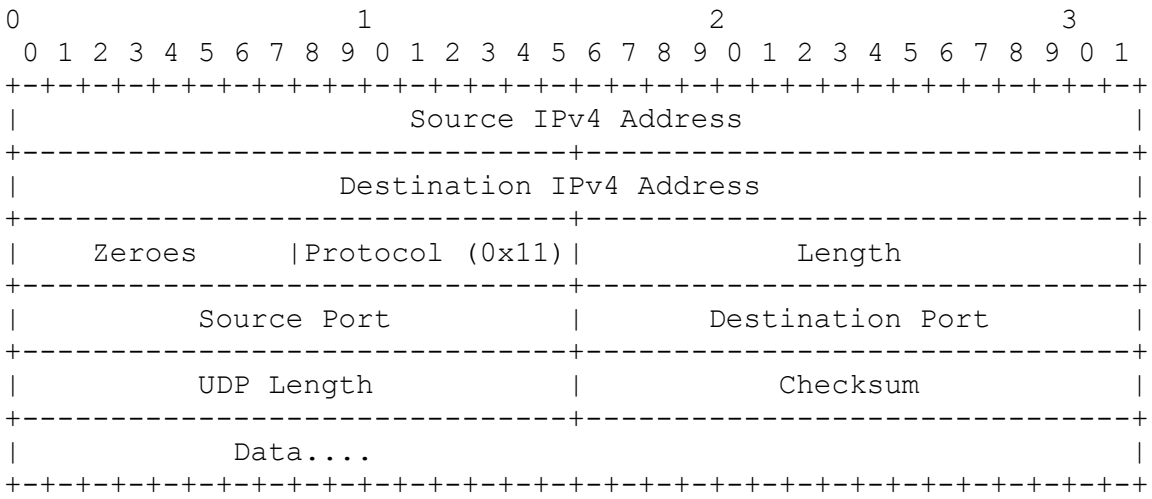
The UDP checksum is calculated over more than just the whole UDP datagram – it is additionally calculated over a pseudoheader that contains most of the IP header.

Remember that the UDP header is transport layer, with strict layering the transport layer doesn't need to know about the IP header that will be added – so this breaks layering.

The checksum calculation is the same as that already described for IPv4 in the section on IP networking. As a quick reminder, it is the 16-bit one's-complement sum of the one's-complement sum of all the 16-bit words in the pseudoheader and data. That means that the receiver can perform the one's complement sum of the entire header including checksum and check against 0xFFFF.

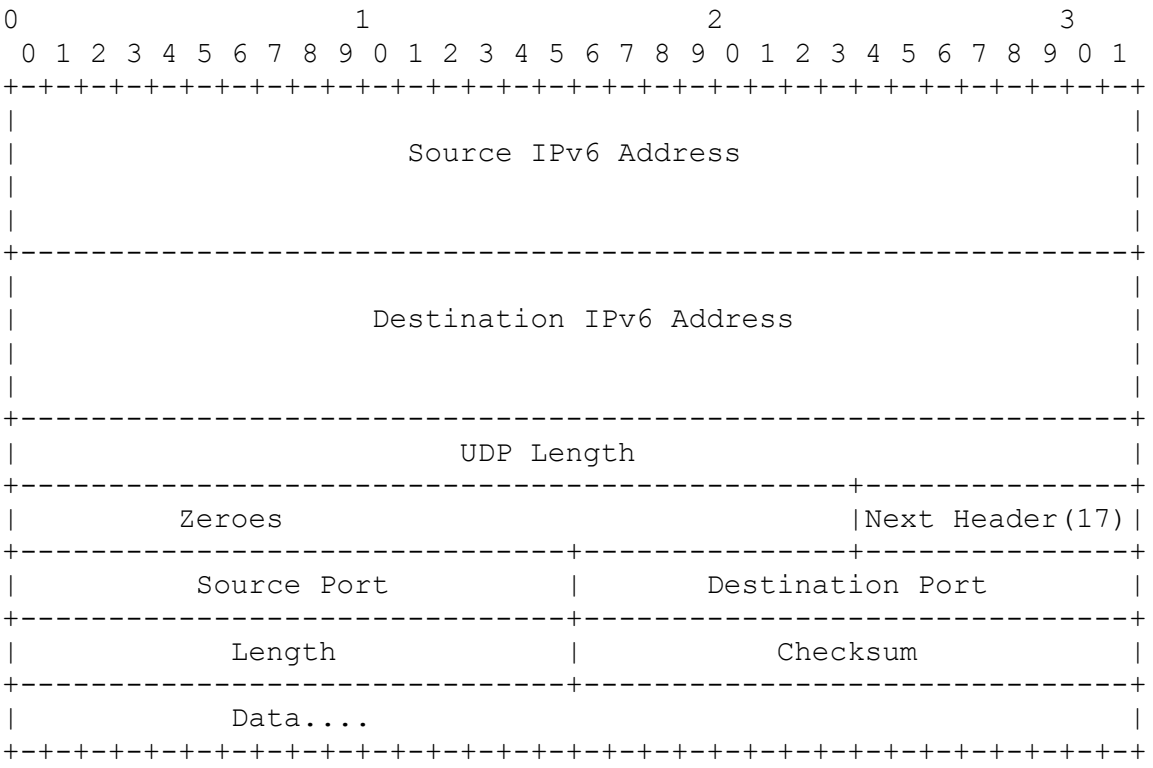
The IPv4 and IPv6 pseudoheaders used for checksum calculation are shown below:

IPv4 Pseudoheader



Note that there are some hard-coded values in this pseudoheader. The Protocol in the IPv4 pseudoheader is always 0x11 (UDP), and the offset to optional headers is instead assumed to be zeroes. This is because those are fields that may be sorted out by lower layers, whereas the rest of the fields in the pseudoheader are fields that the transport layer can reasonably know or calculate.

IPv6 Pseudoheader



Note again that there are parts of the pseudoheader (those that the transport layer is unlikely to be able to predict) that are assumed to be zero for the checksum calculation.

10.7.4 Example flow

10_UDP.pcap is an example flow of two UDP messages between two endpoints. (More precisely, it is an example flow of two Chargen protocol messages that flow over UDP over IPv4. We don't care in this course what the Chargen protocol is – it's just some data for the UDP datagrams.) You will see in this flow that both the IPv4 and UDP checksums have been calculated and included by the endpoints. Depending on your settings, Wireshark's checking of checksums may have been disabled - you can enable using the options under Edit->Preferences->Protocols->IPv4 or UDP. You could also try calculating the checksums yourself!

Bad Checksum

One of the checksums in 10_UDP.cap is wrong! What's going on? In order to speed up packet processing, many modern network interfaces have specialist checksum calculators that calculate and fill out the checksum just before the packet is sent on the wire. However, most packet capture tools will get a copy of the packet *before* this is done, meaning some of the checksums for *outgoing* packets may just be full of junk, as they've not actually been filled out yet. Wireshark disables checksum checking by default, mainly to prevent people being fooled by this false error.

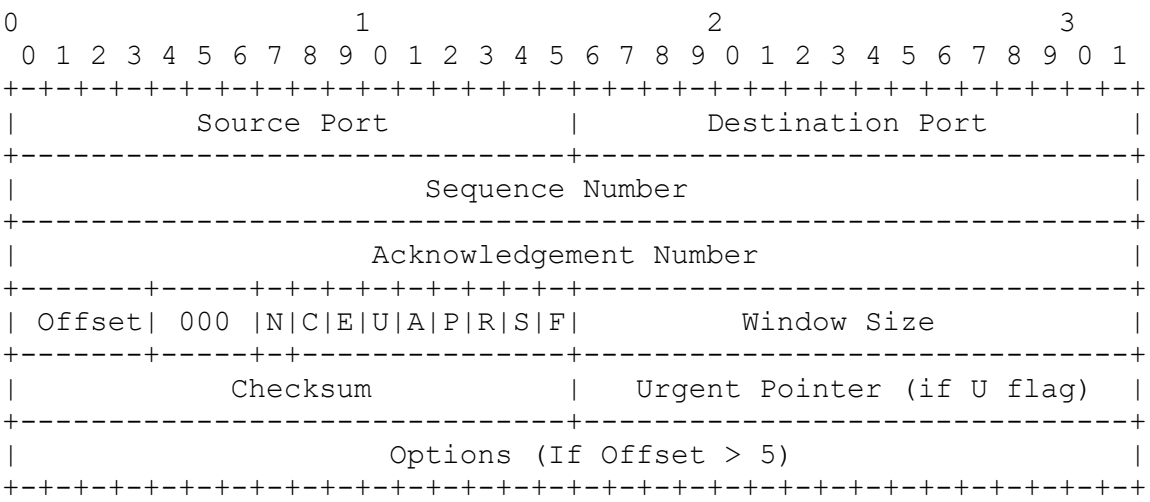
10.8 Transmission Control Protocol (TCP) (RFC 793)

10.8.1 Abstract

TCP provides all the additional services that the transport layer promises. It is connection oriented and allows two endpoints to establish a point to point bidirectional connection which is a flow limited, guaranteed delivery, guaranteed ordering byte stream.

TCP flows consist of two parts – a message exchange to set up or tear down a connection, and a message exchange to provide data transfer. TCP packets are called segments and each segment has a TCP header at the top of it. The TCP header is used in both setup/teardown and data transfer.

10.8.2 TCP Header



Flags

There are a set of 1-bit fields (collectively called the flags fields) in the middle of the header. For reference these are named here. They'll be covered as we work through the protocol below.

- N – Nonce
- C – Congestion Window Reduced
- E – Explicit Congestion Echo
- U – Urgent
- A – Acknowledge
- P – Push
- R – Reset
- S – Synchronise
- F – Finish

Header contents

Much of the header matches the UDP counterparts:

- **Source Port** (2 bytes). The source port – the same as UDP.
- **Destination Port** (2 bytes). The destination port – the same as UDP.
- **Offset** (1 byte). The offset from the start of the TCP header to the start of the data (measured in 32 bit words).
- **000** (3 bits). The first 3 bits after the offset are reserved for future use and are always 0.
- **Checksum** (2 bytes). As with UDP, the checksum runs over the IPv4/IPv6 pseudoheader and the entire TCP segment.

We'll unpack the rest of the fields (including the flags fields) as we work through how TCP sets up and tears down connections and handles sending data.

10.8.3 TCP Connection establishment and teardown

Establishment

TCP Session establishment happens as follows:

- One side listens on a known port. For example, the BGP Protocol listens for incoming TCP connection requests from BGP Peers on port 179.
- On initial receipt of a new connection message, the listening port spawns a connection from a new dynamic port, keeping the listening port open for further incoming connections.
- On the new set of ports, each side sends a **synchronise** request (SYN) and receives an **acknowledgement** response (ACK). This is actually a 3-way handshake, as the first side to respond with an ACK, piggy-backs its SYN in the same message – the message flow goes SYN (A->B), ACK+SYN (B->A), ACK (A->B).
    - If either side doesn't receive an ACK to its SYN, it retries on a timer, eventually giving up.
    - What exactly are the sides synchronising? As we'll see shortly, TCP labels each segment of data sent sequentially – tracked using the **sequence number** in the header – which wraps. Each initial SYN/ACK announces (and acknowledges) the initial sequence number that each side will use to label data.
- SYN failures (no ACK response received) are retried and eventually result in a failure to set up the connection.

Teardown

Both directions are independently closed. For each direction, one side sends a **finish** (FIN) – which is the very last message that it will send on the connection, and the other responds with an ACK. Note that because it's possible that the connection teardowns can overlap, then it's possible that no ACK will ever be received, so the endpoint that sends the **finish** also closes the connection on a timer.

Most OSs prevent reuse of TCP ports for ~2 mins after closing to ensure that any remaining in flight messages don't arrive in the new session.

TCP Header fields:

- The **S** flag is set to 1 to indicate a synchronise request, and 0 otherwise.
- The **A** flag is set to 1 to indicate an acknowledgement response and 0 otherwise.
- The **F** flag is set to 1 to indicate the finish segment and 0 otherwise.
- The **R** flag is the reset flag and used (set to 1) by TCP to indicate that something has gone wrong (for example, there is no process listening on the port a TCP SYN has been sent to) and this resets the connection.

10.8.4 TCP Data Transfer

The data stream in TCP is transferred in chunks called segments with a maximum size. Segments are filled with data passed by the application, and sent when they are full, or on a timer, or when pushed by the application. Segments are numbered with sequence numbers, so the receiver can correctly order the data, and each one is acknowledged (acked) so that the sender knows what has and has not been received. The sender resends unacknowledged segments on a timer until it receives an acknowledgement for that segment. The TCP checksum identifies corruption of segments and the receiver treats such corrupt packets as unreceived and does not ack them.

TCP Header fields:

- **Sequence Number** is the sequence number of the current segment.
    - The synchronise message opening the connection in each direction sets this field to the first sequence number the sender will use.
- **Acknowledgement Number** is the sequence number of the first unreceived segment, and acts as an acknowledgement of every segment up to (but not including) that number.
    - Note that this means that a receiver that has received segments 1,2,4,5,6,7, will send an acknowledgement number of 3, even though it has received more recent segments. As soon as it has received 3, it can "jump" to an acknowledgement number of 8.

10.8.5 Pushing Data

TCP batches up data and buffers it. On the sending end, it won't necessarily send data received from the application straight away, but may group it to be more efficient. Likewise the receiving TCP will not necessarily bother the application with everything received immediately, but may batch into more "useful" chunks. If an application wants to send data "now" (for example at the end of a file transfer), it can ask TCP to push the data.

TCP Header fields:

The **P** flag is set to 1 by the sending TCP to indicate to the receiving TCP that it should flush its buffers up to the application.

10.8.6 Urgent Data

Normally TCP delivers all data in order, buffering received data if previous segments haven't yet been received (even if pushed). An application can indicate that a chunk of data-stream is urgent (for example some kind of "abort" message), and this is delivered to the receiving application by the receiving TCP immediately on receipt – even if out of order.

TCP Header fields:

The **U** flag is set to 1 by the sending TCP to indicate that this segment contains urgent data.

The **Urgent Pointer** indicates the last urgent byte in the segment (in case there is more data in the segment that is not urgent).

10.8.7 Flow control – sliding window

When dealing with endpoints of different speeds, we need a way for a slower receiver to push back on a faster sender to prevent it becoming overloaded. Controlling the flow of data to prevent a receiver becoming overloaded is called **flow control.** TCP provides this by allowing the receiver to inform the sender how much data it is permitted to send.

In TCP, the receiver advertises to the sender the maximum number of un-acknowledged bytes the sender may have outstanding. This means that the sender is ultimately limited to the rate at which the receiver is acknowledging segments (with a variable buffer on top that the receiver is allowed to set). Note that the receiver can set this to a lower number than the amount of data currently unacked and the sender will stop sending data until it receives further acknowledgements. The receiver can also set this value to 0, meaning "ARGH! STOP!", which causes a sender to send no further data for a short time (it will restart sending on a timer).

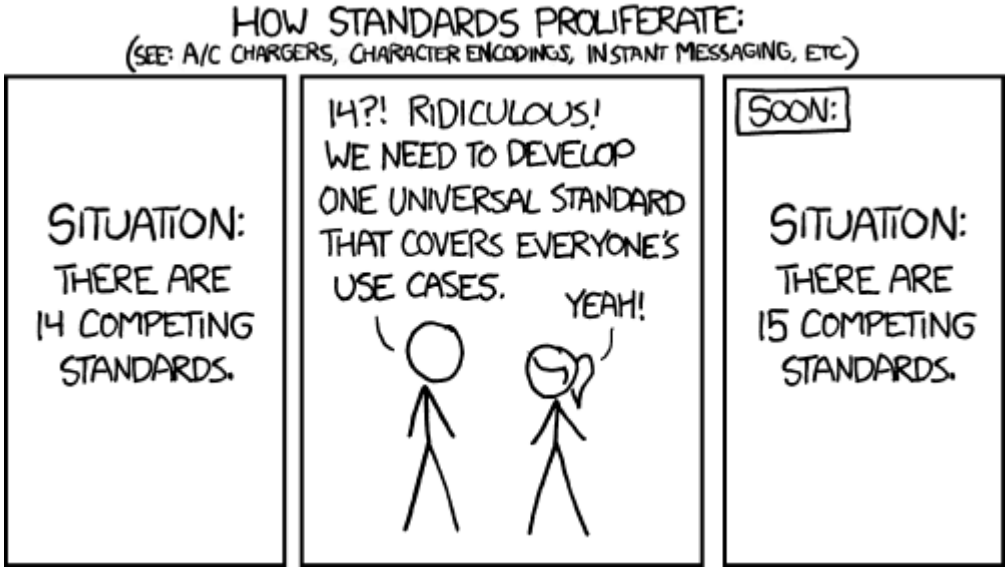
TCP Header fields:

The **Window Size** is set by the receiver to the number of un-acked bytes (not TCP segments!) that the receiver will permit.

As a side note, the windows size is 2 bytes long and so cannot by default advertise a window of more than 65k bytes. To allow much larger window sizes, there is a TCP option (defined in RFC 1323) that is sent on the initial acknowledgement message in a connection, which is simply a constant scale factor to multiply up the window size. This scale factor is maintained for the lifetime of the connection.

10.8.8 Congestion

Congestion occurs when a lower layer network infrastructure is overloaded – and **congestion control** is about rate limiting the data flow to avoid congestion in the network. TCP has "a lot" (20+ at last count) of different algorithms and methods to detect, deal with or avoid congestion – very much a case of the issue covered in XKCD 927 (Copyright Randal Munroe – used under creative commons CC BY-NC 2.5). We'll only cover a few key ones here.



Explicit Congestion Notification

TCP includes the ability to indicate congestion using "Explicit Congestion Notification" – using the last 3 flags in the TCP header (The **N**, **C** and **E** flags). Explicit Congestion Notification (RFC 3168) allows routers in the network to set bits to indicate they're spotting congestion. In TCP, then:

- The **E** (Explicit Congestion Echo) flag allows the receiver to indicate to the sender that it is receiving IP packets with ECN bits set.
- The **C** (Congestion Window Reduced) flag allows the sender to indicate that it has taken action on receipt of an **E** flag.
- The **N** flag allows a simple 1-bit NONCE (used to prevent routers or endpoints exploiting ECN to get unfair use of resources).

We won't explicitly cover this further in the course.

Slow Start

If we assume that the limit in data flow is the network and not the endpoints, then a TCP endpoint can use something similar to the sliding window covered earlier under flow control to self-limit the data that it sends.

TCP endpoints maintain this second internal sliding-window (the congestion window) independently of the receiver-driven sliding-window, and when sending, the endpoint uses the smallest of the two

– so if the limit between a particular sender and receiver *is* the receiver rather than the network between them, then flow control will kick in and the sender will be limited as described in the previous section.

A TCP endpoint determines the "best" window size that maximises use of the network without causing congestion using the following very simple algorithm:

- On a successful ack, data is getting through. Increase the window size.
- On a lost segment, data is not getting through. Halve the window size.

By default, the sender increases the window by 1 segments worth of data each successful ack, *but* linear incrementing is very slow at the start if we start with a very small window size. Instead, the sender increases the window exponentially, doubling the size of the window, until it hits its first missed segment.

Fast Retransmit

Halving the window size to safely recover from congestion is a reasonable thing to do – but when is a lost segment due to an overloaded network suffering ongoing congestion, and when is it just a network glitch that lost an individual segment?

By default TCP retransmits unacknowledged segments on a timer. However, if TCP detects duplicate ACKs for a particular sequence number, then rather than waiting for the timer, it could assume that the segment was lost and send a bonus retransmission straight away (but leave the retransmission timer running). This means that in the case when it was just a glitch and *not* general network congestion, the bonus retransmit will get through and we'll get an ack for that segment before the retransmission timer times out. In turn we can then assume that a retransmission timeout *does* indicate a loss of data caused by congestion and trigger the window-size-halving.

I'm not going to cover the explicit details or internals of exactly how a TCP sender updates its congestion window based on Slow Start and Fast Retransmit. TCP congestion is a hole down which we need to stop exploring at some point.

10.8.9 Options

The TCP header has options, and their presence is indicated by the **Offset** field being greater than 5 (* 32 bits) that makes up the TCP header. These allow for future extensibility. Each Option is a standard "TLV" (Type, Length, Value) triplet.

- Type (1 byte)
- Length (1 byte) (length of the entire TLV measured in bytes)
- Value (N-2 bytes where N is the value in the length field)

An example mentioned above: the TCP window scale option:

- Type (1 byte): 3
- Length (1 byte): 3
- Value (1 byte): A number between 1-255, indicating how many powers of 2 to multiply the window size by. (For example, a value of 6 means multiply the window size by $2^6 = 64$.)

10.8.10 TCP example flows

10_BGP_over_TCP.pcap alongside this lecture (actually the same as **8_BGP.pcap**) shows all of the process of setting up and tearing down a TCP connection, and also segments sent over that connection. You'll remember that BGP runs over TCP, so for the two BGP routers to talk to each other, they must set up a TCP connection.

- Packets 1-4 in the trace show teardown of an earlier TCP connection.

- Packets 5-7 show the 3 way handshake starting the new TCP connection (for the BGP messages).
- The data passed over the TCP connection shows up as BGP messages (which it is!) because Wireshark unpacks the BGP too.

It's worth working through the IP headers and the TCP headers (even for the BGP messages) to get your head into what TCP is doing at each point.

10_LDP_Session_over_TCP.pcap (actually the same as **9_LDP.pcap**) shows two LDP routers setting up an LDP session over TCP. Now that you understand TCP, you should be able to work out what is going on here too.

10_General_TCP_Streams.pcapng alongside this lecture shows a variety of TCP streams in a busier environment. This includes:

- Segments at the start that are acking other segments for which the original packet wasn't captured (which Wireshark helpfully flags up).
- TCP running over IPv4 and IPv6.
- Two TCP connections running between the same endpoints.