Flexible Transport

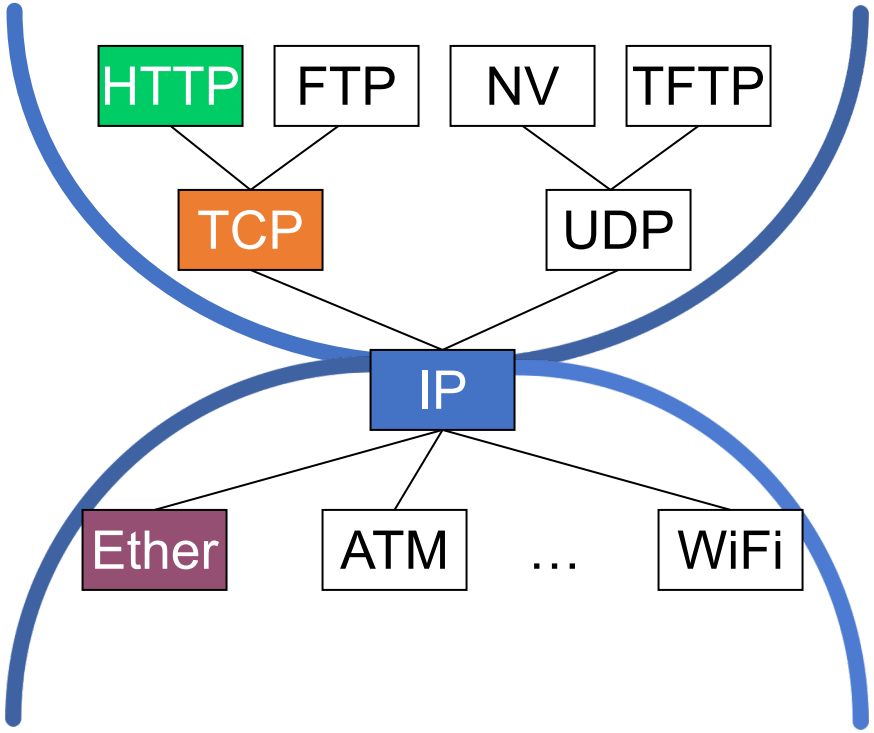
Lecture 11, Computer Networks (198:552)

Fall 2019

RUTGERS
UNIVERSITY | NEW BRUNSWICK

Modularity through layering

Protocols “stacked” in endpoint and router software/hardware



Two Main Transport Layers

- User Datagram Protocol (UDP)
  - Abstraction of independent messages between endpoints
  - Just provides demultiplexing and error detection
  - Header fields: port numbers, checksum, and length
  - Low overhead, good for query/response and multimedia

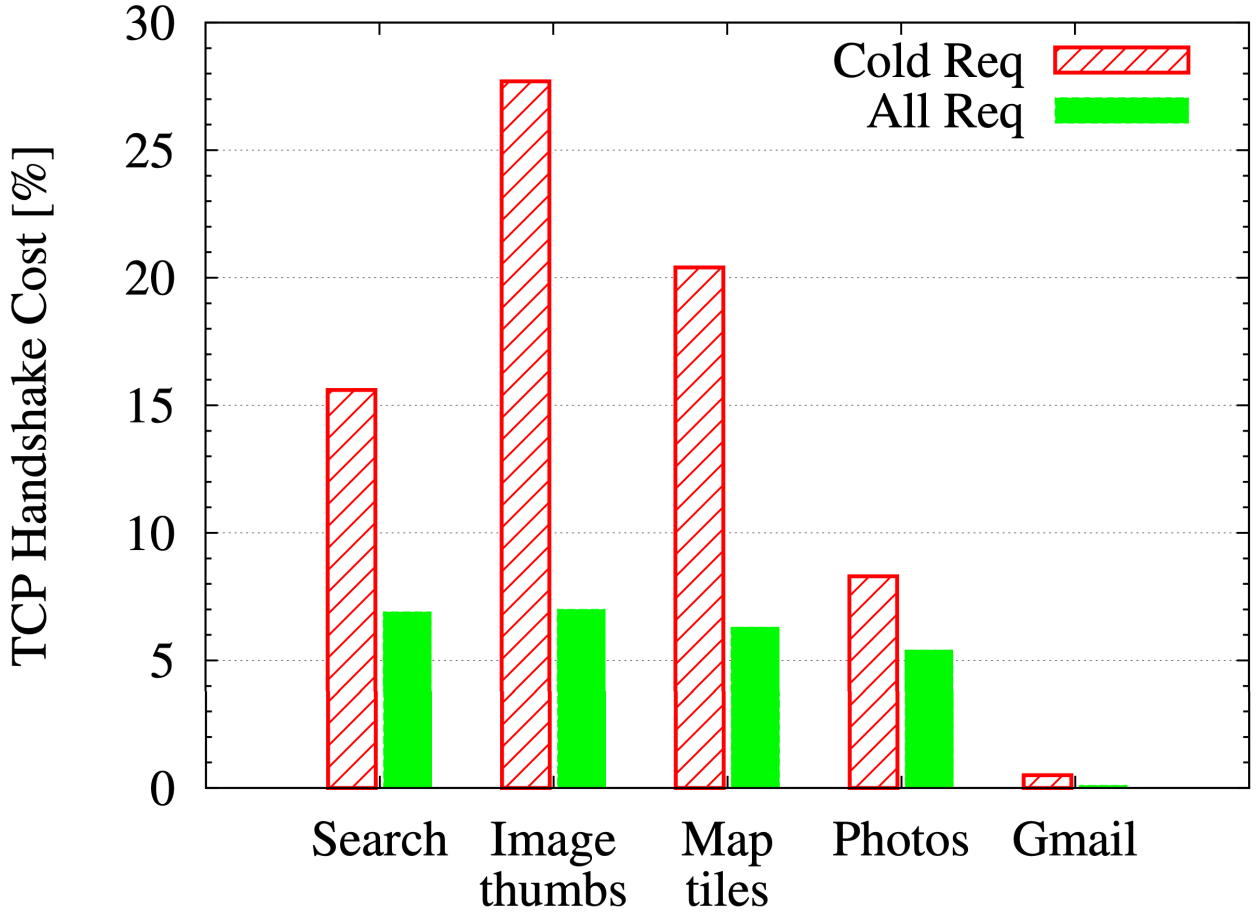
- Transmission Control Protocol (TCP)
  - Provides support for a stream of bytes abstraction

Transmission Control Protocol (TCP)

- Multiplexing/demultiplexing
  - Determine which conversation a given packet belongs to
  - All transports need to do this
- Reliability and flow control
  - Ensure that data sent is delivered to the receiver application
  - Ensure that receiver buffer doesn’t overflow
- Ordered delivery
  - Ensure bits pushed by sender arrive at receiver app in order
- Congestion control
  - Ensure that data sent doesn’t overwhelm network resources

Things we'd like to change about TCP (1/N)

- 3-way handshake: too much latency (esp. small web requests)



Request latencies to Google services

Source: "TCP Fast Open," CoNext 2011

Problem made worse by adding security (HTTPS / TLS) over TCP.
TLS handshake adds more round trips.

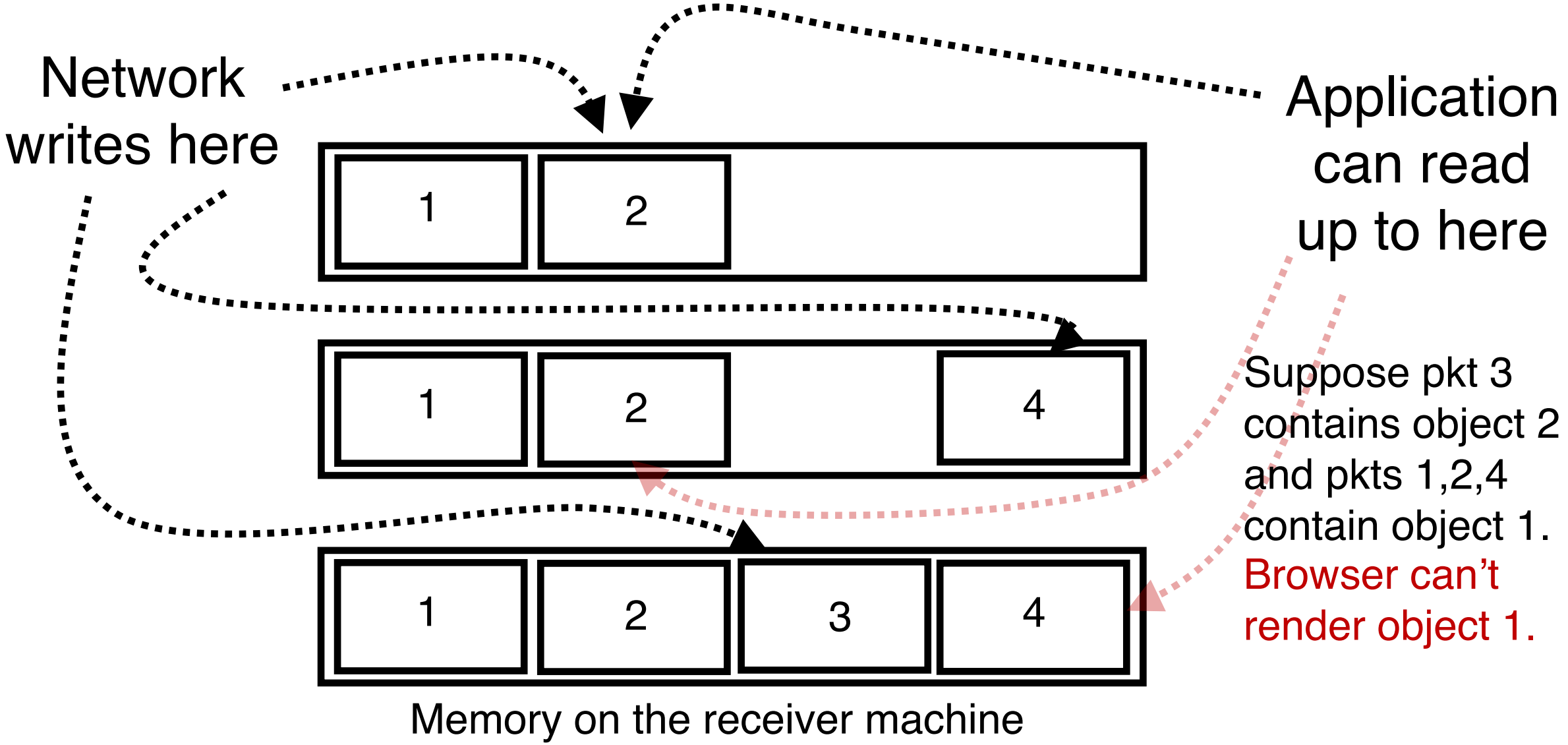
Things we'd like to change about TCP (2/N)

- Ability to handle streams of data within a connection
  - Q: examples of apps that might need it?
  - Typical web page contain tens, if not hundreds of objects
  - A browser could render partial pages with objects that finished downloading to provide a better user experience

- One solution is to open a TCP connection for each object
  - High overhead -- at client and server
  - Connections cannot learn about network conditions together

- Streams within single TCP connection:
  - head-of-line blocking

Head-of-line blocking at client side



Things we'd like to change about TCP (3/N)

- TCP measures a connection round-trip time RTT as the time between transmitting each packet and receiving its ACK
  - Recall: What's the RTT useful for?
  - Good RTT estimate critical to know when to retransmit lost pkts
  - Also for algorithms using RTT to determine sending rate

- What happens to the RTT estimate when a packet is retransmitted?
  - Today: ignore RTT for any packet transmitted more than once (Karn's algorithm)

- Issue: TCP conflates two uses of sequence numbers:
  - Ordered data delivery (know which data to deliver first to app)
  - Reliable delivery (track which data was received)

Things we'd like to change about TCP (3/N)

There are many other things we'd like to change, but we'll only talk about the 3 above in the context of QUIC.

What's so hard about changing TCP?

- You may need to change the operating system kernel
- You may need to change the operating system kernel on all servers and clients (ex: all your laptops and phones!)
- You may need to change the entire network!
- Middleboxes sitting in the network may change, drop, or add info on packets
- Middleboxes may drop packets if they don't understand something on the packet

QUIC

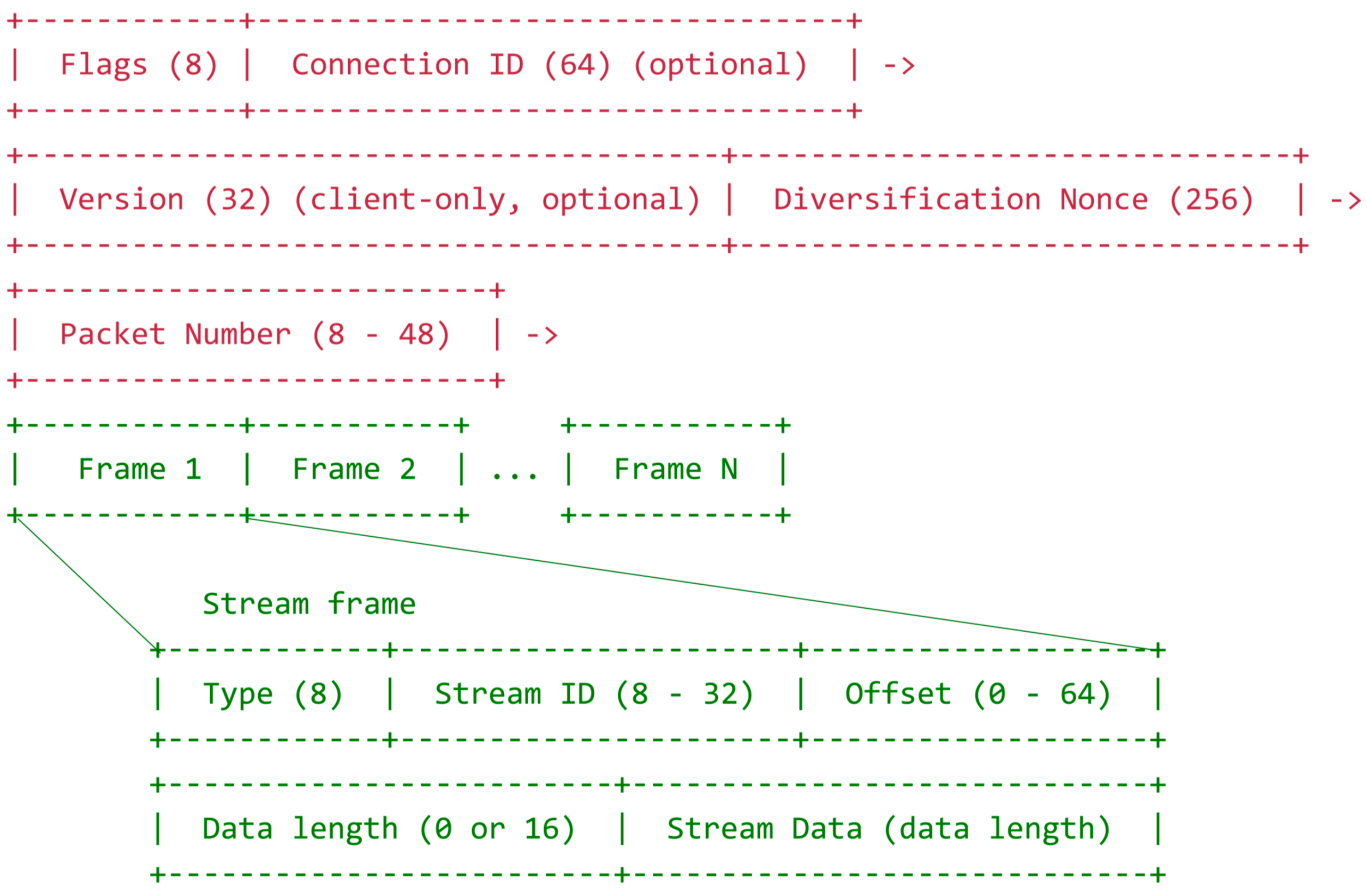
- Designed over UDP with fresh packet formats at app layer

- Issue #1: Better handshake procedure
  - Designed with security & encryption in mind from the beginning
  - Almost everything is encrypted
  - If middlebox can’t read a piece of info, it can’t make any decisions based on that info
  - In particular, can’t change packet without endpoints noticing

- Issue #2: Support lightweight streams natively
- Issue #3: Better RTT estimation for retransmissions

#2: Streams without HOLB

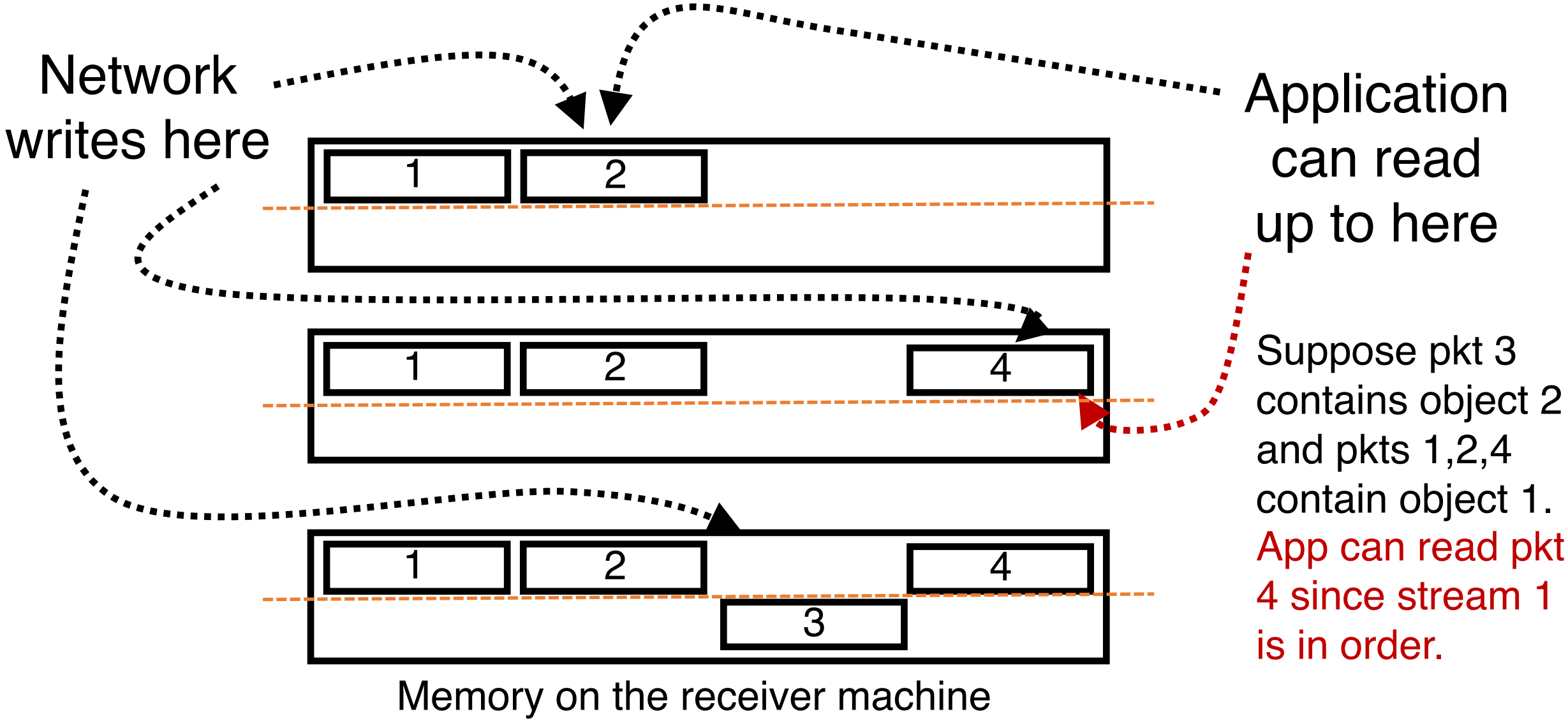
- QUIC supports packet format with frame-level data



#2: Streams without HOLB

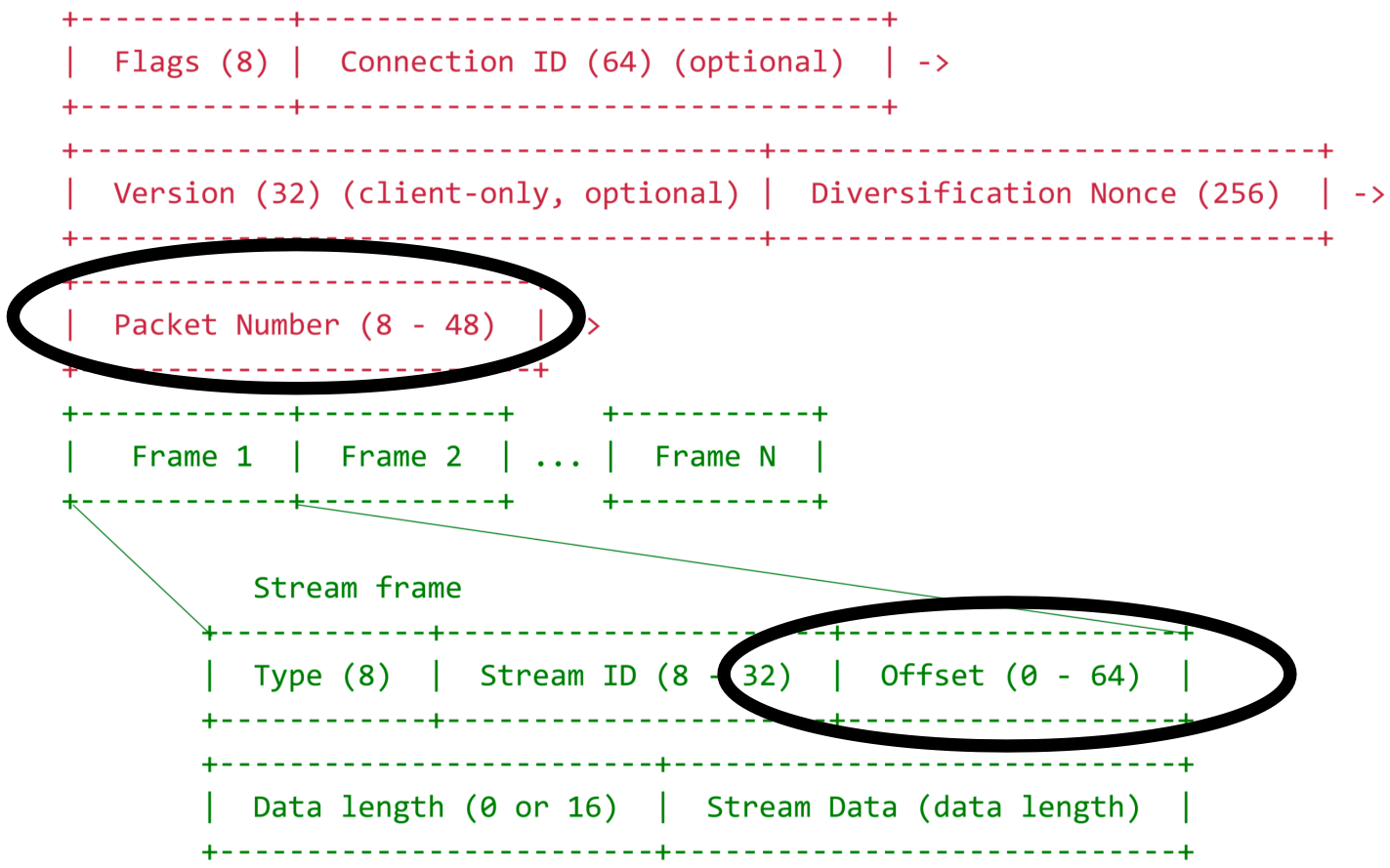
- Stream-level flow control
  - Separate advertised window per stream and for connection
  - Window update frames per stream
- Stream-level sending rate mechanisms (“priorities”)
- QUIC receiver can deliver packets to app as long as stream packets received in order (even if connection’s packets are not)
- Ideas are known from prior work on “Structured Streaming”

No HOLB across streams



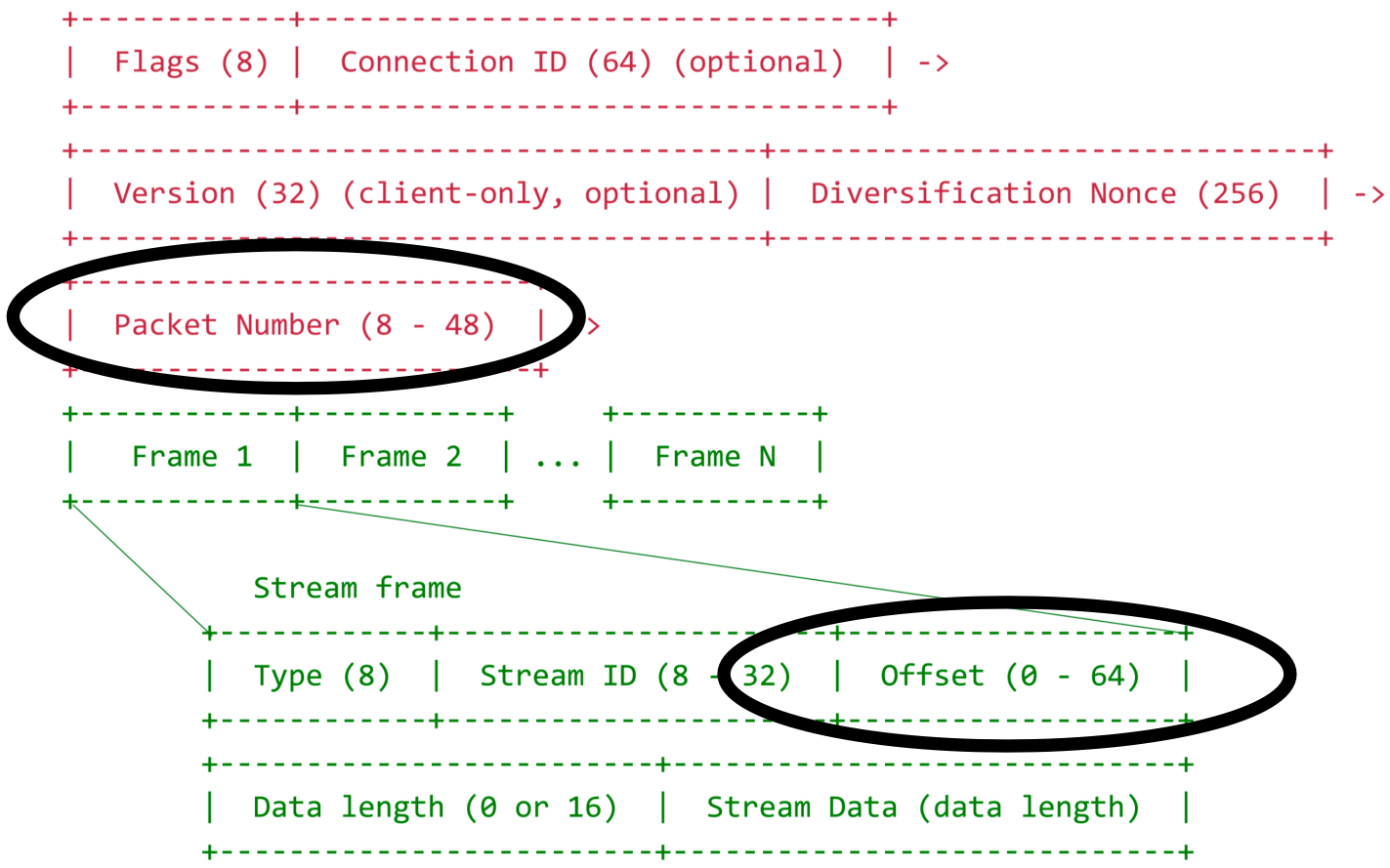
#3: Better RTT estimation

- Use separate sequence numbers for data and packet delivery



#3: Better RTT estimation

- Decouple reliability completely from ordered delivery



#3: Better RTT estimation

- Decouple reliability completely from ordered delivery
- Packet numbers are monotonically increasing (and hence unique)
  - Distinct from frame offsets within stream

- Use time between the transmission of a packet and its ACK (identifiable using unique packet number) for RTT estimate
  - Regardless of original transmission or retransmission

- Better RTTs when you need them the most: loss recovery

- Also have more ACK blocks (256) than TCP SACK

#1 Lower-latency handshake

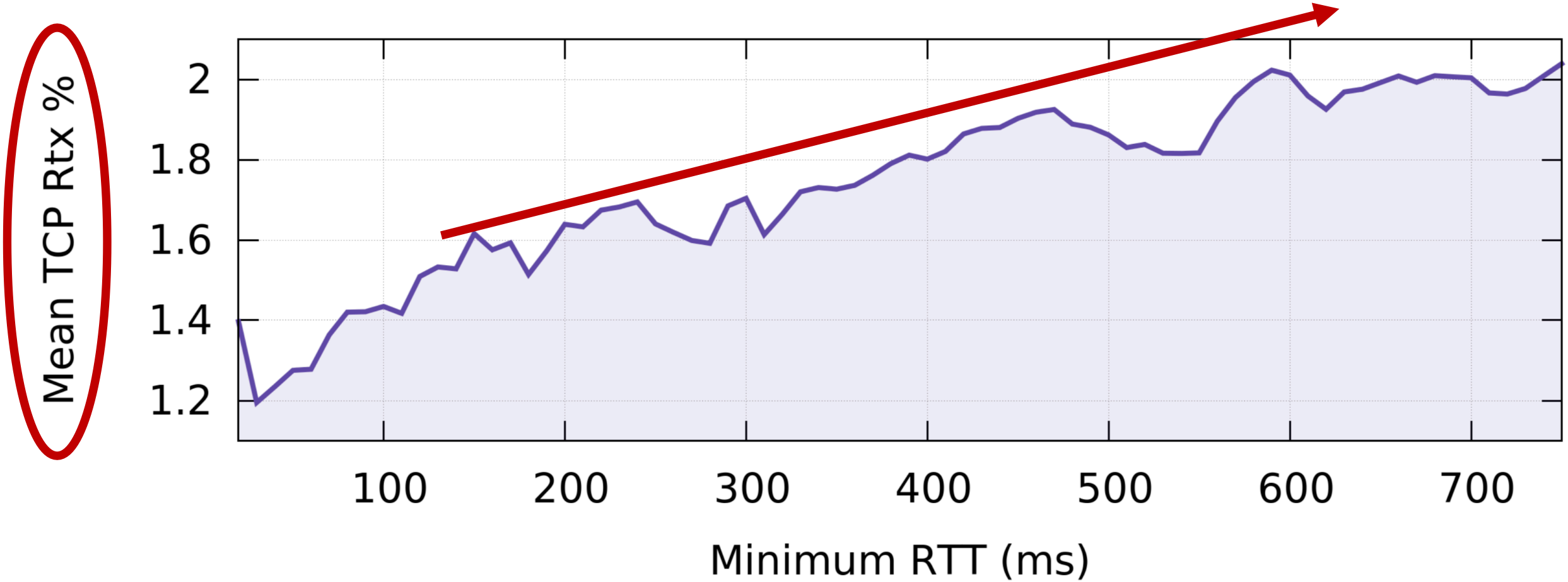
- “Common case” 0-RTT handshake
- Otherwise, 1-RTT or (more rarely) 2-RTT handshake
- Merge cryptographic information within the first packet
- Use cached credentials to make subsequent handshakes faster
  - Optimistically assume handshake succeeds
  - Server can reject or drop request without doing too much work
- (More details to come…)

Results

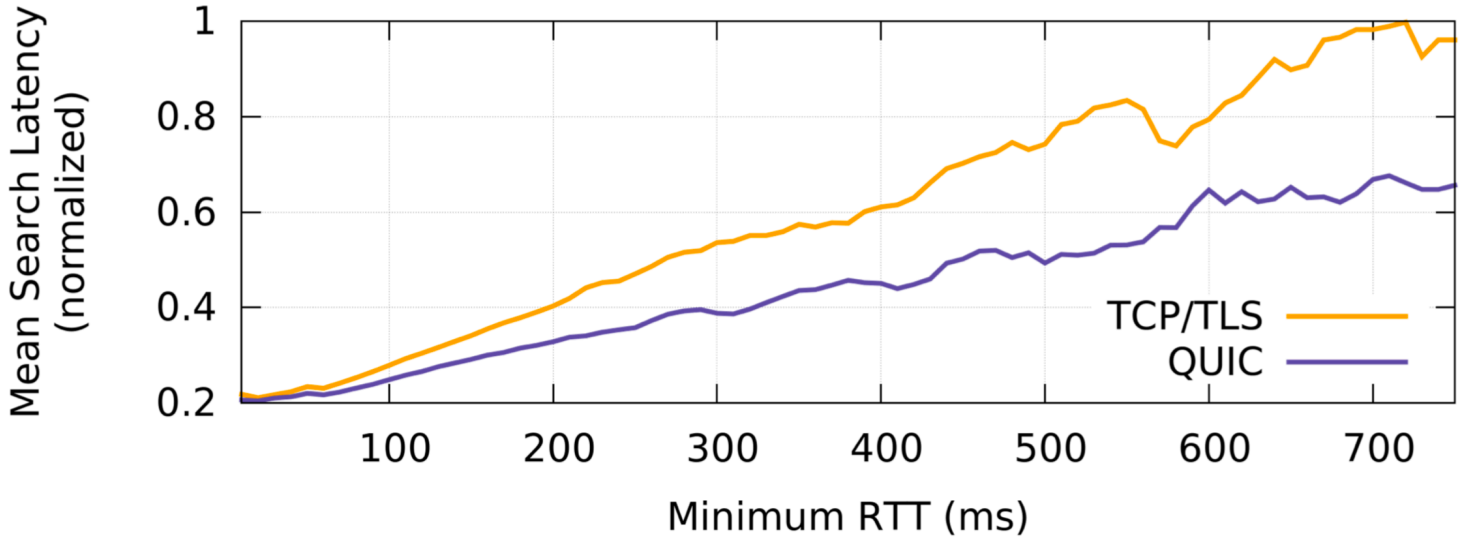
Latency improvements

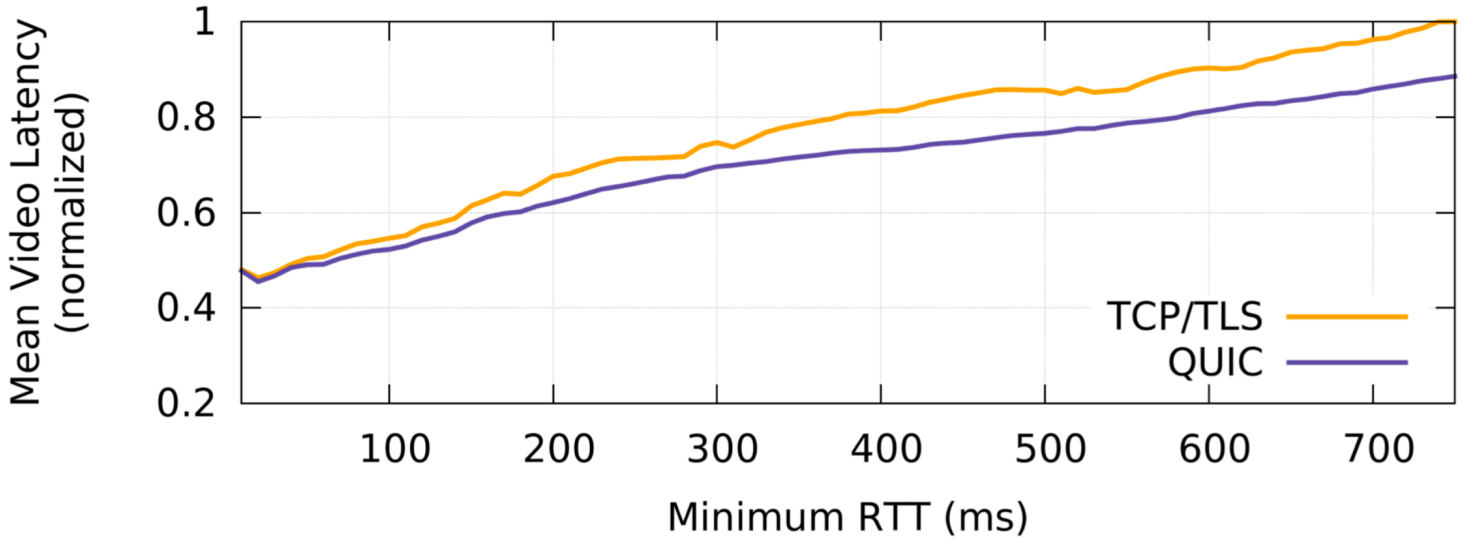
	% latency reduction by percentile							
		Lower latency				Higher latency		
	Mean	1%	5%	10%	50%	90%	95%	99%
Search								
Desktop	8.0	0.4	1.3	1.4	1.5	5.8	10.3	16.7
Mobile	3.6	-0.6	-0.3	0.3	0.5	4.5	8.8	14.3
Video								
Desktop	8.0	1.2	3.1	3.3	4.6	8.4	9.0	10.6
Mobile	5.3	0.0	0.6	0.5	1.2	4.4	5.8	7.5

Retransmission prevalence to Google clients



Improvements by minimum RTT





Hypothesis: better loss recovery behavior of QUIC results in higher improvements as minRTT increases

Not proven conclusively through targeted experiments in the paper. But possible to do (someone should do it!)

In general, good idea to combine macrobenchmarks with microbenchmarks

Video rebuffer rate

		% rebuffer rate reduction by percentile				
		Fewer rebuffers			More rebuffers	
	Mean	< 93%	93%	94 %	95%	99%
Desktop	18.0	*	100.0	70.4	60.0	18.5
Mobile	15.3	*	*	100.0	52.7	8.7

Improvements by region

			% Reduction in Search Latency		% Reduction in Rebuffer Rate	
Country	Mean Min RTT (ms)	Mean TCP Rtx %	Desktop	Mobile	Desktop	Mobile
South Korea	38	1	1.3	1.1	0.0	10.1
USA	50	2	3.4	2.0	4.1	12.9
India	188	8	13.2	5.5	22.1	20.2

Lessons from QUIC

- Layering enables modularity but can hurt performance (TCP+TLS)
- Benchmark application metrics (macrobenchmarks) and protocol metrics (microbenchmarks) to ensure
  - That features are working correctly
  - That features are useful to applications
- User space networking has performance caveats
  - High CPU usage
  - Harder to write applications (PACKET_RX_RING + mmap)
- Extra data transfer while bandwidth-limited (ex: video start) is bad
  - Google tried FEC, but performance didn’t improve as imagined

Discussion of QUIC

What you need to know to understand the QUIC handshake

- Security and cryptography basics:
  - Confidentiality. Encryption techniques. Example of a technique (ex: Diffie-hellman key exchange) for secret communication. Difference between public and private key crypto. Forward secrecy. Specific reasons to prefer private or public.
  - Authentication. How certificates allow you to do this.
  - Integrity: how one-way functions allow you to do this.
- TLS: goals: provide server authentication, confidentiality, integrity. The TLS handshake. What it achieves.
- QUIC handshake: how the different components come together to achieve TLS goals. How the results of the handshake can be used later.

Some lessons from QUIC handshake

- Encrypt as much as possible:
  - Security properties, but also avoid protocol entrenchment: need endpoints to be able to change without the core interfering
- If you exchange metadata in plaintext (e.g., handshake packets), use all the plaintext while generating shared secrets. Otherwise, someone can tamper with integrity.
  - More a security principle than a networking principle
  - TLS does the same thing (prevent “downgrade” attacks)