Lecture 6: Moving Transport Protocols to the Application Layer

Amy Babay

University of Pittsburgh School of Computing and Information
Department of Informatics and Networked Systems
Department of Computer Science

Revisiting Reliable Data Transport

- Remember the questions we asked a couple weeks ago:
  - Are there any drawbacks to requiring completely reliable, in-order packet delivery?
  - Any types of applications this might cause problems for?

Challenges for a completely reliable, ordered service

- What happens when a packet is lost?
  - **All** following packets are blocked from being delivered until that packet is recovered (head of line blocking)
- Good fit for certain applications (e.g. file transfer, remote login)

- When might this be a problem?
  - When applications have **real-time constraints** and would prefer to drop the packet (e.g. interactive video, audio, gaming)
  - When the application doesn't need **a single total order** on all packets
    - e.g. Web browsing: each webpage typically consists of multiple distinct files (base html file, images, embedded videos other elements)

Supporting new applications (or improving performance of old ones)

- What can we do for these applications?
- One option: just use UDP
- But, that doesn’t give us what we want either…
  – Best effort isn’t good enough!

Supporting new applications (or improving performance of old ones)

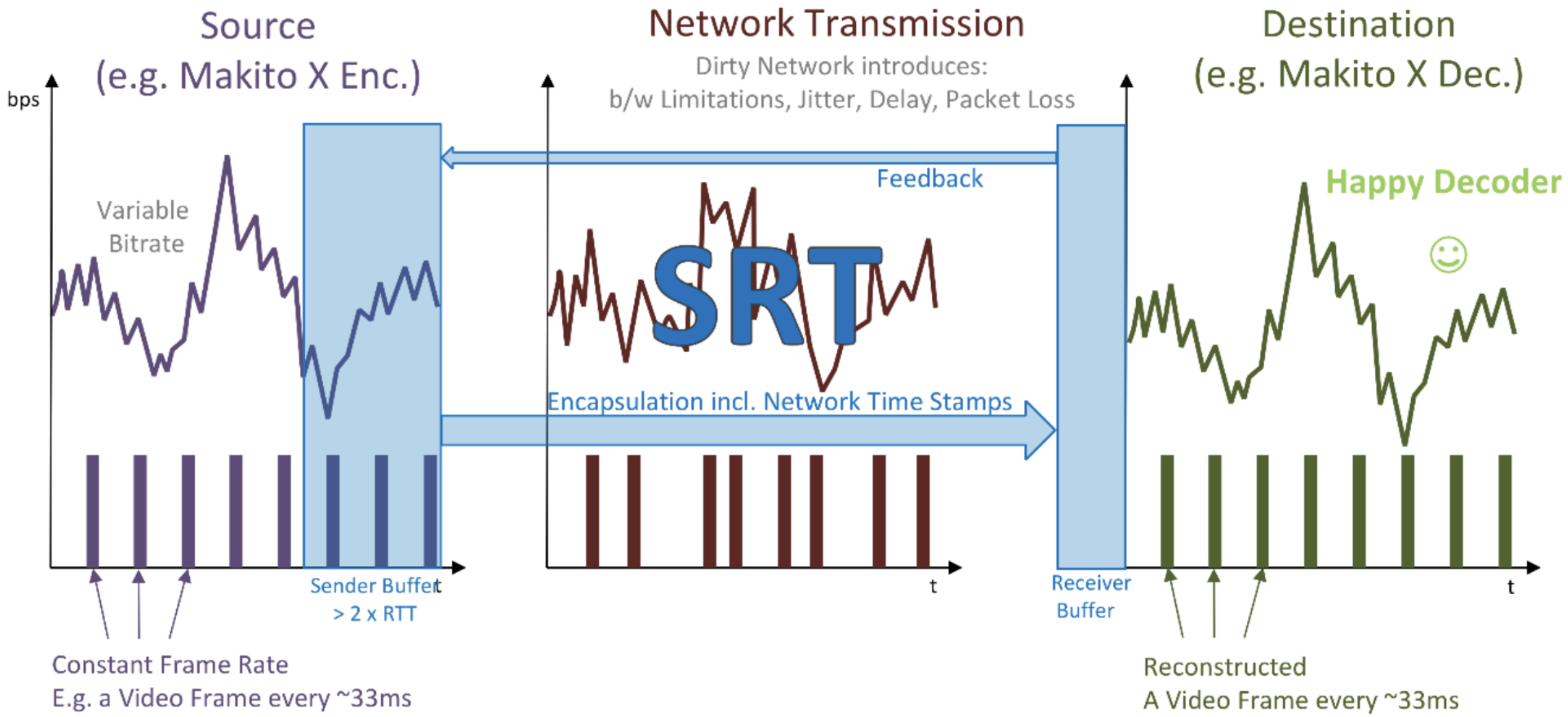
- We want new protocols to match application needs
  - Are there transport layer protocols other than TCP and UDP?
    - Yes! but, often not supported in all OSes, and therefore not widely used
    - Current trend is toward implementing **new transport layer functionality in userspace / at application layer**
  - Why? Making transport layer changes is hard!
    - Breaks others' assumptions…Firewalls block unfamiliar traffic, NAT relies on knowing header structure to rewrite IP address/ports. Timescale for standardization + adoption > 10 years
    - Requiring OS updates (to change TCP implementation in the kernel) -> slow adoption

SRT: Secure Reliable Transport

- Overall goal: provide high quality of experience for real-time video
  - Specifically targets ***video contribution*** - e.g. interview, coverage in the field that needs to be transported back to a TV studio at high quality for further distribution to end users
    - Traditional approach: satellite links – slow and expensive
  - Note that this is different from what you see as an end user watching video from Youtube, Facebook, etc
    - Those are mainly using adaptive bitrate protocols that run over HTTP (over TCP). Works well with existing CDN architectures

SRT: Secure Reliable Transport

- Overall goal: provide high quality of experience for real-time video
  - Low latency, low jitter, high reliability (few dropped packets)
  - Output stream pattern should **match** input stream pattern



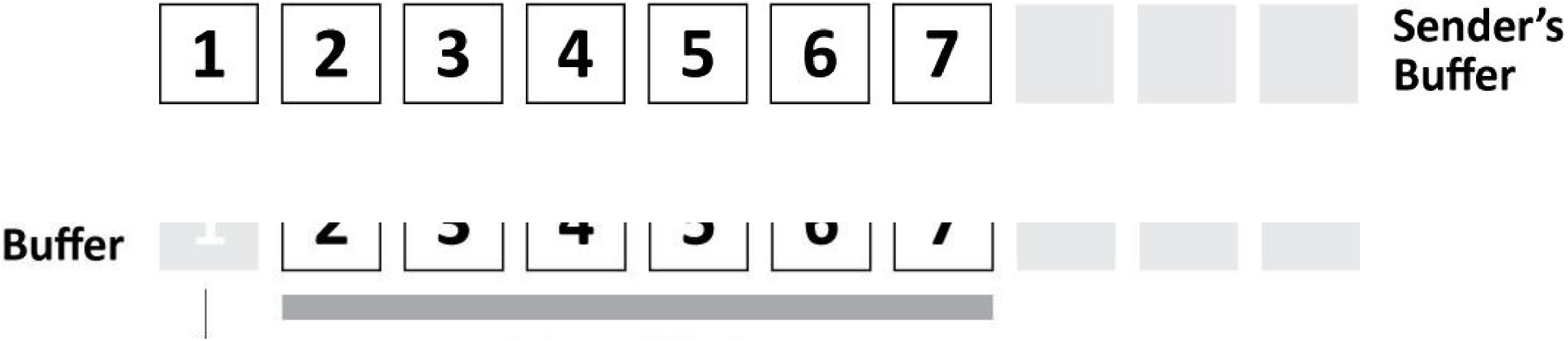
https://github.com/Haivision/srt/blob/master/docs/misc/why-srt-was-created.md

SRT: Secure Reliable Transport

- **Custom transport protocol, implemented in userspace on top of UDP**
- **Why not try to improve TCP?**
  - Reliable semantics are a bad fit for real-time video applications
    - Prefer to drop some packets vs. wait for retransmissions
    - Congestion control is problematic for fixed bit rate video
- **Why isn't UDP already good enough?**
  - *Perfect* reliability isn't a good fit, but we still want to **try** to recover lost packets (if we can do it fast enough)
  - We want to maintain a consistent delivery pattern that matches input

SRT: Key Features

- Packet-based sliding window ARQ protocol that uses cumulative ACKs and explicit NACKs (*should sound familiar!*)



So, what's new?

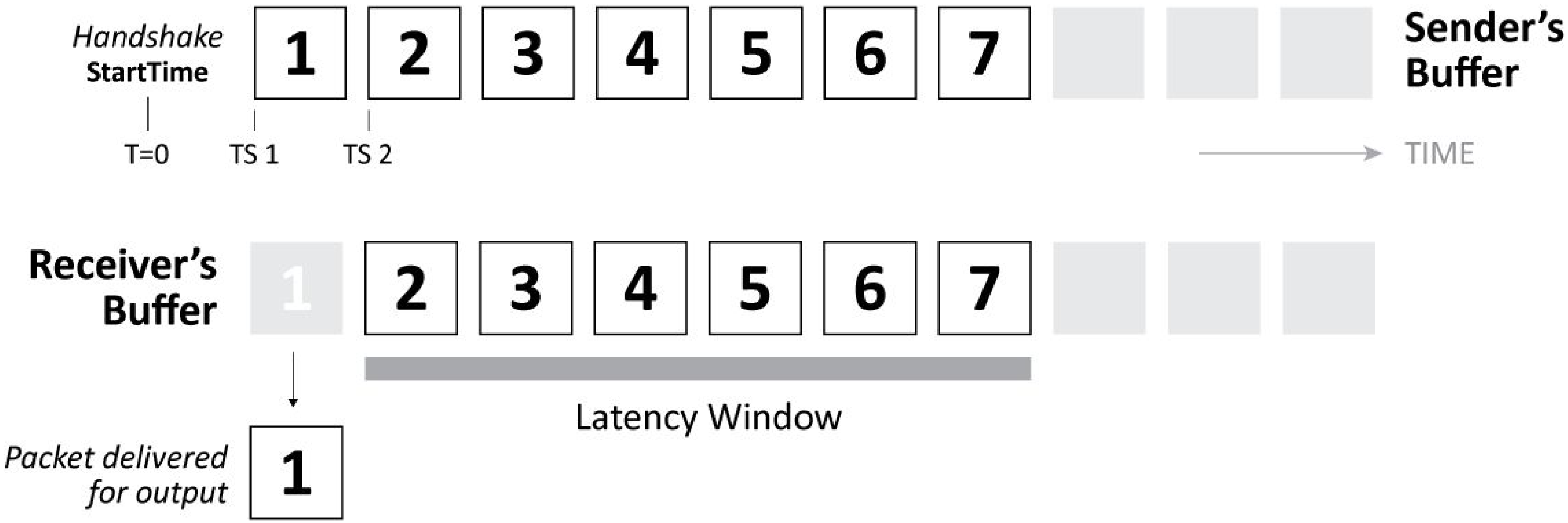
SRT: Sacrificing Reliability & Reproducing Input Timing

- **In the protocols you designed for project 1, when does your receiver decide to “deliver” a packet (write its data to file) from its buffer?**

- Is this a good fit for the application requirements we discussed for real-time video?
- What would you want instead?

SRT: Sacrificing Reliability & Reproducing Input Timing

- **Latency window** is used to determine when to deliver (or give up on) a packet in the window

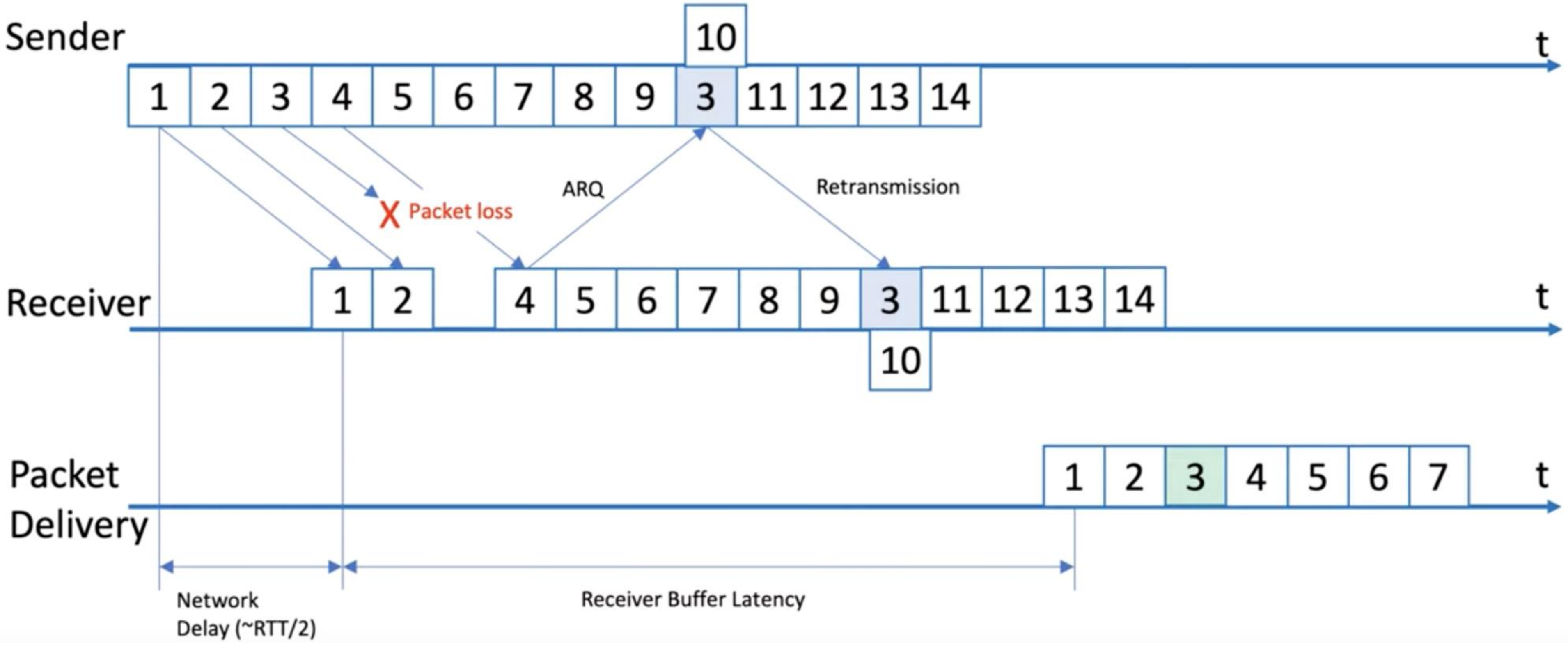


In more detail…

- Each packet gets a **timestamp** from the sender (TS1, TS2, …)
- The **latency window** is a value in milliseconds that says how long to buffer the packet
- At time TS+latency_window:
  - Receiver delivers the packet
  - Sender can remove it from its buffer (even if it hasn't been ACKed)

In more detail…

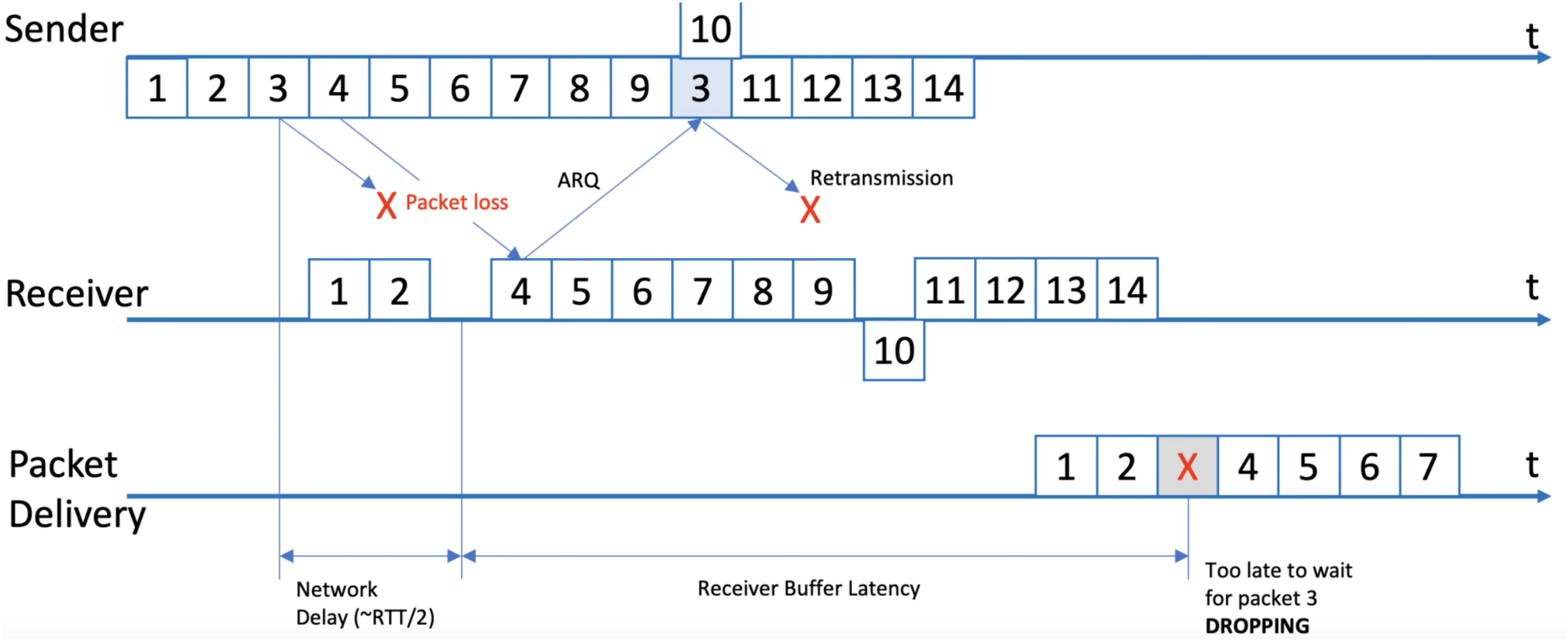
- Latency window gives some time to recover requested packets…



https://www.youtube.com/watch?v=VrE3dJej5IE

In more detail…

- But, if the time to deliver a packet arrives, the receiver delivers it, *even if some prior packets are still missing*



https://www.youtube.com/watch?v=VrE3dJej5IE

A complication

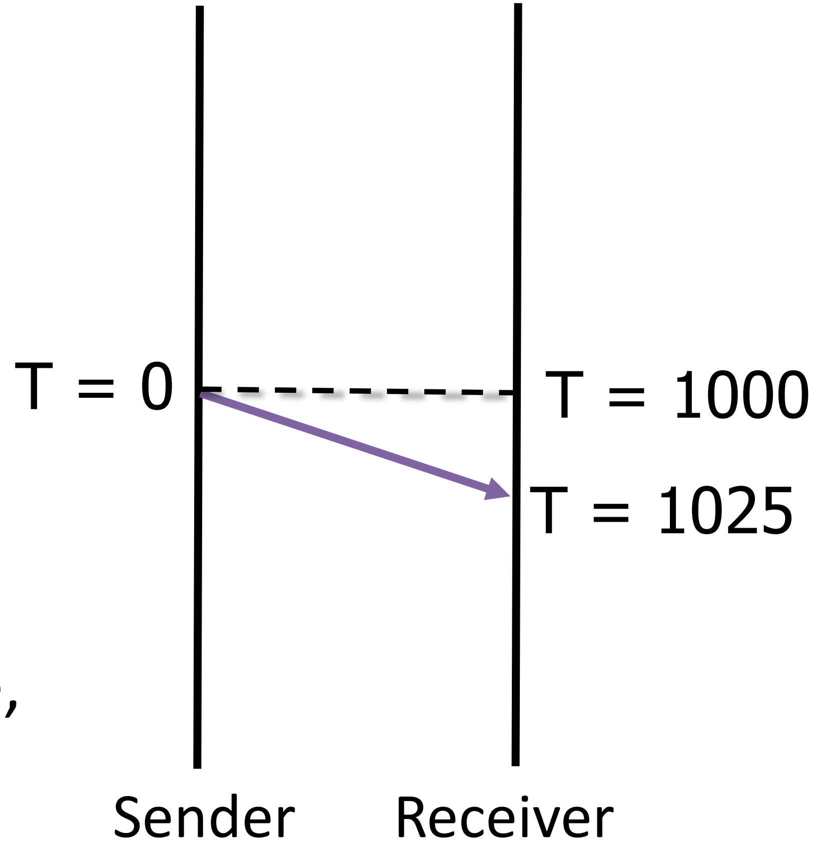
- Do you see any problem in implementing this latency window?
- Any assumptions we're making?

- Classic distributed systems problem:
  – What happens if our sender and receiver clocks aren't synchronized?!

Clock synchronization issues

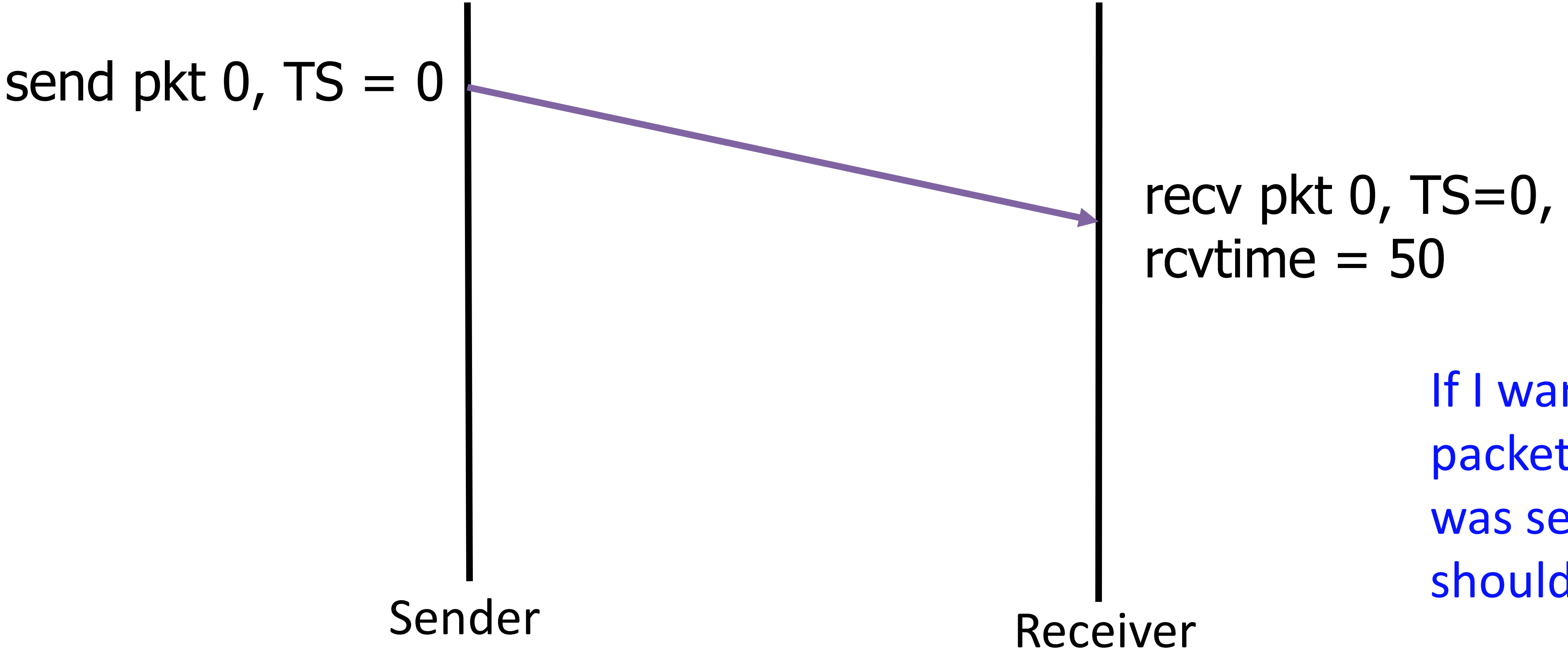
- Say our sender clock is 1 second (1000 ms) *behind* our receiver clock
- RTT between sender and receiver is 50ms
- Latency window is 500 ms

- What happens?
  - Sender sends packet with TS=0
  - Delivery time **should** be TS+latency_window = 500
  - Receiver gets it 25 ms (RTT/2) after it was sent. To the receiver, this looks like time 1025, packet is late, should just drop it.



Clock Skew Compensation

- Goal: map time on sender's clock (pkt timestamp) to time on receiver's clock



If I want to deliver a packet 500ms after it was sent, how long should I buffer it?

Clock Skew Compensation

- rcvtime – sendTS = 50
- I want to deliver the packet 500ms after it was sent (i.e. time 500 on sender’s clock)
- When do I deliver it?

- If clocks are synchronized, this is easy:
  - Target delivery time = 500, so wait 450ms, then deliver
- If I know the sender’s clock is 10ms behind mine, this is also easy:
  - Target delivery time = 500+10, so wait 460ms
  - But I don’t know this!
- **Observation: rcvtime – sendTS = oneway_delay + clock_diff**

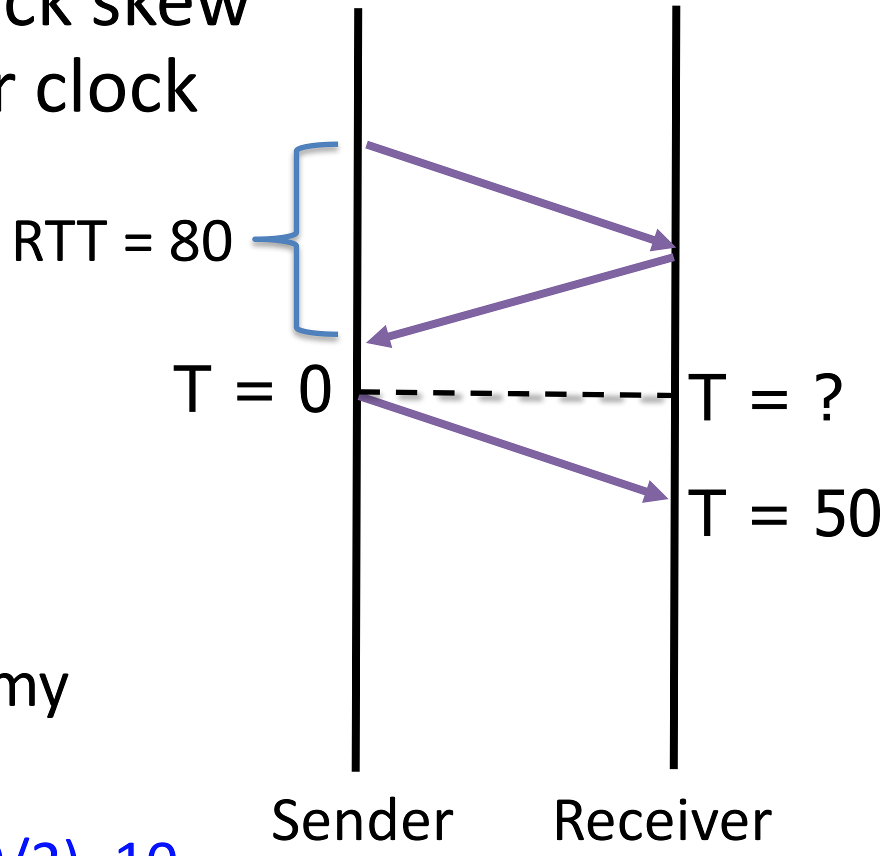
Clock Skew Compensation

- *If I know the normal one-way delay* between sender and receiver, I can estimate the clock skew to translate sender timestamps to receiver clock
- But, I don't know the one-way delay!
- But I can measure the roundtrip…

rcvtime – sendTS = RTT/2 + clock_diff

clock_diff = rcvtime – sendTS - RTT/2

then, I can use clock diff to translate sendTS to my (receiver's) local clock



clock_diff = 50-0-(80/2)=10

Clock Skew Compensation

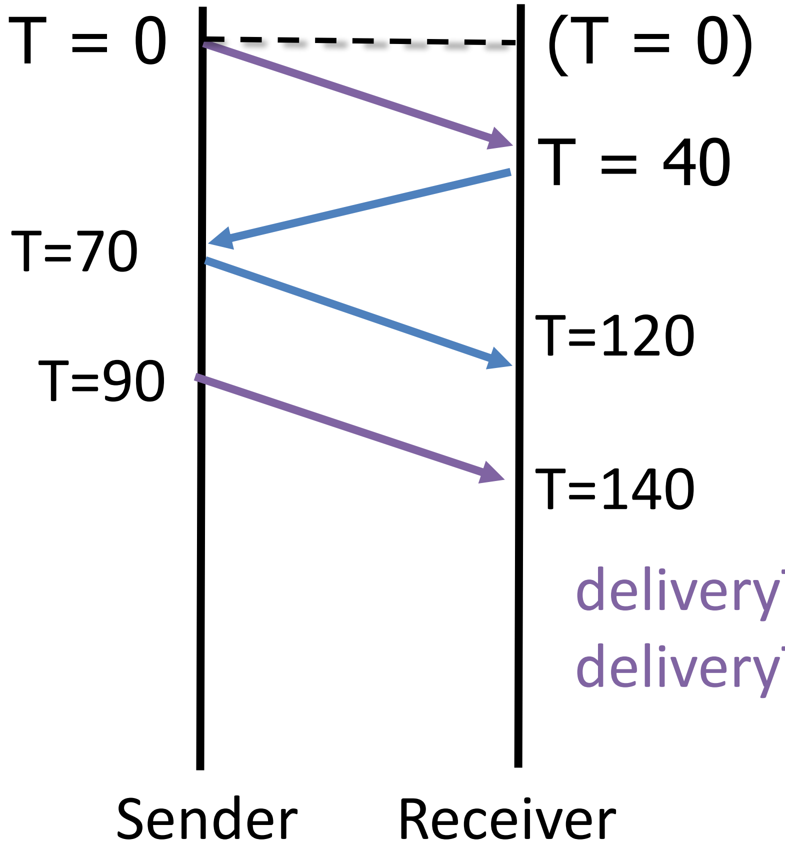
- In practice, it’s a bit more complicated…
- Clocks can drift, so offset between sender clock and receiver clock changes over time
- RTT can change over time
  - Rerouting, congestion
- Need to measure periodically and update our estimates

SRT Approach

- SRT's goal is to maintain the same delivery offset from senderTS despite clock drift and changing RTT
- Slightly different approach than what we described: not trying to "subtract out" network delay
- Basic mechanism:
  - Initialize baseDelta between sender and receiver as rcvtime – sendTS (includes both network delay and clock skew)
    - deliveryTime = sendTS + baseDelta + latencyWindow
  - **RTT Measurement**: sender sends ACKACK in response to ACKs to allow receiver to calculate RTT (time from sending ACK to getting ACKACK) and update its estimate on each ACK
    - use these measurements to adjust baseDelta

Maintaining smooth delivery with clock drift

clock drift! network delay is 40ms, but sender clock slowed down, T=70 (instead of 80)



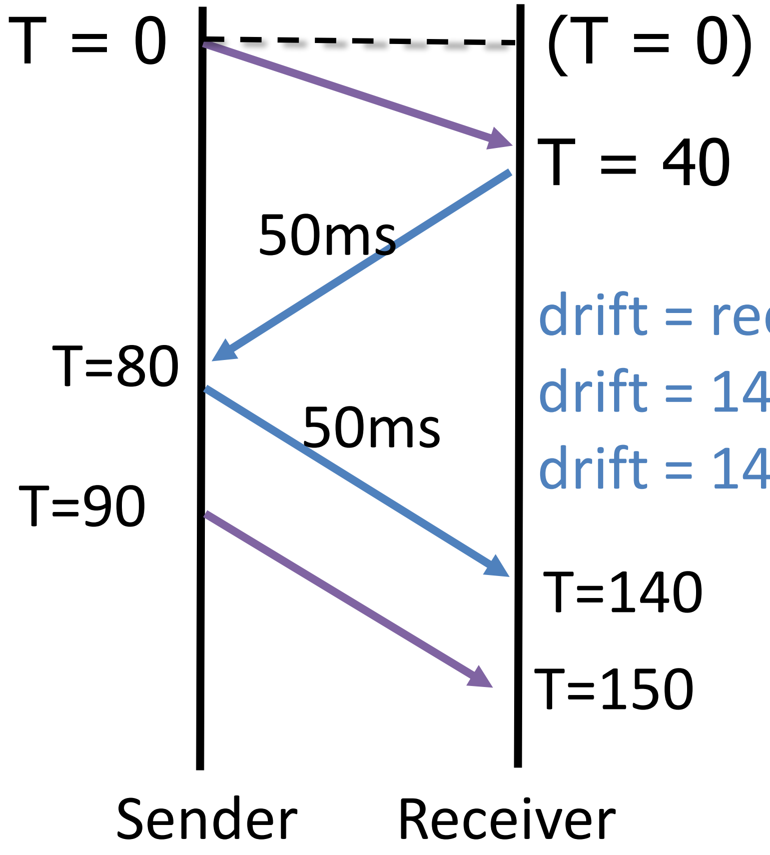
let latencyWindow = 500
baseDelta = 40-0 = 40
deliveryTime = 0+40+500 = 540

drift = recvTime – (ACKACK_TS + baseDelta)
drift = 120 – (70+40) = 10

deliveryTime = sendTS + baseDelta + drift + latencyWin
deliveryTime = 90 + 40 + 10 + 500 = 640

Maintaining smooth delivery with clock drift + changing RTT

clock drifted (sender is now 10ms behind) AND RTT increased from 80ms to 100ms



let latencyWindow = 500
baseDelta = 40-0 = 40
deliveryTime = 0+40+500 = 540

drift = recvTime – (ACKACK_TS + baseDelta) - **deltaRTT/2**
drift = 140 – (80+40) - (100-80)/2
drift = 140 – 120 - 10 = 10

So, I can say T=90 -> T=100 on **rcv clock** if I want to maintain 540ms offset from when the packet was really sent to when it is delivered, deliver at time T=640

Congestion control?

- LiveMode vs FileMode
- LiveMode is what we've been discussing, FileMode is for file transfers (and uses TCP-like congestion control)
- For video, we have a fixed rate we want to send at…just reducing the sending rate would cause problems
- Options:
  - Do congestion control by *dropping* some packets at the sender
  - Signal application to change the bitrate

QUIC Motivation

- Overall goal: Reduce latency for web applications
- New protocol, implemented in userspace on top of UDP

- Why not just try to improve TCP?
  - Making changes is hard…
  - Layering has a cost
    - TCP handshake + TLS handshake -> 3 RTTs before data can be sent
  - Totally ordered bytestream abstraction limits performance (head of line blocking)

QUIC Role in the Web Stack

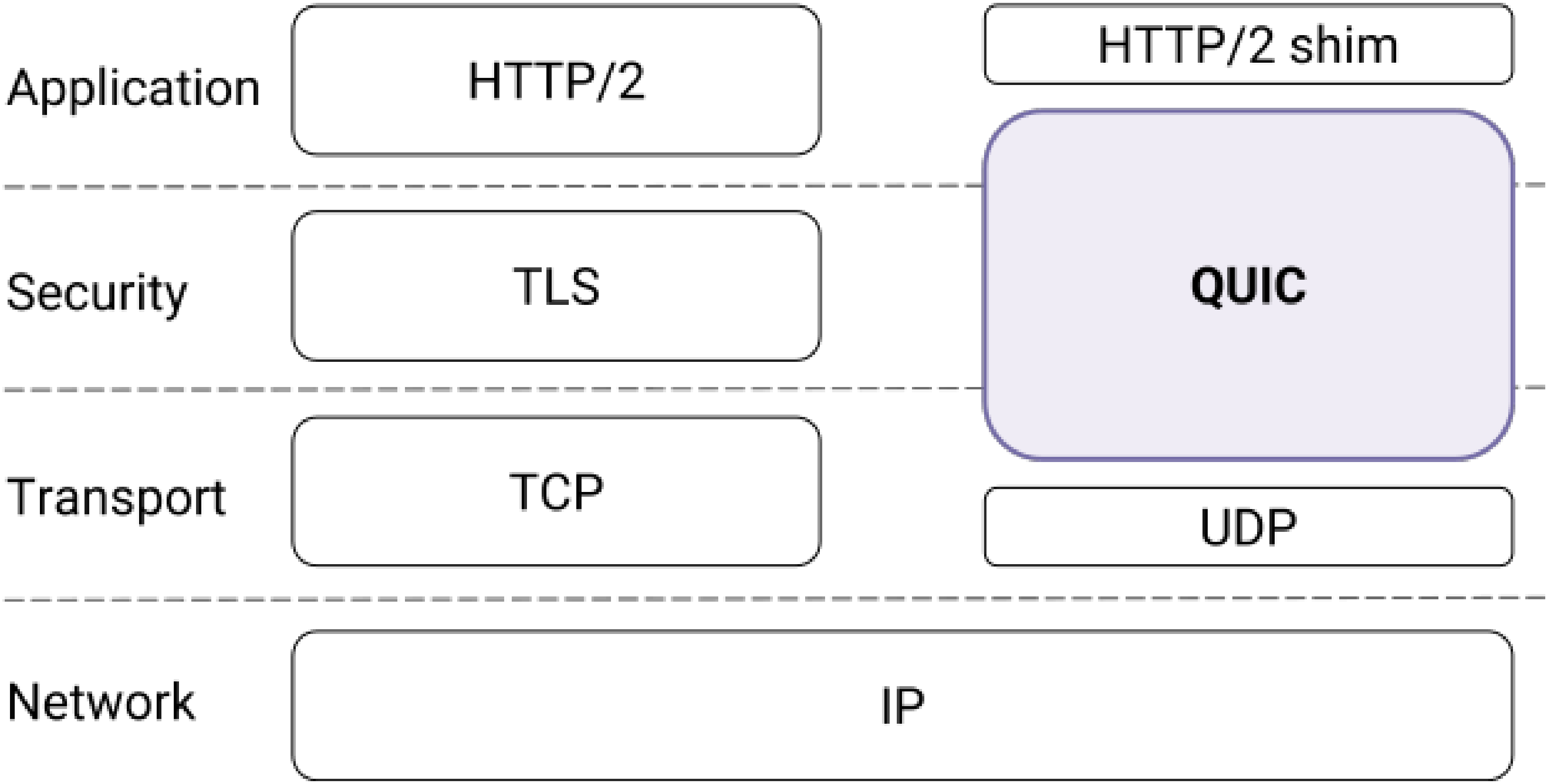
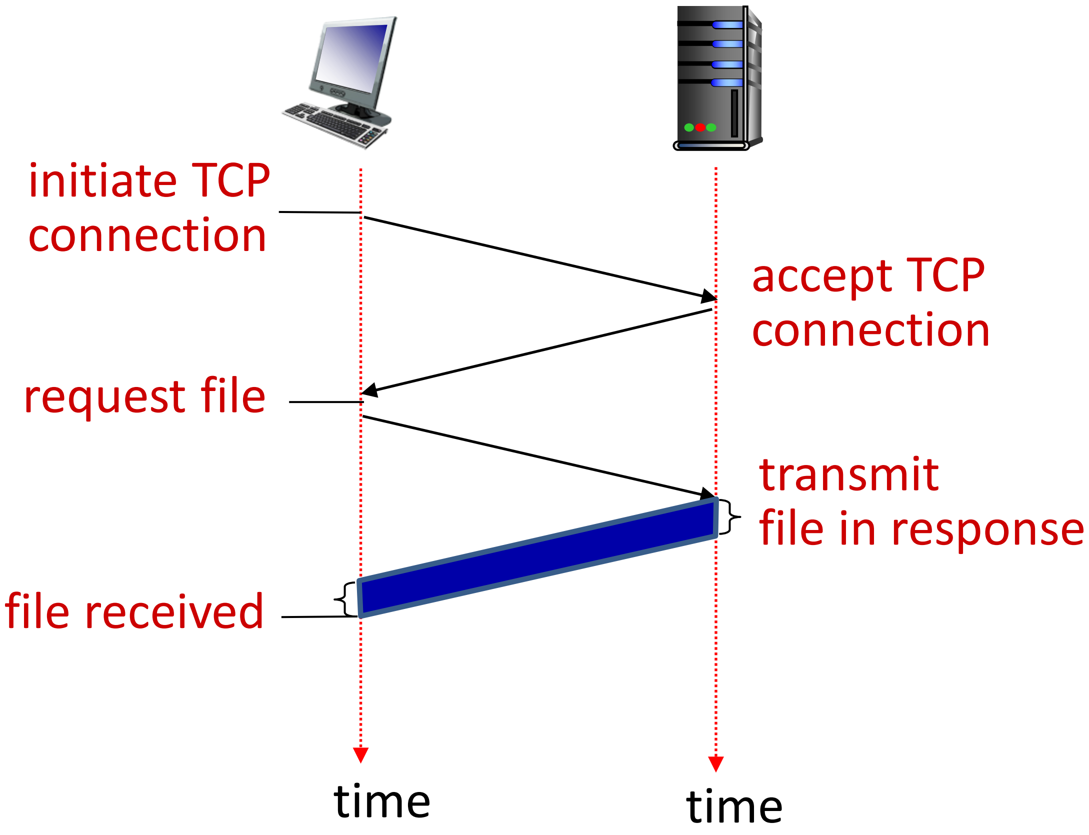


Figure 1: QUIC in the traditional HTTPS stack.

But first...a review of HTTP communication patterns

User enters URL: **`www.someSchool.edu/someDepartment/home.index`**

1. Client **initiates TCP connection** to server at www.someSchool.edu on port 80
2. Server listening for TCP connections on port 80 **accepts** the connection
3. Client sends **HTTP request** message for object someDepartment/home.index to server on TCP connection
4. Server receives request and sends **HTTP response** message containing requested object to client on TCP connection

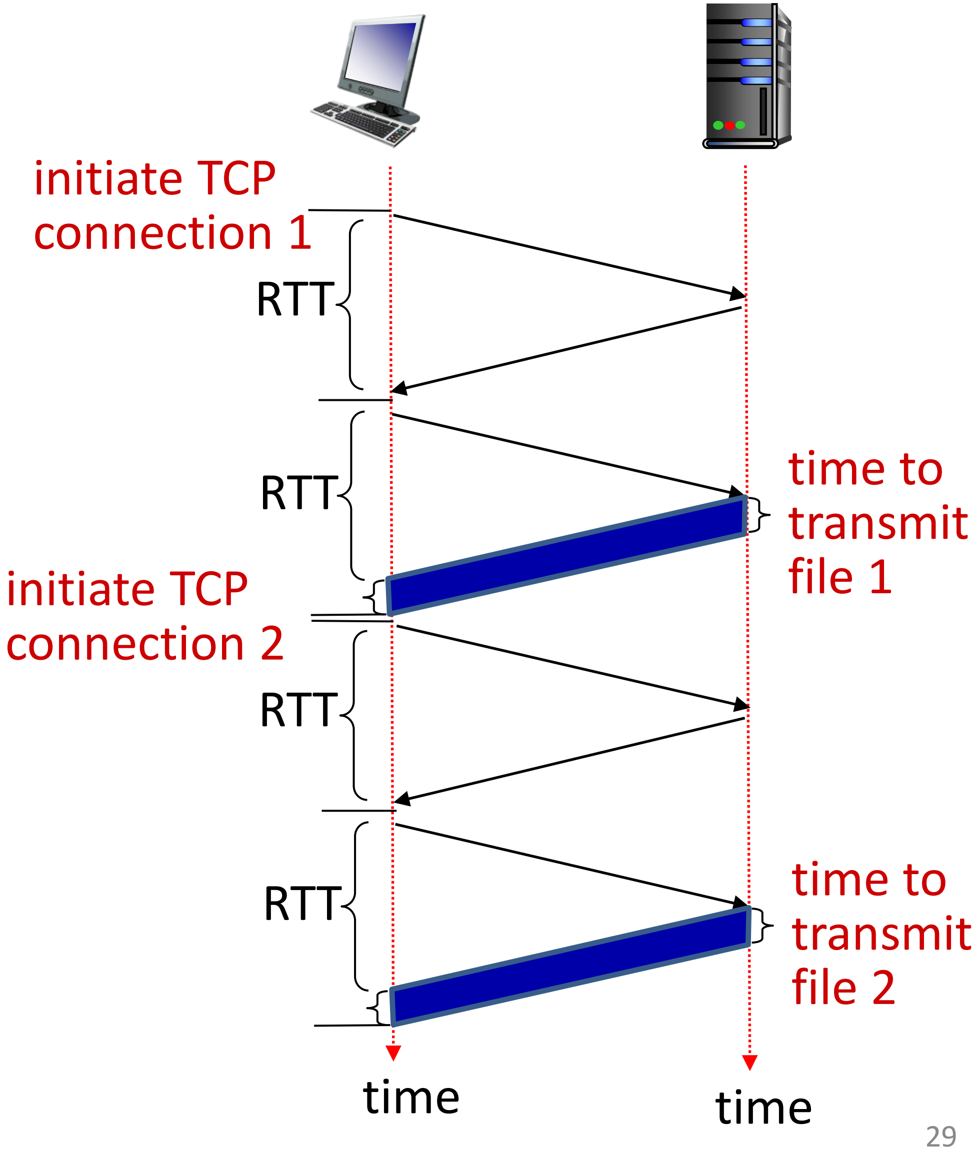


HTTP Performance

- Most webpages include more than one object…how do we retrieve all of them?

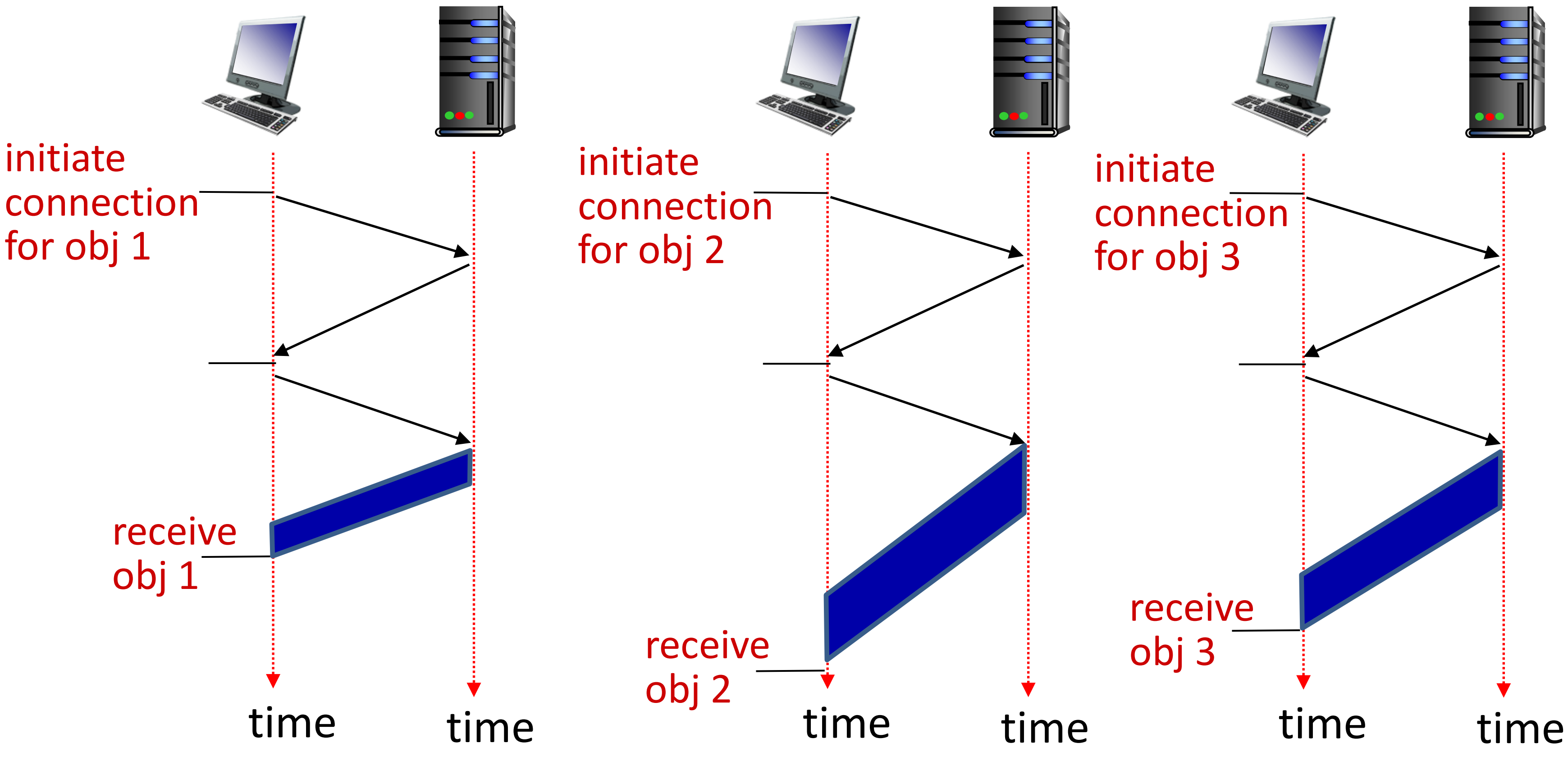
Non-Persistent HTTP (HTTP/1.0)

- Separate TCP connection for each object
- Naively, request objects one at a time (serially)… 2RTT + Transmission time **for each object**



Non-Persistent HTTP (HTTP/1.0)

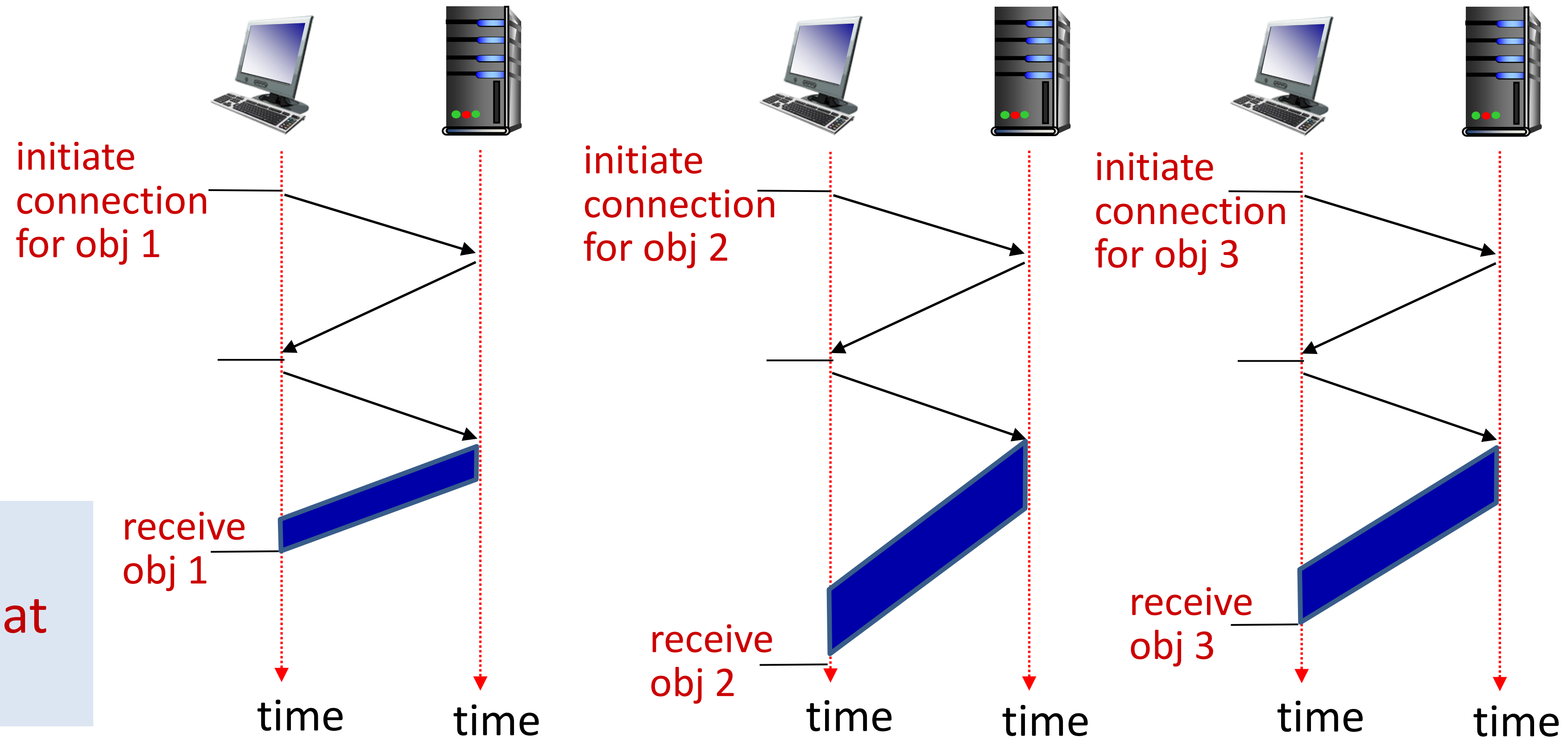
- **How can we reduce response time?**
- Separate TCP connection per object, but run them in **parallel**



Non-Persistent HTTP (HTTP/1.0)

- How can we reduce response time?
- Separate TCP connection per object, but run them in parallel

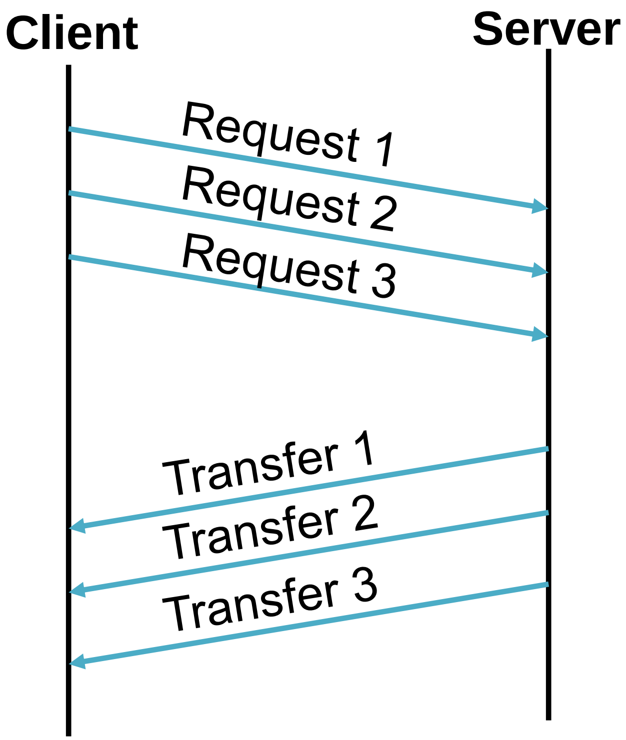
But, not so nice from a network perspective…what about TCP fairness?



Persistent HTTP (HTTP/1.1)

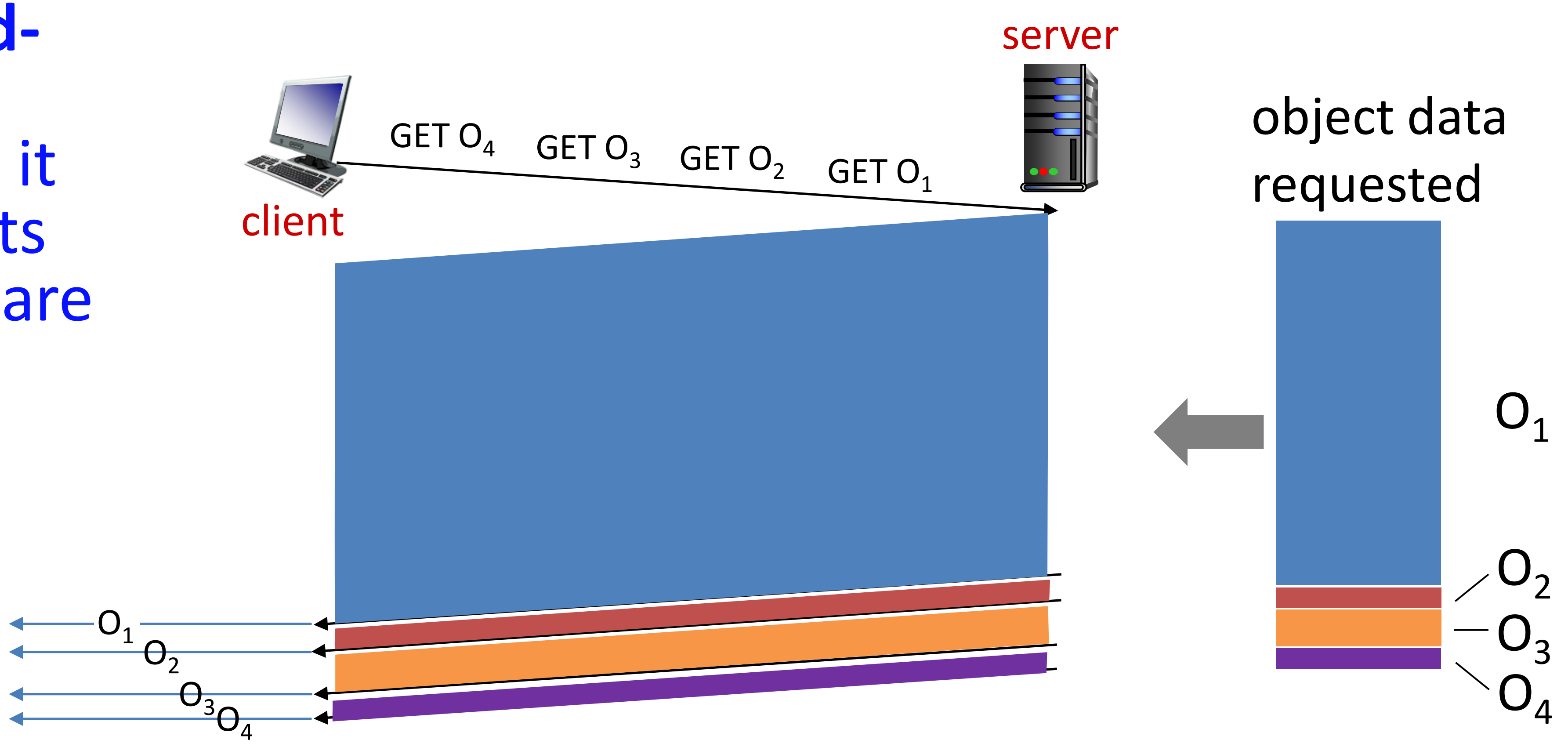
- **Maintain TCP connection across multiple requests**
  - Avoid overhead of setting up and tearing down many connections
  - Better match TCP expectations: allow TCP to learn RTT and bandwidth characteristics, support fair bandwidth sharing
- **Pipelining** to further reduce response time

Pipelined communication pattern



But, persistent pipelined option didn't really stop browsers from opening parallel connections…

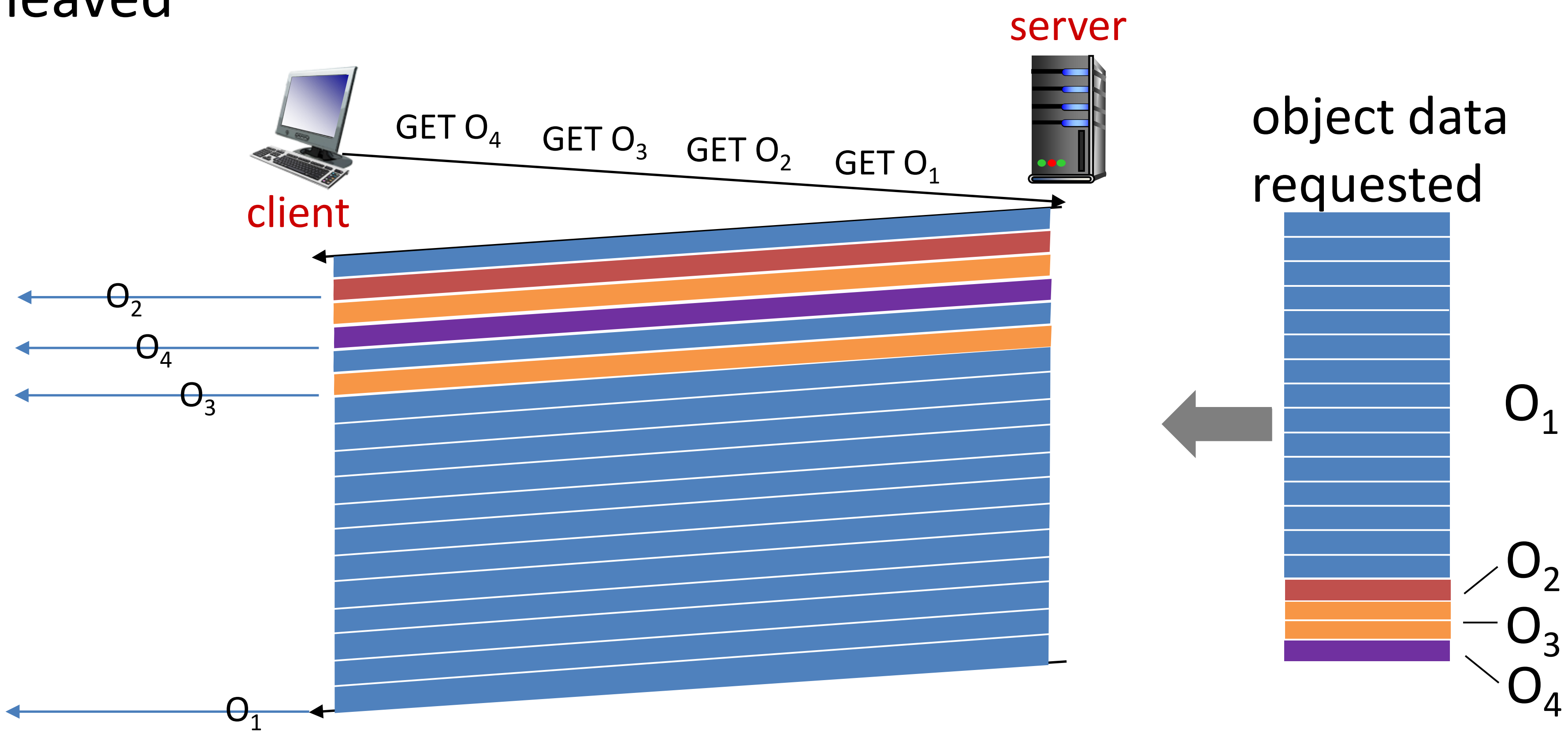
HTTP/1.1 pipelining suffers from **Head-of-Line (HOL) Blocking** because it processes requests in the order they are received



objects delivered in order requested: O_2, O_3, O_4 wait behind O_1

HTTP/2 Multiplexing

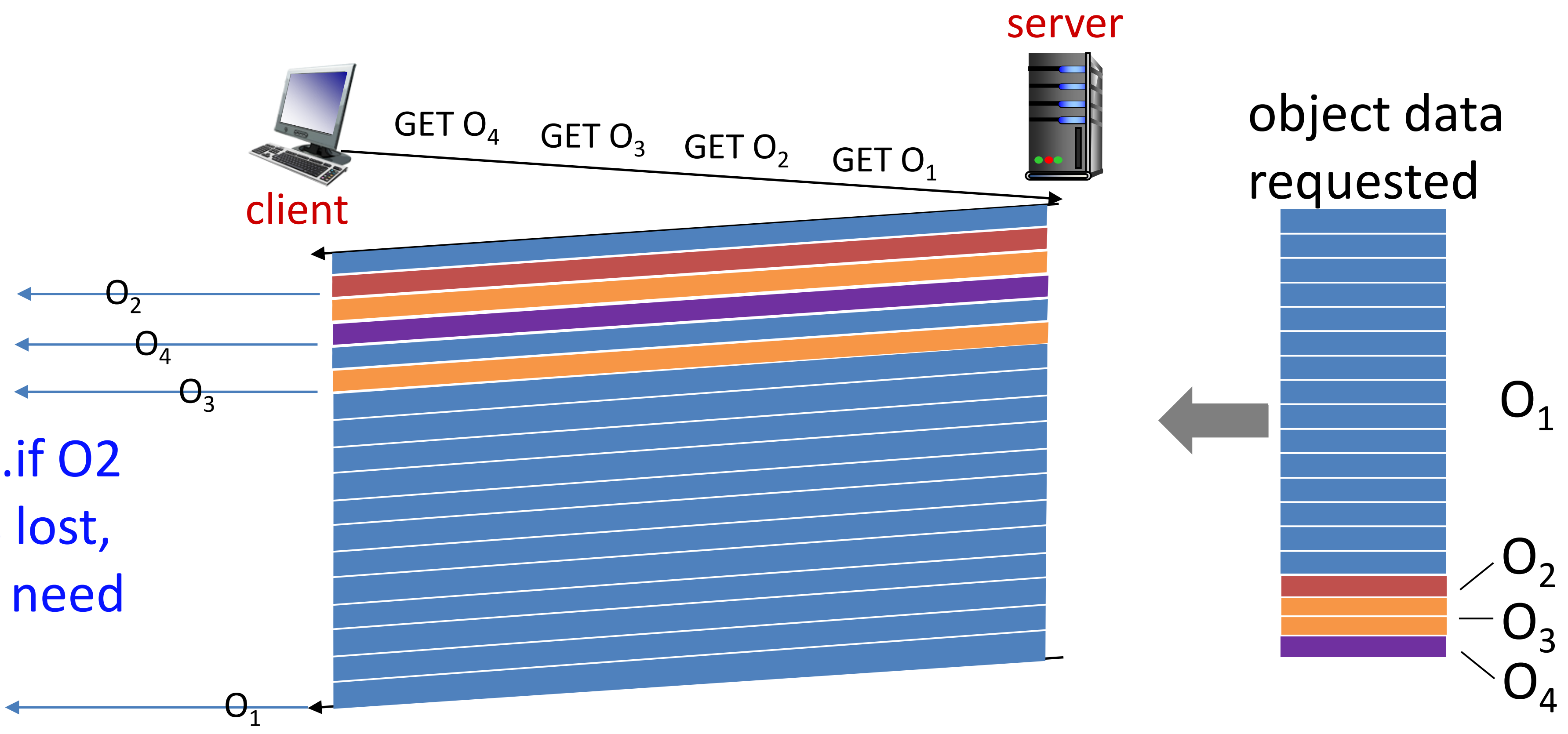
- Objects are divided into *frames*, and frames for different objects can be interleaved



O_2, O_3, O_4 delivered quickly, O_1 slightly delayed

HTTP/2 Multiplexing

- Does this solve head of line blocking problem??



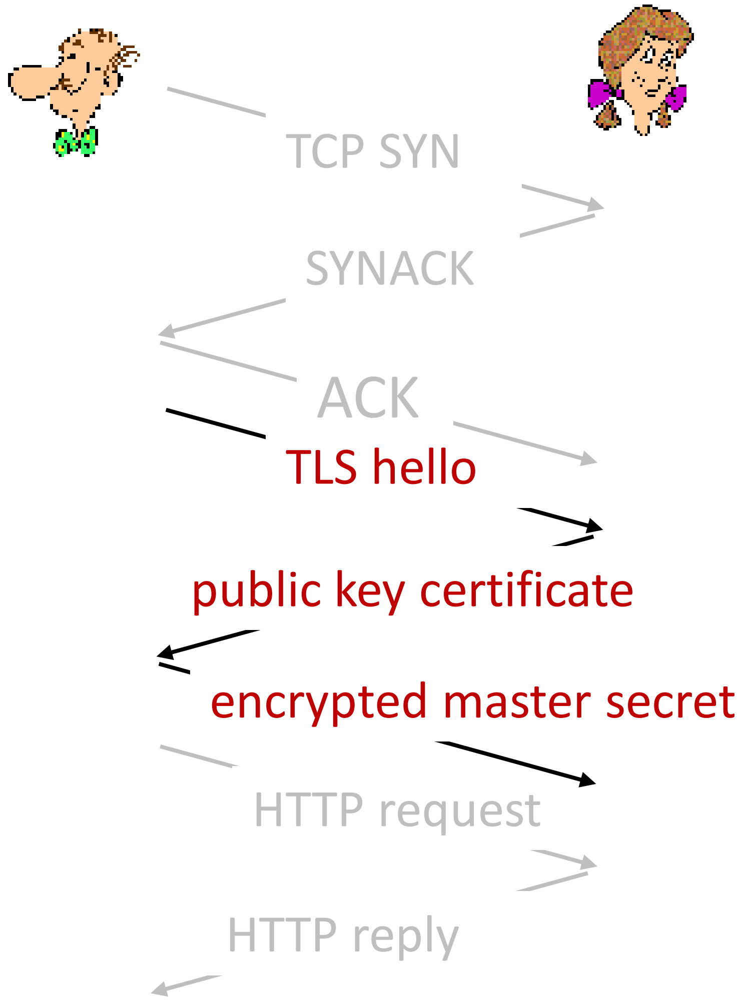
Not quite…if O2 frame gets lost, others still need to wait

O_2, O_3, O_4 delivered quickly, O_1 slightly delayed

Another performance challenge

- We care a lot more about web security than we used to!
- That's mostly a good thing, but what does HTTPS adoption mean for performance?

A simplified view of TLS



TLS handshake phase:

- Bob establishes TCP connection with Alice
- Bob **verifies** that Alice is really Alice
  - public key certificate demonstrates that some trusted authority confirms that Alice really owns this key
- Bob sends Alice a master secret key (MS)
  - MS is **encrypted with Alice's public key** (she can *only read it if she really knows the private part*)
  - MS used to generate session keys used to **encrypt** and **integrity check + authenticate** session data
    - Attacker can't read data or generate valid data without knowing the session keys

QUIC Innovation 1: Squashing the Layers

- **Combines transport handshake with crypto handshake**
- Also improved the handshake itself:
  - Reduces crypto handshake that took 2 RTT in TLS 1.2 to single RTT (now part of TLS 1.3)
  - Caches information about a a given origin to enable 0-RTT handshake for connections to known origin (now part of TLS 1.3)

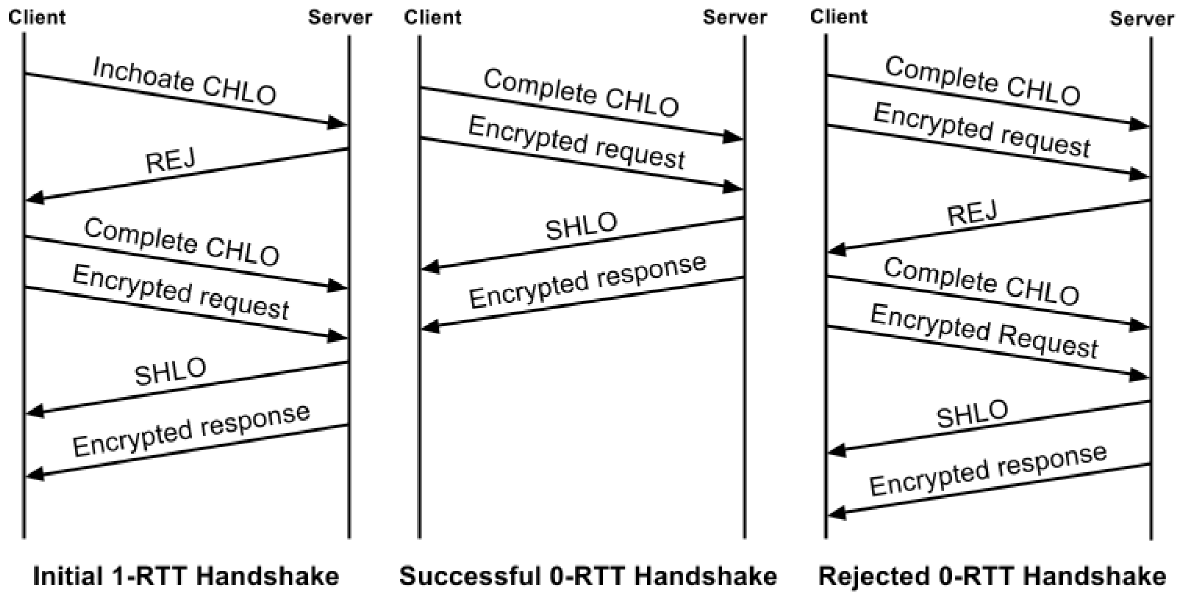
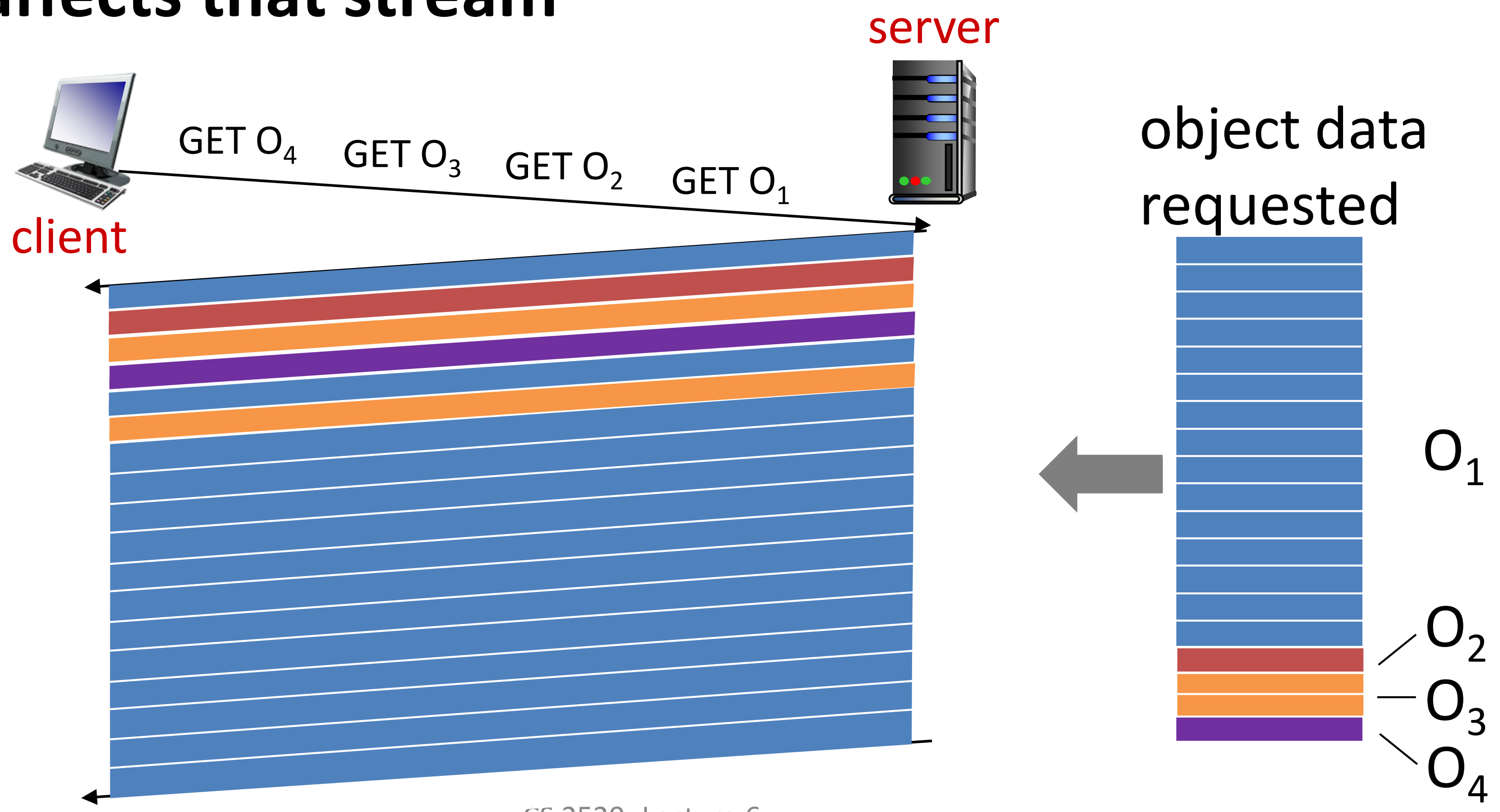


Figure 4: Timeline of QUIC's initial 1-RTT handshake, a subsequent successful 0-RTT handshake, and a failed 0-RTT handshake.

QUIC Innovation 2: Stream Multiplexing

- Changes the single *totally ordered* byte stream abstraction
- Explicitly provides a “stream” abstraction, where a lost packet in one stream **only affects that stream**



“Fixing” TCP issues: a couple other interesting points

- QUIC separates *data* sequence number from *packet* sequence number
  - Retransmission of same data gets a **new packet sequence number**
  - Lets lost retransmissions be identified without waiting for a timeout (can see gap in ACKed packet sequence numbers)
- From its 2017 SIGCOMM paper: “QUIC acknowledgments explicitly encode the delay between the receipt of a packet and its acknowledgment being sent.”
- Why is this useful? (think about BBR)

Summary

- TCP and UDP have worked really well as our foundational transport protocols on the internet, but aren't a perfect fit for all applications
- We can implement custom protocols to get better performance and better match application requirements
- Implementing new protocols at the transport layer (and getting them adopted) is tough due to long timescales, and complex web of assumptions and dependencies
- Innovation at the application layer is much easier! And has been quite successful in practice

Extra resources

- Academic papers:
  - "The QUIC Transport Protocol: Design and Internet-Scale Deployment": https://dl.acm.org/doi/abs/10.1145/3098822.3098842
- Standards docs:
  - https://datatracker.ietf.org/doc/html/draft-sharabayko-srt-01
  - https://datatracker.ietf.org/doc/html/rfc9000/
- Code is available to investigate:
  - https://github.com/Haivision/srt
  - https://github.com/google/quiche