TCP Timers

Next: TCP has several timers. We have seen

- 2MSL
- Delayed ACK

These are just the start!

TCP Timers

Retransmission Timer

We now consider the timer that determines when to resend in the absence of an ACK: a *retransmission timeout* (RTO)

TCP Timers

Retransmission Timer

We now consider the timer that determines when to resend in the absence of an ACK: a *retransmission timeout* (RTO)

- too short a time is wasteful on slow but otherwise reliable networks
- too long a time is poor for the data rate

TCP Timers

Retransmission Timer

We now consider the timer that determines when to resend in the absence of an ACK: a *retransmission timeout* (RTO)

- too short a time is wasteful on slow but otherwise reliable networks
- too long a time is poor for the data rate

And we want a dynamic behaviour that adapts to changing conditions rather than a simple fixed timeout

TCP Timers

Retransmission Timer

If the network slows down (e.g., heavy other traffic causes less bandwidth for your packets) the timeout should increase

TCP Timers

Retransmission Timer

If the network slows down (e.g., heavy other traffic causes less bandwidth for your packets) the timeout should increase

If the network speeds up (e.g., other traffic reduces) the timeout should decrease

TCP Timers

Retransmission Timer

If the network slows down (e.g., heavy other traffic causes less bandwidth for your packets) the timeout should increase

If the network speeds up (e.g., other traffic reduces) the timeout should decrease

Jacobson gave an easy algorithm: keep a variable, the *round trip time* RTT for each connection

TCP Timers

Retransmission Timer

If the network slows down (e.g., heavy other traffic causes less bandwidth for your packets) the timeout should increase

If the network speeds up (e.g., other traffic reduces) the timeout should decrease

Jacobson gave an easy algorithm: keep a variable, the *round trip time* RTT for each connection

RTT is the best current estimate for the time of a segment going out and the ACK returning

TCP Timers

Retransmission Timer

If the network slows down (e.g., heavy other traffic causes less bandwidth for your packets) the timeout should increase

If the network speeds up (e.g., other traffic reduces) the timeout should decrease

Jacobson gave an easy algorithm: keep a variable, the *round trip time* RTT for each connection

RTT is the best current estimate for the time of a segment going out and the ACK returning

If we haven't received an ACK in approximately this time, deem it lost

TCP Timers

Retransmission Timer

In more detail: when a segment is sent, its timer starts

TCP Timers

Retransmission Timer

In more detail: when a segment is sent, its timer starts

If the ACK returns before the timeout, TCP looks at the actual round trip time M and updates RTT using

$$RTT = \alpha RTT + (1 - \alpha) M$$

TCP Timers

Retransmission Timer

In more detail: when a segment is sent, its timer starts

If the ACK returns before the timeout, TCP looks at the actual round trip time M and updates RTT using

$$RTT = \alpha RTT + (1 - \alpha)M$$

α is a smoothing factor, usually 7/8 for easy arithmetic

TCP Timers

Retransmission Timer

Thus RTT increases or decreases smoothly as conditions change and doesn't get too upset by the occasional straggler that is unusually late (or early)

TCP Timers

Retransmission Timer

Thus RTT increases or decreases smoothly as conditions change and doesn’t get too upset by the occasional straggler that is unusually late (or early)

Next, we need to determine a timeout interval given RTT

TCP Timers

Retransmission Timer

Thus RTT increases or decreases smoothly as conditions change and doesn't get too upset by the occasional straggler that is unusually late (or early)

Next, we need to determine a timeout interval given RTT

This should take the standard deviation of the RTT into account: if the measured RTTs have a large deviation it makes sense to have a larger timeout

TCP Timers

Retransmission Timer

Thus RTT increases or decreases smoothly as conditions change and doesn't get too upset by the occasional straggler that is unusually late (or early)

Next, we need to determine a timeout interval given RTT

This should take the standard deviation of the RTT into account: if the measured RTTs have a large deviation it makes sense to have a larger timeout

True standard deviations are tricky to compute quickly (square roots), so Jacobson suggested using the *mean deviation*

TCP Timers

Retransmission Timer

Mean deviation:

$$D = \beta D + (1 - \beta)|RTT - M|$$

TCP Timers

Retransmission Timer

Mean deviation:

$$D = \beta D + (1 - \beta)|RTT - M|$$

D is close to the standard deviation and is much easier to calculate quickly

TCP Timers

Retransmission Timer

Mean deviation:

$$D = \beta D + (1 - \beta)|RTT - M|$$

D is close to the standard deviation and is much easier to calculate quickly

A typical value for β is 3/4

TCP Timers

Retransmission Timer

The timeout value is set to

$$T = RTT + 4D$$

TCP Timers

Retransmission Timer

The timeout value is set to

$$T = RTT + 4D$$

The 4 and the values for α, β were found to be good in practice

TCP Timers

Retransmission Timer

The timeout value is set to

$$T = RTT + 4D$$

The 4 and the values for α, β were found to be good in practice

When sending a segment (or, in practice, a burst of segments) set the timer to expire after time T

TCP Timers

Retransmission Timer

What if the timer expires before the ACK is received?

TCP Timers

Retransmission Timer

What if the timer expires before the ACK is received?

- we resend the segment, of course

TCP Timers

Retransmission Timer

What if the timer expires before the ACK is received?

- we resend the segment, of course
- but we also need to update RTT somehow

TCP Timers

Retransmission Timer

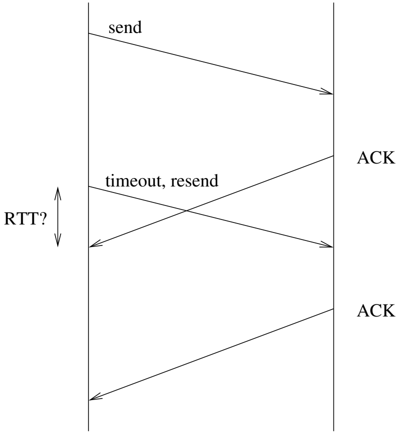
What if the timer expires before the ACK is received?

- we resend the segment, of course
- but we also need to update RTT somehow

But we can't use RTT of the resent segment as we might get the somewhat delayed ACK of the original segment, not of the resent segment

TCP Timers

Retransmission Timer



Retransmission Ambiguity

This is the *retransmission ambiguity problem*

TCP Timers

Retransmission Timer

The measured RTT would be much too small

TCP Timers

Retransmission Timer

The measured RTT would be much too small

Karn's algorithm is to double the timeout T on each failure, but do not adjust RTT

TCP Timers

Retransmission Timer

The measured RTT would be much too small

Karn's algorithm is to double the timeout T on each failure, but do not adjust RTT

When segments start getting through normal RTT updates continue and RTT quickly reaches the appropriate value

TCP Timers

Retransmission Timer

The measured RTT would be much too small

Karn's algorithm is to double the timeout T on each failure, but do not adjust RTT

When segments start getting through normal RTT updates continue and RTT quickly reaches the appropriate value

This doubling is called *exponential backoff*

TCP Timers

Retransmission Timer

The measured RTT would be much too small

Karn's algorithm is to double the timeout T on each failure, but do not adjust RTT

When segments start getting through normal RTT updates continue and RTT quickly reaches the appropriate value

This doubling is called *exponential backoff*

Alternatively, as is common these days, we have the option header timestamp and this solves the retransmission ambiguity directly

TCP Timers

Persist Timer

The next timer in TCP is the *persist timer*, sometimes called the *persistence timer*

TCP Timers

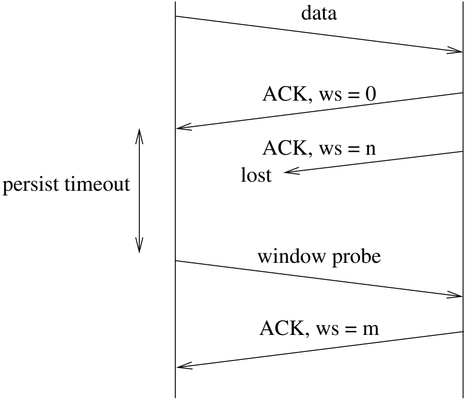
Persist Timer

The next timer in TCP is the *persist timer*, sometimes called the *persistence timer*

Its role is to prevent deadlock through the loss of window update segments

TCP Timers

Persist Timer

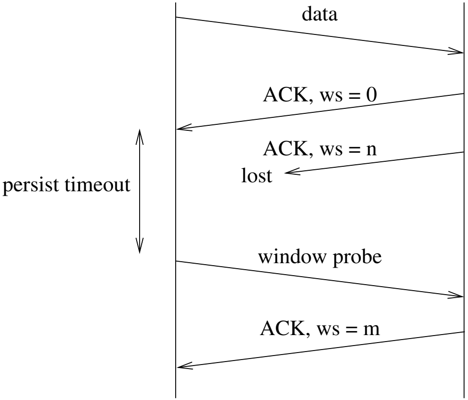


Persist timer

A sends to B;

TCP Timers

Persist Timer

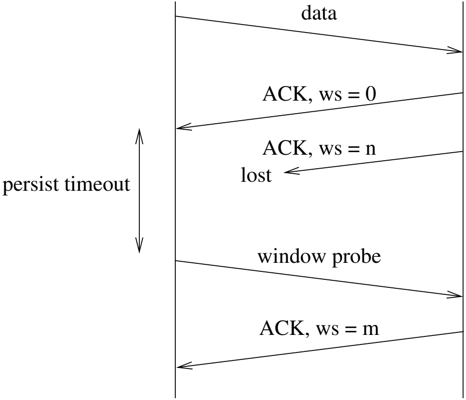


Persist timer

B replies with an ACK and a window size of 0;

TCP Timers

Persist Timer

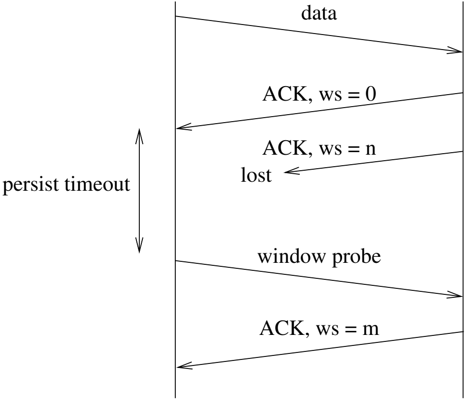


Persist timer

A gets the ACK and holds off sending to B;

TCP Timers

Persist Timer

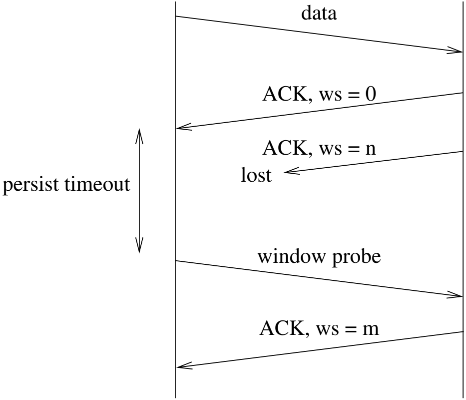


Persist timer

B frees up some buffer space and sends a window update to A;

TCP Timers

Persist Timer

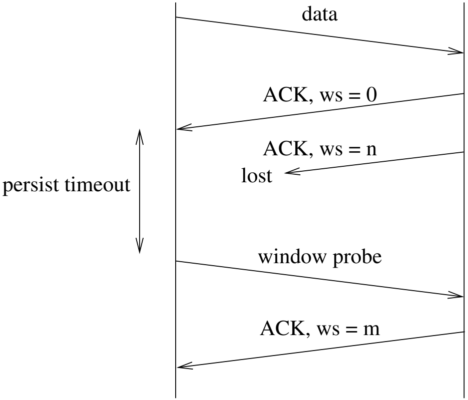


Persist timer

This is lost;

TCP Timers

Persist Timer

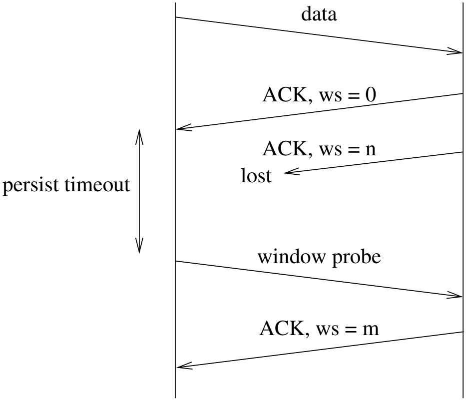


Persist timer

Now A is waiting for the window update from B and B is waiting for more data from A: deadlock;

TCP Timers

Persist Timer

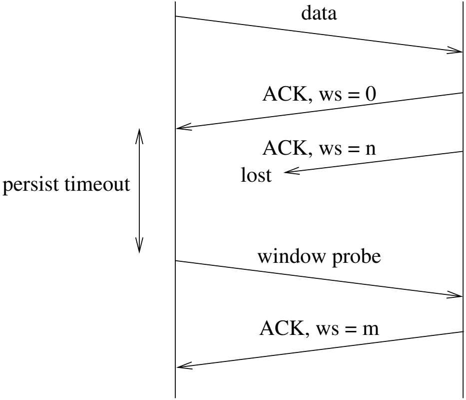


Persist timer

To prevent this, A starts the persist timer when it gets the 0 window from B;

TCP Timers

Persist Timer

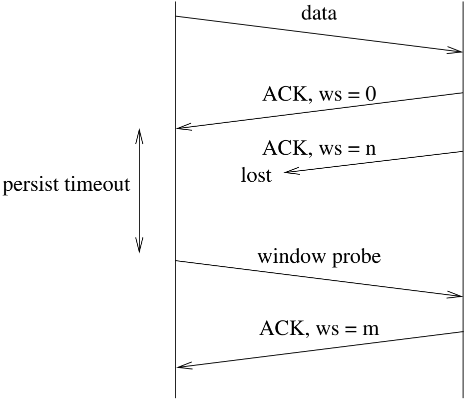


Persist timer

If the timer expires, A prods B by sending a 1 byte segment: a *window probe*;

TCP Timers

Persist Timer

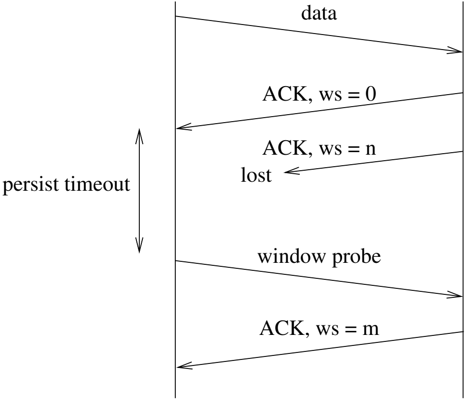


Persist timer

If B gets this, the ACK will contain B’s current window size;

TCP Timers

Persist Timer



Persist timer

If the window is still 0, A resets the timer and tries again later

TCP Timers

Persist Timer

The persist timer starts with something like 1.5 sec, doubling with each probe and is rounded up or down to lie within 5 to 60 seconds

TCP Timers

Persist Timer

The persist timer starts with something like 1.5 sec, doubling with each probe and is rounded up or down to lie within 5 to 60 seconds

So the timeouts are 5, 5, 6, 12, 24, 48, 60, 60, 60, ...

TCP Timers

Persist Timer

The persist timer starts with something like 1.5 sec, doubling with each probe and is rounded up or down to lie within 5 to 60 seconds

So the timeouts are 5, 5, 6, 12, 24, 48, 60, 60, 60, ...

The persist mechanism never gives up, sending window probes until either the window opens, or the connection closes

TCP Timers

Persist Timer

The persist timer starts with something like 1.5 sec, doubling with each probe and is rounded up or down to lie within 5 to 60 seconds

So the timeouts are 5, 5, 6, 12, 24, 48, 60, 60, 60, ...

The persist mechanism never gives up, sending window probes until either the window opens, or the connection closes

The persist timer is unset when a non-zero window is received

TCP Timers

Keepalive Timer

Yet another timer in TCP is the *keepalive*

TCP Timers

Keepalive Timer

Yet another timer in TCP is the *keepalive*

This one is an optional part of the TCP/IP standard, and some implementations do not have it as it is occasionally regarded as controversial

TCP Timers

Keepalive Timer

Yet another timer in TCP is the *keepalive*

This one is an optional part of the TCP/IP standard, and some implementations do not have it as it is occasionally regarded as controversial

When a TCP connection is idle no packets flow between source and destination

TCP Timers

Keepalive Timer

Yet another timer in TCP is the *keepalive*

This one is an optional part of the TCP/IP standard, and some implementations do not have it as it is occasionally regarded as controversial

When a TCP connection is idle no packets flow between source and destination

So part of the path could break and be restored and the connection is none the wiser

TCP Timers

Keepalive Timer

Yet another timer in TCP is the *keepalive*

This one is an optional part of the TCP/IP standard, and some implementations do not have it as it is occasionally regarded as controversial

When a TCP connection is idle no packets flow between source and destination

So part of the path could break and be restored and the connection is none the wiser

This gives us a bit of resilience against flaky networks

TCP Timers

Keepalive Timer

On the other hand, sometimes a server wants to know if a client is still alive: each client TCP connection uses some resources in the server (buffers, timers, etc.)

TCP Timers

Keepalive Timer

On the other hand, sometimes a server wants to know if a client is still alive: each client TCP connection uses some resources in the server (buffers, timers, etc.)

If the client has crashed these resources could better be used elsewhere

TCP Timers

Keepalive Timer

On the other hand, sometimes a server wants to know if a client is still alive: each client TCP connection uses some resources in the server (buffers, timers, etc.)

If the client has crashed these resources could better be used elsewhere

To do this the server sets a keepalive timer when the connection goes idle

TCP Timers

Keepalive Timer

On the other hand, sometimes a server wants to know if a client is still alive: each client TCP connection uses some resources in the server (buffers, timers, etc.)

If the client has crashed these resources could better be used elsewhere

To do this the server sets a keepalive timer when the connection goes idle

A typical value is 2 hours

TCP Timers

Keepalive Timer

When the timer expires, the server can send a *keepalive probe*

TCP Timers

Keepalive Timer

When the timer expires, the server can send a *keepalive probe*

This is simply an empty segment (i.e., no data)

TCP Timers

Keepalive Timer

When the timer expires, the server can send a *keepalive probe*

This is simply an empty segment (i.e., no data)

If the server gets an ACK, everything is OK

TCP Timers

Keepalive Timer

When the timer expires, the server can send a *keepalive probe*

This is simply an empty segment (i.e., no data)

If the server gets an ACK, everything is OK

If not, the server might conclude the client is no longer active

TCP Timers

Keepalive Timer

There are four cases

TCP Timers

Keepalive Timer

There are four cases

1. the client is up and running: the keepalive probe is ACKed and everybody is happy. The keepalive timer is reset to 2 hours

TCP Timers

Keepalive Timer

There are four cases

1. the client is up and running: the keepalive probe is ACKed and everybody is happy. The keepalive timer is reset to 2 hours
2. the client has crashed or is otherwise not responding to TCP: the server gets no ACK and resends after 75 seconds. After 10 probes, 75 seconds apart, if there is no response, the server terminates the connection with “connection timed out” sent to the server application

TCP Timers

Keepalive Timer

3. the client has crashed and rebooted. The client gets the probe and responds with a RST. The server gets the RST and terminates the connection with “connection reset by peer” sent to the application

TCP Timers

Keepalive Timer

3. the client has crashed and rebooted. The client gets the probe and responds with a RST. The server gets the RST and terminates the connection with “connection reset by peer” sent to the application
4. the client is up and running, but is unreachable, e.g., broken routing. This is indistinguishable from case 2, so the same events ensue

TCP Timers

Keepalive Timer

There are several reasons not to use keepalive

TCP Timers

Keepalive Timer

There are several reasons not to use keepalive

- they can cause a generally good connection to be closed because of a temporary failure of a router

TCP Timers

Keepalive Timer

There are several reasons not to use keepalive

- they can cause a generally good connection to be closed because of a temporary failure of a router
- they use bandwidth

TCP Timers

Keepalive Timer

There are several reasons not to use keepalive

- they can cause a generally good connection to be closed because of a temporary failure of a router
- they use bandwidth
- some network operators charge per packet

TCP Timers

Keepalive Timer

The latter two are not particularly good arguments as the cost is just a couple of packets every 2 hours

TCP Timers

Keepalive Timer

The latter two are not particularly good arguments as the cost is just a couple of packets every 2 hours

It is usually possible to disable keepalive in the application: some people think that keepalive should not be in the TCP layer, but should be handled by the application layer (i.e., the non-existent session layer)

TCP Strategies

Many other strategies to improve throughput have been proposed

TCP Strategies

Many other strategies to improve throughput have been proposed

Some have been widely adopted

TCP Strategies

Many other strategies to improve throughput have been proposed

Some have been widely adopted

Exercise Read about the problems of *long fat pipes*

TCP Strategies

Many other strategies to improve throughput have been proposed

Some have been widely adopted

Exercise Read about the problems of *long fat pipes*

Exercise Read about Protect Against Wrapped Sequence numbers (PAWS), Selective Acknowledgement (SACK)

TCP Extensions

Exercise Multipath TCP (MPTCP) has been suggested both for extra performance, failover and for mobile hosts that roam between, say, cellular and Wi-Fi (used in iOS7). It layers one MPTCP connection over one or more TCP connections, e.g., using both the cellular and Wi-Fi links simultaneously for one MPTCP connection

Exercise And potential alternatives to TCP. Read about TCP for Transactions (TTCP), Stream Control Transmission Protocol (SCTP), Datagram Congestion Control Protocol (DCCP)

TCP Alternatives

QUIC (originally “quick UDP Internet connection”, now just a name, not an acronym) is a Google-originated alternative to TCP (RFC9000)

TCP Alternatives

QUIC (originally “quick UDP Internet connection”, now just a name, not an acronym) is a Google-originated alternative to TCP (RFC9000)

Originally designed as a transport layer for HTTP/3 (the next version of HTTP), QUIC can be used as a general transport protocol

TCP Alternatives

QUIC (originally “quick UDP Internet connection”, now just a name, not an acronym) is a Google-originated alternative to TCP (RFC9000)

Originally designed as a transport layer for HTTP/3 (the next version of HTTP), QUIC can be used as a general transport protocol

It is reliable, connection oriented, has congestion control, is encrypted and authenticated and is transmitted within UDP datagrams (port 443, mostly)

TCP Alternatives

The last is important as routers have a tendency to mess with (or drop) packets if they don't recognise the protocol

TCP Alternatives

The last is important as routers have a tendency to mess with (or drop) packets if they don't recognise the protocol

There have been several new protocols in the past that have failed to gain popular use as routers would not recognise them

TCP Alternatives

The last is important as routers have a tendency to mess with (or drop) packets if they don't recognise the protocol

There have been several new protocols in the past that have failed to gain popular use as routers would not recognise them

In fact, the QUIC header is encrypted (inside the UDP packet) to prevent routers inspecting or trying to modify it

TCP Alternatives

Note: QUIC uses UDP purely to avoid router problems: it would be better to layer directly over IP, but history won't let us do that

TCP Alternatives

Note: QUIC uses UDP purely to avoid router problems: it would be better to layer directly over IP, but history won't let us do that

QUIC is *not* a lightweight protocol: it is as heavyweight as TCP+TLS

TCP Alternatives

Note: QUIC uses UDP purely to avoid router problems: it would be better to layer directly over IP, but history won't let us do that

QUIC is *not* a lightweight protocol: it is as heavyweight as TCP+TLS

It is "quick" in the sense of "fast", not "simple"

TCP Alternatives

Support for QUIC is growing in OSs and applications, for example the Chrome browser uses QUIC whenever possible to fetch Web pages

TCP Alternatives

Support for QUIC is growing in OSs and applications, for example the Chrome browser uses QUIC whenever possible to fetch Web pages

It has a 3 way opening handshake, like TCP, but this handshake also negotiates encryption

TCP Alternatives

Support for QUIC is growing in OSs and applications, for example the Chrome browser uses QUIC whenever possible to fetch Web pages

It has a 3 way opening handshake, like TCP, but this handshake also negotiates encryption

This saves time over the current schemes that open TCP and then establishes encryption (see TLS, later)

TCP Alternatives

Multiple data streams are multiplexed over a single connection, again saving time over TCP that would need to start up a connection for each stream

TCP Alternatives

Multiple data streams are multiplexed over a single connection, again saving time over TCP that would need to start up a connection for each stream

For example, a Web page might fetch dozens of items (text, images, JavaScript, . . .) from the same server

TCP Alternatives

Multiple data streams are multiplexed over a single connection, again saving time over TCP that would need to start up a connection for each stream

For example, a Web page might fetch dozens of items (text, images, JavaScript, ...) from the same server

These could all be sent within a *single* QUIC connection

TCP Alternatives

Current browsers do try to multiplex multiple streams over a single TCP connection, but this causes problems as an error in one stream causes TCP's error mechanisms to kick in, affecting *all streams* in the connection, even if the other streams had no error in themselves

TCP Alternatives

Current browsers do try to multiplex multiple streams over a single TCP connection, but this causes problems as an error in one stream causes TCP's error mechanisms to kick in, affecting *all streams* in the connection, even if the other streams had no error in themselves

QUIC does this multiplexing more efficiently, never stopping a good stream within a connection

TCP Alternatives

Current browsers do try to multiplex multiple streams over a single TCP connection, but this causes problems as an error in one stream causes TCP's error mechanisms to kick in, affecting *all streams* in the connection, even if the other streams had no error in themselves

QUIC does this multiplexing more efficiently, never stopping a good stream within a connection

QUIC manages errors at the stream level, not the connection level

TCP Alternatives

And:

- more sophisticated ACK mechanisms
- connection migration, e.g., WiFi to cellular
- sophisticated flow control (still under development)
- and lots of other stuff building on the knowledge gained since TCP was first invented

TCP Alternatives

QUIC is growing, but it will be a long time before it replaces TCP (lots of code to rewrite!)

TCP Alternatives

QUIC is growing, but it will be a long time before it replaces TCP (lots of code to rewrite!)

And TCP with TLS has had decades of tuning, so QUIC has a lot of work to do to catch up

TCP Alternatives

QUIC is growing, but it will be a long time before it replaces TCP (lots of code to rewrite!)

And TCP with TLS has had decades of tuning, so QUIC has a lot of work to do to catch up

Exercise Read about how QUIC reduces connection overheads and about the *head-of-line blocking* problem

Exercise Read about SPDY, the predecessor to QUIC, and its relationship to HTTP/2

Exercise Read about the *middlebox* (router) problem and why it means that new protocols will have a hard time on the Internet

UDP Alternatives

Exercise And don't forget UDP: UDPLite, RUDP, UDT, etc.

TCP

TCP is a huge success: from 1200 bits/sec telephone lines to gigabit networks and beyond it has turned out to be massively flexible and scalable

TCP

TCP is a huge success: from 1200 bits/sec telephone lines to gigabit networks and beyond it has turned out to be massively flexible and scalable

It took a lot of work, though!

TCP

Here is a small part of the output from `ss -io` (socket statistics) on a Linux machine:

```
tcp   ESTAB 0 0 172.16.2.1:34956  34.117.14.220:https
timer:(keepalive,31sec,0)
ts sack cubic wscale:7,7 rto:220 rtt:18.341/0.5 ato:40 mss:1368
pmtu:1420 rcvmss:647 advmss:1368 cwnd:2 ssthresh:7
bytes_sent:7179 bytes_retrans:240 bytes_acked:6939
bytes_received:6747 segs_out:515 segs_in:508 data_segs_out:198
data_segs_in:188 send 1.19Mbps lastsnd:28652 lastrcv:29228
lastack:28632 pacing_rate 2.39Mbps delivery_rate 634kbps
delivered:191 app_limited busy:32268ms retrans:0/8
rcv_space:13800 rcv_ssthresh:64156 minrtt:17.318
```