TCP

The *Transmission Control Protocol* (TCP) is the transport layer for a reliable, connection-oriented protocol

TCP

The *Transmission Control Protocol* (TCP) is the transport layer for a reliable, connection-oriented protocol

Often called “TCP/IP”

TCP

The *Transmission Control Protocol* (TCP) is the transport layer for a reliable, connection-oriented protocol

Often called "TCP/IP"

It is *hugely* more complicated than UDP as it must create a reliable transport from the unreliable IP it runs on

TCP

The *Transmission Control Protocol* (TCP) is the transport layer for a reliable, connection-oriented protocol

Often called “TCP/IP”

It is *hugely* more complicated than UDP as it must create a reliable transport from the unreliable IP it runs on

There is a lot of complication to deal with the error cases, such as packet loss and packet duplication

TCP

The *Transmission Control Protocol* (TCP) is the transport layer for a reliable, connection-oriented protocol

Often called “TCP/IP”

It is *hugely* more complicated than UDP as it must create a reliable transport from the unreliable IP it runs on

There is a lot of complication to deal with the error cases, such as packet loss and packet duplication

There is overhead in setting up (and taking down) the connection to manage these mechanisms

TCP

The *Transmission Control Protocol* (TCP) is the transport layer for a reliable, connection-oriented protocol

Often called “TCP/IP”

It is *hugely* more complicated than UDP as it must create a reliable transport from the unreliable IP it runs on

There is a lot of complication to deal with the error cases, such as packet loss and packet duplication

There is overhead in setting up (and taking down) the connection to manage these mechanisms

And more to complexity improve performance and flow control

TCP

The *Transmission Control Protocol* (TCP) is the transport layer for a reliable, connection-oriented protocol

Often called “TCP/IP”

It is *hugely* more complicated than UDP as it must create a reliable transport from the unreliable IP it runs on

There is a lot of complication to deal with the error cases, such as packet loss and packet duplication

There is overhead in setting up (and taking down) the connection to manage these mechanisms

And more to complexity improve performance and flow control

A lot of state about each connection needs to be stored by the OS

TCP

The basis of the reliability is the use of acknowledgement (ACK) packets for every packet sent

TCP

The basis of the reliability is the use of acknowledgement (ACK) packets for every packet sent

If host A sends host B a packet, B must send an ACK packet back to A to inform it of the safe arrival of the packet

TCP

The basis of the reliability is the use of acknowledgement (ACK) packets for every packet sent

If host A sends host B a packet, B must send an ACK packet back to A to inform it of the safe arrival of the packet

If A does not get an ACK, it resends the packet

TCP

The basis of the reliability is the use of acknowledgement (ACK) packets for every packet sent

If host A sends host B a packet, B must send an ACK packet back to A to inform it of the safe arrival of the packet

If A does not get an ACK, it resends the packet

But ACKs on their own do not solve all the problem

TCP

This is due to the *Two Armies Problem*: suppose two armies A and B wish to coordinate an attack on C

TCP

This is due to the *Two Armies Problem*: suppose two armies A and B wish to coordinate an attack on C

A sends a message to B: “attack at dawn”

TCP

This is due to the *Two Armies Problem*: suppose two armies A and B wish to coordinate an attack on C

A sends a message to B: “attack at dawn”

How does A know that B got the message? A cannot safely attack until it knows B is ready

TCP

This is due to the *Two Armies Problem*: suppose two armies A and B wish to coordinate an attack on C

A sends a message to B: “attack at dawn”

How does A know that B got the message? A cannot safely attack until it knows B is ready

So B sends an acknowledgement to A: “OK”

TCP

This is due to the *Two Armies Problem*: suppose two armies A and B wish to coordinate an attack on C

A sends a message to B: “attack at dawn”

How does A know that B got the message? A cannot safely attack until it knows B is ready

So B sends an acknowledgement to A: “OK”

But the ACK might be intercepted and A might not get the ACK

TCP

B can’t attack until it knows A got the ACK

TCP

B can’t attack until it knows A got the ACK

So A should send an ACK for the ACK back to B

TCP

B can’t attack until it knows A got the ACK

So A should send an ACK for the ACK back to B

But this might not get through…

TCP

B can’t attack until it knows A got the ACK

So A should send an ACK for the ACK back to B

But this might not get through…

For full reliability it looks like we need an infinite regress!

TCP

TCP avoids the Two Armies Problem by using timeouts and packet retransmissions

TCP

TCP avoids the Two Armies Problem by using timeouts and packet retransmissions

For every packet:

TCP

TCP avoids the Two Armies Problem by using timeouts and packet retransmissions

For every packet:

A starts a *retransmission timer* when it sends to B

TCP

TCP avoids the Two Armies Problem by using timeouts and packet retransmissions

For every packet:

A starts a *retransmission timer* when it sends to B

If the timer runs out before it gets an ACK, it resends the packet and restarts the timer

TCP

TCP avoids the Two Armies Problem by using timeouts and packet retransmissions

For every packet:

A starts a *retransmission timer* when it sends to B

If the timer runs out before it gets an ACK, it resends the packet and restarts the timer

Repeat until A gets an ACK (or A gives up)

TCP

Problems to solve include:

TCP

Problems to solve include:

- how long to wait before a resend? This might be a slow but otherwise reliable link and resending will just clog the system with extra duplicate packets

TCP

Problems to solve include:

- how long to wait before a resend? This might be a slow but otherwise reliable link and resending will just clog the system with extra duplicate packets
- how many times to resend before giving up? It might be the destination has gone away entirely (perhaps disconnected or crashed)

TCP

Problems to solve include:

- how long to wait before a resend? This might be a slow but otherwise reliable link and resending will just clog the system with extra duplicate packets
- how many times to resend before giving up? It might be the destination has gone away entirely (perhaps disconnected or crashed)
- how long B should wait before sending the ACK? You can *piggyback* an ACK on an ordinary data packet, so it may be better for B to wait until some data is ready to be returned rather than sending an otherwise empty ACK. This saves on packets sent

TCP

Problems to solve include:

- how long to wait before a resend? This might be a slow but otherwise reliable link and resending will just clog the system with extra duplicate packets
- how many times to resend before giving up? It might be the destination has gone away entirely (perhaps disconnected or crashed)
- how long B should wait before sending the ACK? You can *piggyback* an ACK on an ordinary data packet, so it may be better for B to wait until some data is ready to be returned rather than sending an otherwise empty ACK. This saves on packets sent
- IP datagrams can arrive out of order, so we need some way to recognise which ACK goes with which packet

TCP

Other problems TCP also needs to address include:

TCP

Other problems TCP also needs to address include:

- how to maintain order in the data? IP datagrams can arrive out of order, so we need some way of reassembling the original data stream in the correct order

TCP

Other problems TCP also needs to address include:

- how to maintain order in the data? IP datagrams can arrive out of order, so we need some way of reassembling the original data stream in the correct order
- how to manage duplicates? Resends can produce duplicate packets (if the original was not actually lost) so we need some way to recognise and discard extra copies

TCP

Other problems TCP also needs to address include:

- how to maintain order in the data? IP datagrams can arrive out of order, so we need some way of reassembling the original data stream in the correct order
- how to manage duplicates? Resends can produce duplicate packets (if the original was not actually lost) so we need some way to recognise and discard extra copies
- Flow control: how to increase the rate of sending packets when things are going well, and decrease the rate when they are not

TCP

TCP packets are often called *segments*

TCP

TCP packets are often called *segments*

(Reminder: “segment”, “packet”, “datagram”, “frame” all mean pretty much the same thing, just in different layers)

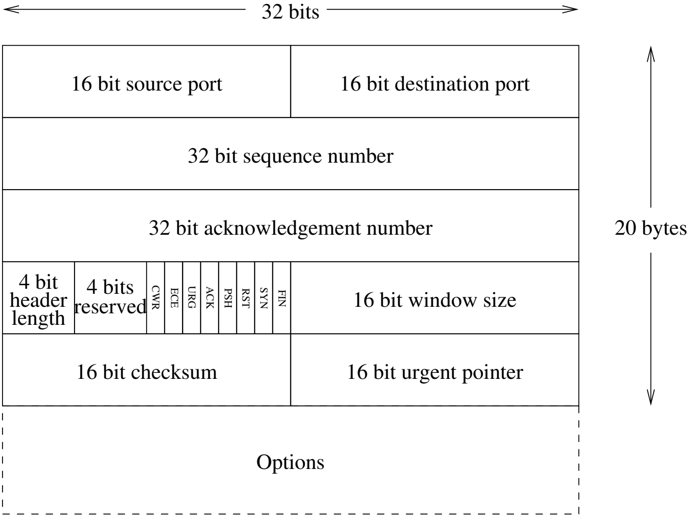
TCP

TCP packets are often called *segments*

(Reminder: “segment”, “packet”, “datagram”, “frame” all mean pretty much the same thing, just in different layers)

A TCP header is complicated as it must address many complex issues

TCP



32 bits

16 bit source port

16 bit destination port

32 bit sequence number

32 bit acknowledgement number

4 bit header length

4 bits reserved

CWR

ECE

URG

ACK

PSH

RST

SYN

FIN

16 bit window size

16 bit checksum

16 bit urgent pointer

Options

20 bytes

TCP header

TCP

- Ports: identical to UDP (on purpose: actually UDP copied TCP)

TCP

- Ports: identical to UDP (on purpose: actually UDP copied TCP)
- Two 32 bit values: *sequence* and *acknowledgement*

TCP

Sequence numbers

These numbers are the heart of TCP’s reliability

TCP

Sequence numbers

These numbers are the heart of TCP’s reliability

Every **byte** in a TCP connection is numbered

TCP

Sequence numbers

These numbers are the heart of TCP’s reliability

Every **byte** in a TCP connection is numbered

The 32 bit sequence number starts at some random value and increases by 1 for each byte sent

TCP

Sequence numbers

These numbers are the heart of TCP’s reliability

Every **byte** in a TCP connection is numbered

The 32 bit sequence number starts at some random value and increases by 1 for each byte sent

So if a segment contains 10 bytes of data, the sequence number on the next segment sent will be 10 greater

TCP

Sequence numbers

The sequence number in the header is the number of the first byte of data in the segment

TCP

Sequence numbers

The sequence number in the header is the number of the first byte of data in the segment

The destination acknowledges those bytes it has received by filling in the ACK field with the appropriate byte number and setting the ACK flag

TCP

Sequence numbers

The reverse connection from destination to source has its own sequence number as TCP is fully duplex

TCP

Sequence numbers

The reverse connection from destination to source has its own sequence number as TCP is fully duplex

Everything we say here is true for data travelling in the reverse direction: the reverse traffic has its own independent sequence numbers and flow control

TCP

Sequence numbers

Note that a destination might not immediately get the whole segment that was sent due to fragmentation in the IP layer

TCP

Sequence numbers

Note that a destination might not immediately get the whole segment that was sent due to fragmentation in the IP layer

IP must wait for all the fragments and reconstruct the segment before it can pass it on to TCP and then TCP can send the ACK

TCP

Sequence numbers

Note that a destination might not immediately get the whole segment that was sent due to fragmentation in the IP layer

IP must wait for all the fragments and reconstruct the segment before it can pass it on to TCP and then TCP can send the ACK

And this can play havoc with TCP's timers

TCP

Sequence numbers

Note that a destination might not immediately get the whole segment that was sent due to fragmentation in the IP layer

IP must wait for all the fragments and reconstruct the segment before it can pass it on to TCP and then TCP can send the ACK

And this can play havoc with TCP's timers

Another reason to avoid fragmentation

TCP

Sequence numbers

The returning ACK field contains the sequence number of the next byte the destination expects to receive, e.g., if the sequence number is 20001 and 14 bytes are received it returns 20015 in the ACK field

TCP

Sequence numbers

The returning ACK field contains the sequence number of the next byte the destination expects to receive, e.g., if the sequence number is 20001 and 14 bytes are received it returns 20015 in the ACK field

ACKs can be *piggybacked* on normal returning data packets, they don’t need to be separate packets

TCP

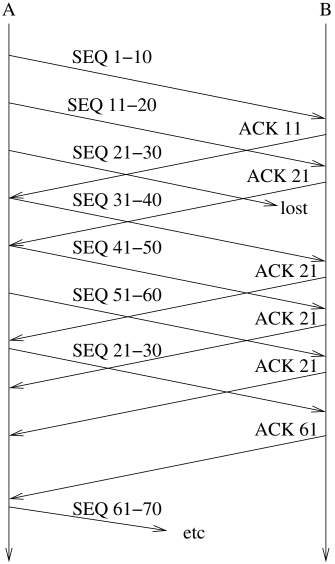
Sequence numbers

The returning ACK field contains the sequence number of the next byte the destination expects to receive, e.g., if the sequence number is 20001 and 14 bytes are received it returns 20015 in the ACK field

ACKs can be *piggybacked* on normal returning data packets, they don’t need to be separate packets

This helps reduce the amount of network traffic

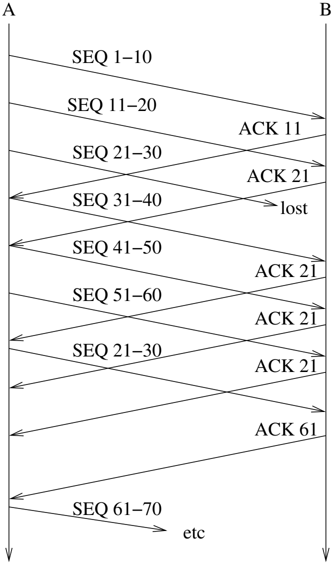
TCP



ACKing lost segments

A is sending 10 byte segments to B, and B is ACKing them;

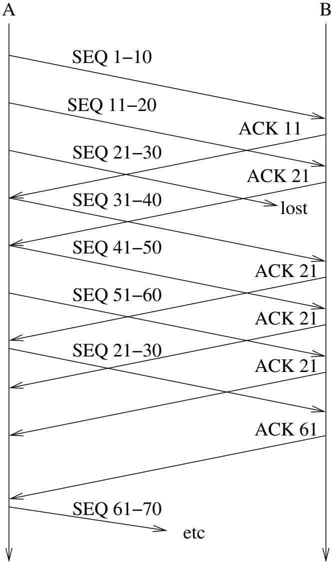
TCP



ACKing lost segments

The segment containing bytes 21-30 is lost;

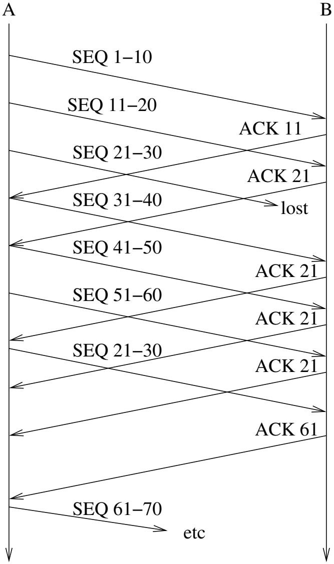
TCP



ACKing lost segments

When B next gets a segment it still ACKS with 21: that’s the byte it wants next;

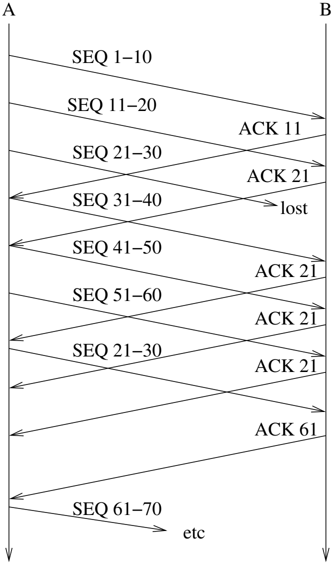
TCP



ACKing lost segments

While the ACK travels back to A, A is still sending new data;

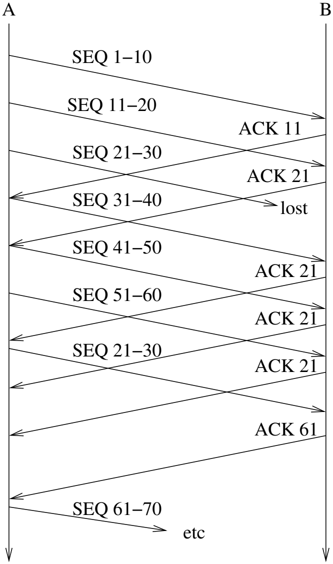
TCP



ACKing lost segments

Eventually A gets duplicate ACKs from B: this is a sign of a problem;

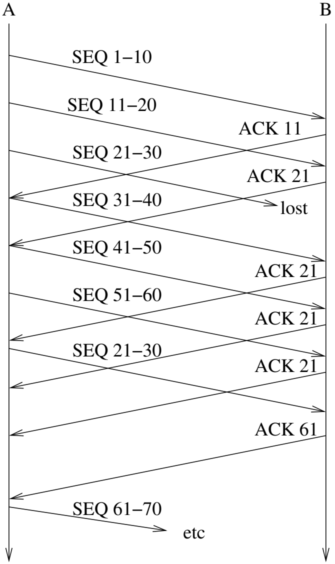
TCP



ACKing lost segments

A resends bytes 21-30;

TCP



ACKing lost segments

When B gets these bytes it can ACK all the way up to 60

TCP

Sequence numbers

In fact this diagram is not realistic: it is over-simplified to fit on the slide

TCP

Sequence numbers

In fact this diagram is not realistic: it is over-simplified to fit on the slide

TCP specifies that A should continue until it get gets *three duplicate ACKs* (i.e., four ACKs with the same sequence number, not piggybacked on data and not changing the advertised window) before resending

TCP

Sequence numbers

In fact this diagram is not realistic: it is over-simplified to fit on the slide

TCP specifies that A should continue until it get gets *three duplicate ACKs* (i.e., four ACKs with the same sequence number, not piggybacked on data and not changing the advertised window) before resending

This is to avoid triggering resends too easily, e.g., it might be just a case of A's packets being slightly reordered in transit, where a resend is not actually required (remember TCP runs on top of the unreliable IP)

TCP

Sequence numbers

In fact this diagram is not realistic: it is over-simplified to fit on the slide

TCP specifies that A should continue until it get gets *three duplicate ACKs* (i.e., four ACKs with the same sequence number, not piggybacked on data and not changing the advertised window) before resending

This is to avoid triggering resends too easily, e.g., it might be just a case of A's packets being slightly reordered in transit, where a resend is not actually required (remember TCP runs on top of the unreliable IP)

Exercise When might we receive many ACKs with the same sequence number, but nothing is in error?

TCP

Sequence numbers

The sequence number wraps around after $2^{32} - 1 = 4294967295$ bytes

TCP

Sequence numbers

The sequence number wraps around after $2^{32} - 1 = 4294967295$ bytes

This is under 10 seconds for a 10Gb/s Ethernet

TCP

Sequence numbers

The sequence number wraps around after $2^{32} - 1 = 4294967295$ bytes

This is under 10 seconds for a 10Gb/s Ethernet

Additional mechanisms to extend the count have had to be devised in the light of modern fast networks

TCP

Sequence numbers

The sequence number wraps around after $2^{32} - 1 = 4294967295$ bytes

This is under 10 seconds for a 10Gb/s Ethernet

Additional mechanisms to extend the count have had to be devised in the light of modern fast networks

Exercise E.g., using the TCP header timestamp option. Read about PAWS

TCP

Sequence numbers

The sequence number wraps around after $2^{32} - 1 = 4294967295$ bytes

This is under 10 seconds for a 10Gb/s Ethernet

Additional mechanisms to extend the count have had to be devised in the light of modern fast networks

Exercise E.g., using the TCP header timestamp option. Read about PAWS

Much more on SEQ and ACKing later, but note that sequence numbers solve the segment ordering problem, too

TCP

Back to the TCP header

TCP

Back to the TCP header

- 4 bits header length: measured in 32 bit words: the header can have options, so is of variable length

TCP

Back to the TCP header

- 4 bits header length: measured in 32 bit words: the header can have options, so is of variable length

So maximum is 60 bytes. Minimum is the fixed part: 20 bytes

TCP

Back to the TCP header

- 4 bits header length: measured in 32 bit words: the header can have options, so is of variable length

So maximum is 60 bytes. Minimum is the fixed part: 20 bytes

- Many flags performing various functions

TCP

Most of these will be described in more detail as we go along:

- URG: urgent data
- ACK: the acknowledgement field is active
- PSH: push this data to the application as fast as possible
- RST: reset (break) the connection
- SYN: synchronise a new connection
- FIN: finish a connection
- ECE: congestion notification
- CWR: congestion window reduced
- 4 reserved bits, set to 0