4TC

PRS

Programmation Réseau et Système

membre de
UNIVERSITÉ DE LYON

Département Télécommunications
Services & Usages

INSA LYON

PRS

- Pedagogical team:
  - **Oana IOVA**, Frédéric LE MOUEL, Razvan STANICA
  - Thank you Razvan STANICA for the initial slides

- Objectives
  - Make the link between “network” and “programming”
  - Understand the role and the working of the transport layer
  - First use of the Sockets API

PRS

- Connaissances :
  - Couche transport: rôle, mécanismes de base, services offerts
  - Les protocoles TCP et UDP
  - Les notions de contrôle de flux et contrôle de congestion
  - Concept de socket
  - Appels système
  - Architecture client-serveur

- Capacités :
  - Concevoir et développer une couche réseau transport adaptée à des scénarios donnés
  - Implanter des architectures réseau en utilisant l'API Sockets

PRS

- Structure du cours
  - 4h CM (TCP reminder and Sockets API)
  - 2h TD (understanding TCP)
  - 8h “guided” TP on Sockets API
  - 16h TP on TCP mechanisms
  - 12h Projet – Implementation of a transport layer for a given scenario



- Evaluation
  - Project presentation
  - Project testing

4TC

PRS

Transmission Control Protocol

membre de
UNIVERSITÉ DE LYON

Département Télécommunications
Services & Usages

INSA LYON

TCP

- TCP = Transmission Control Protocol
  - Basic concepts already discussed in 3TC NET
- Pre-requisites for PRS
  - TCP header format
  - Connection management
  - TCP state machine
- PRS objective: TCP congestion control



TCP - Reminder

TCP Functioning

• TCP fragments the application data, forming segments with a size decided by TCP, which are transmitted sequentially.

• When a segment is passed to the network layer, TCP starts a timer waiting for an ACK segment from the destination.

• If the timer reaches 0 without an ACK **…..???**

• Timer value is very important, based on round trip time estimation. **WHY???**

• Each time TCP receives a segment, it sends an ACK.

• The header and the data integrity in a TCP segment are protected using **…..???**

• TCP puts in order the received segments before indicating the reception to **…..???**

TCP - Reminder

TCP Functioning

• TCP fragments the application data, forming segments with a size decided by TCP, which are transmitted sequentially.

• When a segment is passed to the network layer, TCP starts a timer waiting for an ACK segment from the destination.

• If the timer reaches 0 without an ACK, the segment is considered lost and retransmitted.

• Timer value is very important, based on round trip time estimation. **WHY???**

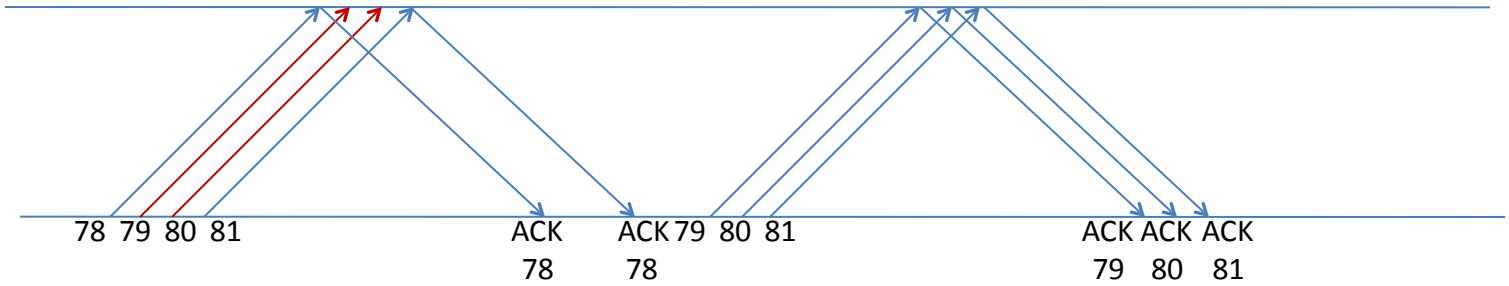
• Each time TCP receives a segment, it sends an ACK.

• The header and the data integrity in a TCP segment are protected using a checksum.

• TCP puts in order the received segments before indicating the reception to the application.

TCP

- Duplicate ACKs
  - In normal operation, a receiver is not allowed to acknowledge discontiguous segments
  - The reception of an out-of-order segment results in the acknowledgement of the last contiguous segment (a duplicate ACK)



TCP

Sliding Window

- Allows the transmission of several segments before the reception of an ACK.
- https://www.youtube.com/watch?v=zY3Sxvj8kZA
- https://www.youtube.com/watch?v=lk27yiITOvU



TCP

Sliding Window

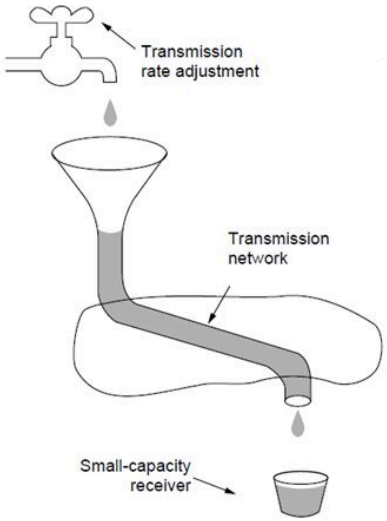
- Allows the transmission of several segments before the reception of an ACK.
- https://www.youtube.com/watch?v=zY3Sxvj8kZA
- https://www.youtube.com/watch?v=lk27yiITOvU



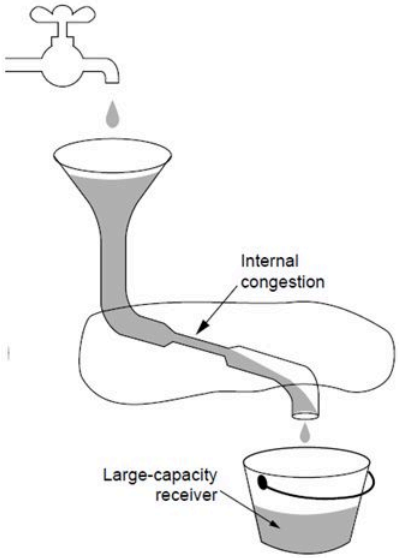
Window Size Matters?

TCP

- What can go wrong?



A fast network feeding a low-capacity receiver



A slow network feeding a high-capacity receiver

TCP

- Flow Control
  - Manage the data rate at the transmitter in order to not overwhelm a slower receiver
- Congestion Control
  - Manage transmission rate in order to avoid network congestion collapse
- End-to-End Argument
  - Represents the philosophy behind TCP (and behind Internet)
  - Data flow rate is controlled by end hosts
  - The network does not provide any congestion or flow control support



TCP

(Back to) Sliding Window

- Allows the transmission of several segments before the reception of an ACK.
- A transmission window for flow control.
- A congestion window for congestion control.
- When window=0, the transmission is frozen.
- Reduce the congestion window when ACKs are not received (a sign of congestion).
- Increase the congestion window when segments are acknowledged.



TCP

- TCP Vocabulary
  - Segment = the TCP payload data unit (different from “message”, “packet”, or “frame”)
  - Maximum Segment Size = the size of the largest segment that can be transmitted/received. The result of a negotiation between end hosts
  - Receiver Window (rwnd or awnd) = the number of segments a host can receive at a given moment. Used for flow control purposes
  - Congestion Window (cwnd) = a maximum number of segments that can be transmitted by a host, decided by congestion control mechanisms



TCP

- Basic transmission principle
  - At any given time, a TCP host must not transmit a segment with a sequence number higher than the sum of the highest acknowledged sequence number and the minimum between cwnd and rwnd
- Important metric
  - Round-Trip Time (RTT): the time between the transmission of the segment and the reception of the ACK. RTT can vary significantly during network operation, so TCP keeps an updated estimated value



TCP

- FlightSize
  - The amount of data that has been sent, but not yet acknowledged
  - A common mistake is to consider FlightSize=cwnd
- Congestion detection
  - Based on the assumption that a segment lost in the network is the result of congestion
  - A sender starts a timer (based on its RTT estimate) every time it transmits a segment
  - If an ACK from the destination is not received before the timeout, a loss is detected



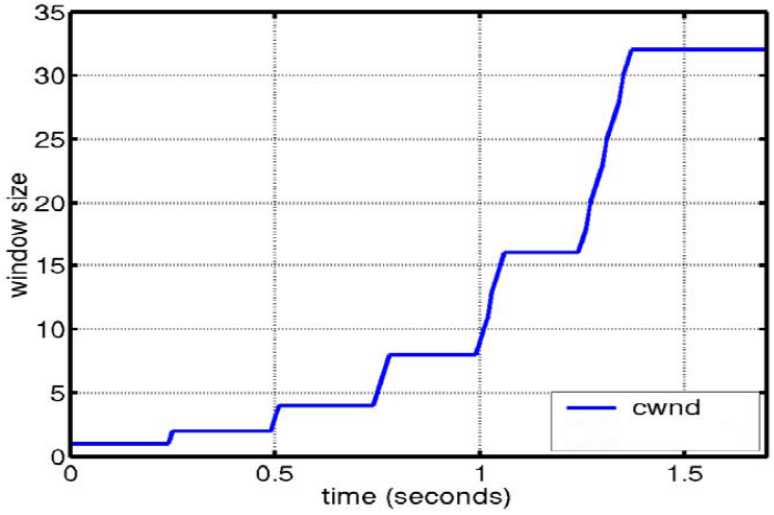
TCP

- TCP Congestion Control
  - Slow Start
  - Congestion Avoidance
  - Fast Retransmit
  - Fast Recovery
  - Selective Acknowledgements
- Standardized mechanisms
  - Original TCP – RFC 793
  - Additional mechanisms – RFC 1122 (clarifications & bug fixes), RFC 5681 (congestion control), RFC 2018 (SACK), etc.



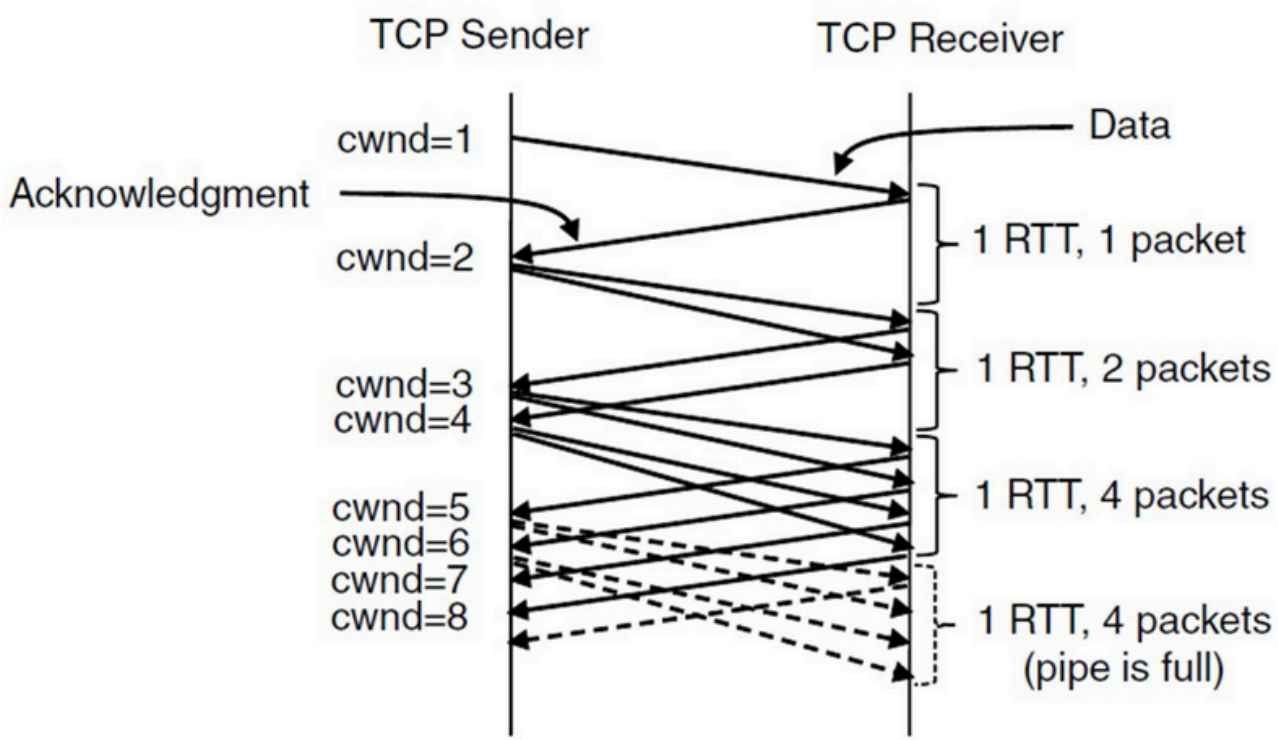
TCP

- Slow Start
  - Motivation: the end hosts do not know the state of the network at the beginning of their connection
  - Start with cwnd= 1
  - For every received ACK: cwnd= cwnd+ 1
  - Practically, cwnd doubles during an RTT interval
  - Despite its name, exponential increase of cwnd



TCP

- Slow Start



TCP

- Slow Start Threshold (ssthresh)
  - Important TCP parameter
  - Decides the moment when the host goes from Slow Start to Congestion Avoidance
  - Arbitrary initial value (usually very high)
  - ssthresh must follow the congestion level
  - After a lost segment (detected through a timeout or duplicate ACK): ssthresh= FlightSize/2
  - After the retransmission: cwnd= 1



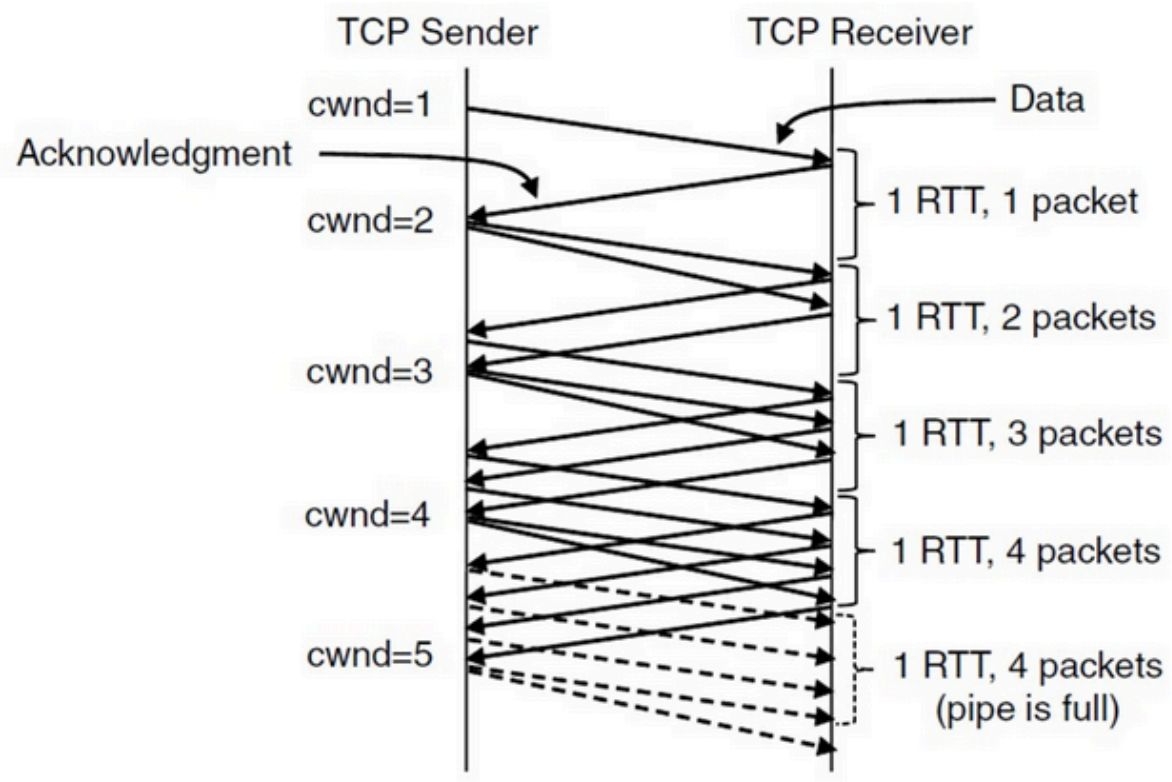
TCP

- Congestion Avoidance
  - Motivation: the exponential increase of Slow Start is too aggressive
  - Once a congestion has been detected, the transmitter tries to avoid reaching the congested state once again
  - A static approach can miss the opportunity of an increased throughput
  - The slow cwnd increase can delay the next congestion, while still testing for transmission opportunities



TCP

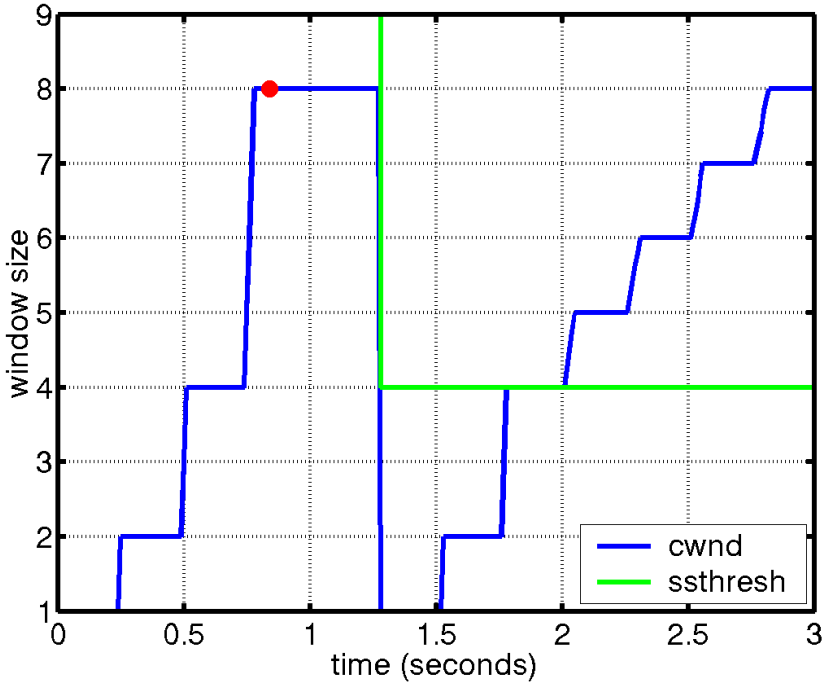
- Congestion Avoidance





TCP

- Congestion Avoidance
  - The TCP host enters in this mode when cwnd > ssthresh
  - cwnd= cwnd+ 1/cwnd
  - For each RTT: cwnd= cwnd+ 1



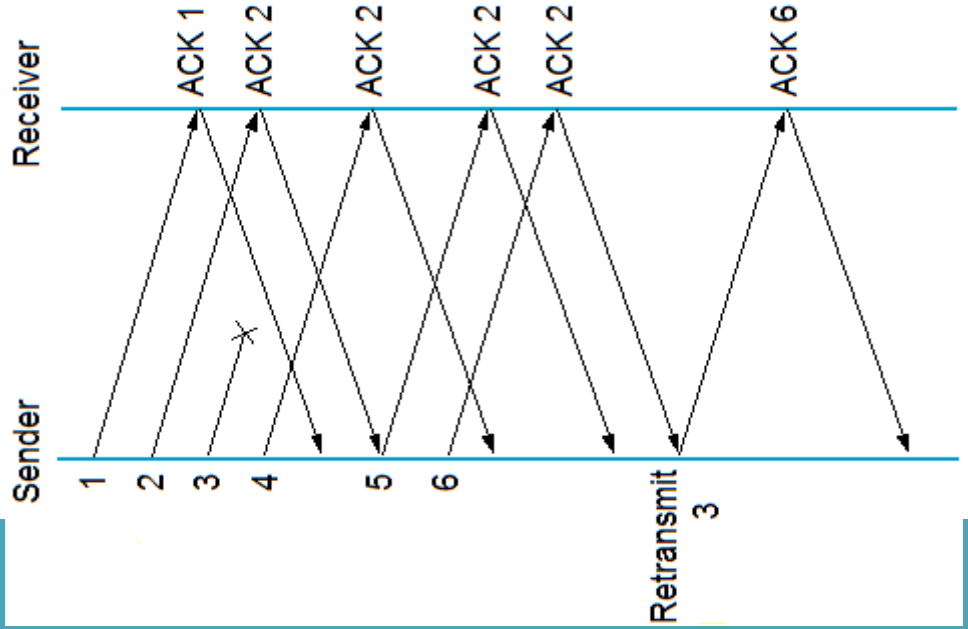
TCP

- Fast Retransmit
  - Lost packet -> retransmit after timeout
  - A timeout is a clear indication of network congestion, but can be very long
  - A duplicate ACK can have different reasons: congestion, segments following different paths, re-ordered ACKs
  - Considering a segment lost after the first duplicate ACK is too aggressive
  - TCP considers a segment lost after 3 duplicate ACKs (that means 4 consecutive ACKs of the same segment)



TCP

- Fast Retransmit
  - The usual operation mode
  - Retransmit lost message
  - Calculate FlightSize= min(rwnd,cwnd)
  - ssthresh= FlightSize/2
  - Enter Slow Start: cwnd= 1



TCP

- Fast Retransmit
  - This mechanism generally eliminates half of the TCP timeouts
  - This yields roughly a 20% increase in throughput
  - It does not work when the transmission window is too small to allow the reception of three duplicate ACKs



TCP

- Fast Recovery
  - The reception of duplicate ACKs also means that network connectivity exists, despite a lost segment
  - Entering Slow Start is not optimal in this case, as the congested state might have disappeared
  - The mechanism allows for higher throughput in case of moderate congestion
  - Complement of Fast Retransmit



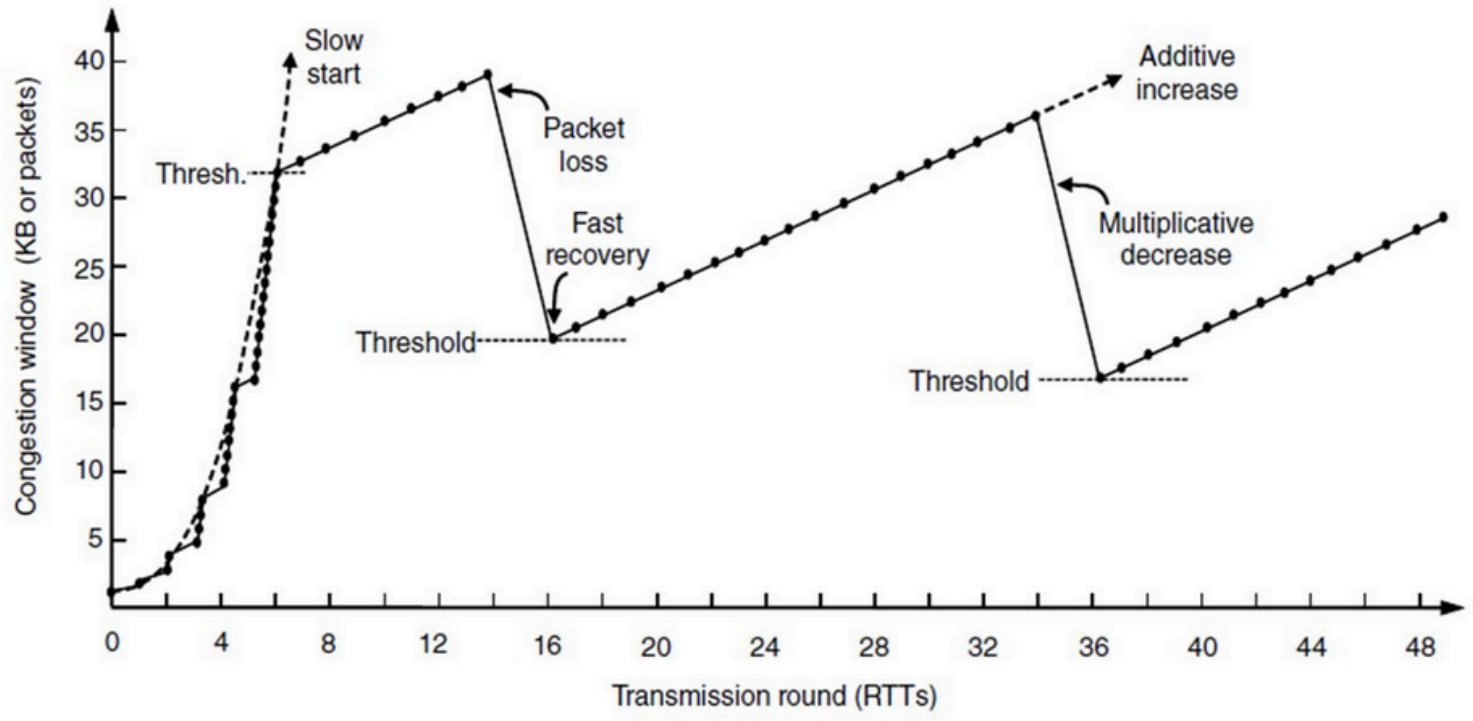
TCP

- Fast Recovery
  - Mode entered after 3 duplicate ACKs
  - As usual, set ssthresh= FlightSize/2
  - Retransmit lost packet
  - Window inflation: cwnd= ssthresh+ ndup (number of duplicate ACKs received)
  - This allows the transmission of new segments
  - Window deflation: after the reception of the missing ACK (one RTT later)
  - Skip Slow Start, enter Congestion Avoidance



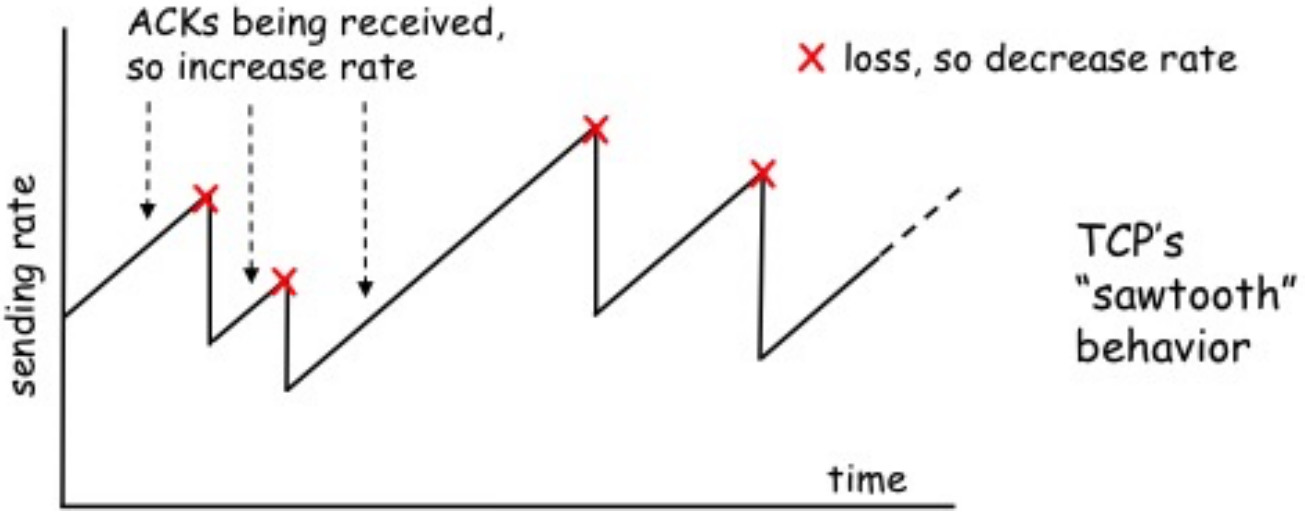
TCP

- TCP Reno: Slow Start, Congestion Avoidance, Fast Retransmit, Fast Recovery



TCP

- Typical TCP Saw-tooth Pattern





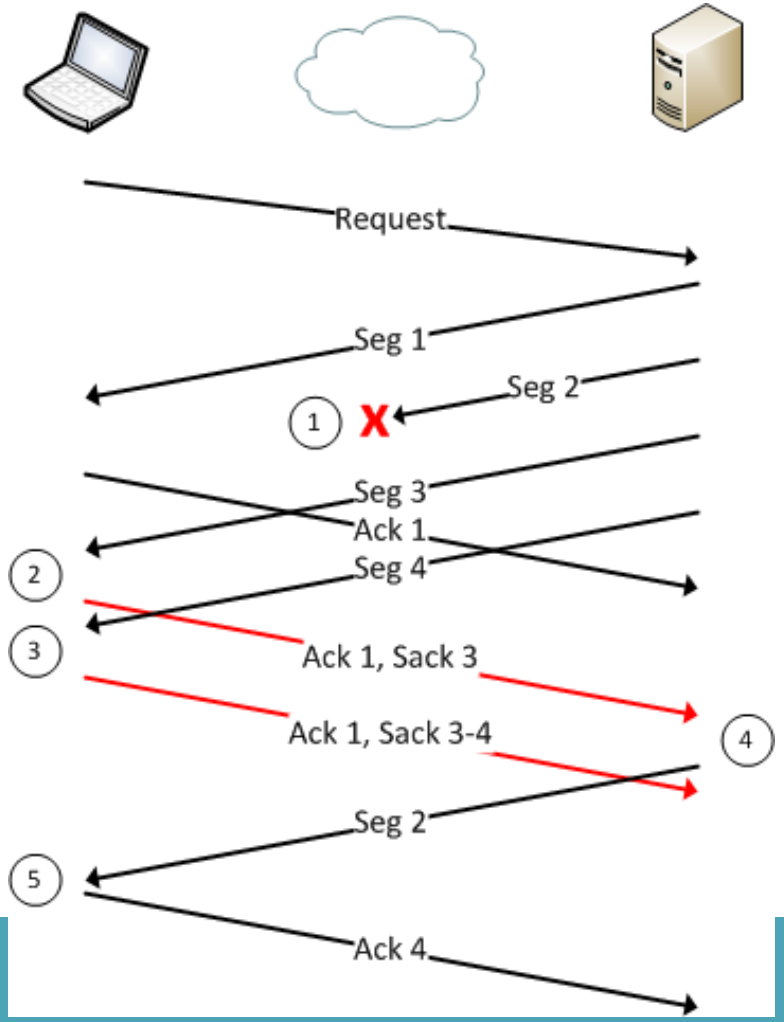
TCP

- Selective Acknowledgements
  - The receiver can only acknowledge contiguous segments
  - No ACK for segments correctly received after a lost segments
  - The sender has no feed-back regarding correctly received segments: retransmit or not?
  - Ideally, the sender should retransmit only the missing segments
  - With SACK, the receiver provides this feed-back to the sender



TCP

- Selective Acknowledgements



TCP

- Delayed ACK
  - RFC 1122
  - Problem:
    - Overhead induced by ACKs
  - Solution:
    - Delay an ACK by up to 500 ms
  - Reduce overhead by:
    - Combining multiple ACKs in one segment
    - Piggy-backing: data and ACK in the same segment
  - For a stream of full-sized incoming segments, an ACK is sent every second segment



TCP

- Nagle's algorithm
  - RFC 896
  - Problem:
    - Overhead induced by small packets
  - Solution:
    - Combine small outgoing data and send one single segment
  - If segment with **un-received ACK**, keep buffering output data until a **full size segment** can be sent
  - Downside:
    - Poor performance for applications that need fast network response
    - Poor interaction with delayed ACKs



TCP

- Clark's algorithm
  - RFC 896
  - Problem:
    - Silly window:
      - Receiver window full
      - Application reads 1 byte at a time
      - Send rcwd update
  - Solution:
    - Receiver waits before sending rcwd update (e.g., MSS free)
  - Complementary to Nagle's algorithm



TCP

- TCP Versions
  - TCP Tahoe: Slow Start, Congestion Avoidance, Fast Retransmit
  - TCP Reno: Fast Recovery
  - TCP New Reno: Modified Fast Recovery (window inflation)
  - Many other proposals exist: Vegas, Hybla, BIC, Westwood, …



TCP

- TCP Versions
  - TCP Tahoe: Slow Start, Congestion Avoidance, Fast Retransmit
  - TCP Reno: Fast Recovery
  - TCP New Reno: Modified Fast Recovery (window inflation)
  - Many other proposals exist: Vegas, Hybla, BIC, Westwood, …



What would be better?

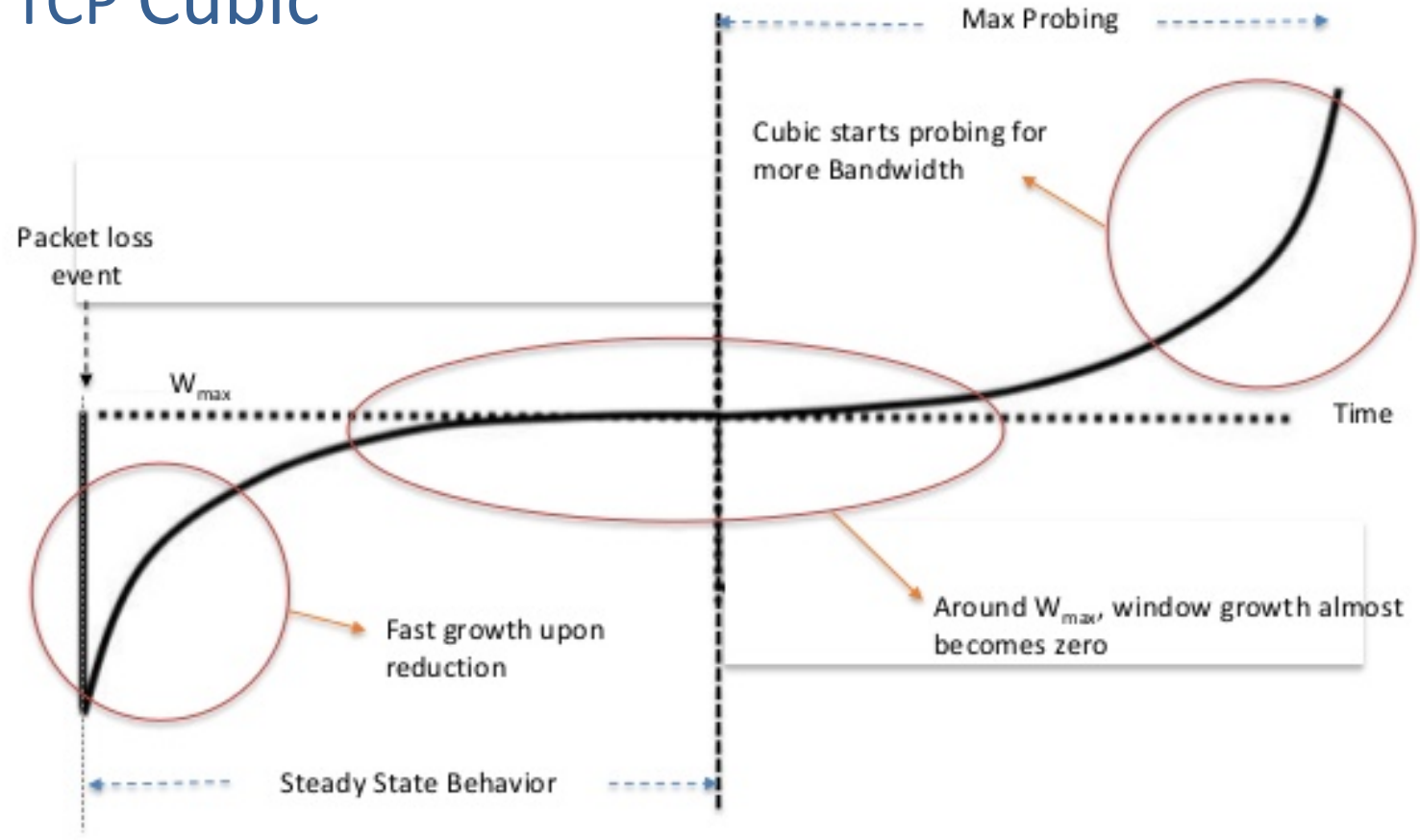
TCP

- TCP Cubic
  - Current state of the art - RFC 8312 (2018)
  - Window size no longer controlled by received ACKs
  - cwnd computed as a cubic function of time since the last congestion
  - Three phases:
    - aggressive increase until ssthresh (similar to slow start)
    - slow probing for higher window
    - aggressive probing for higher window



TCP

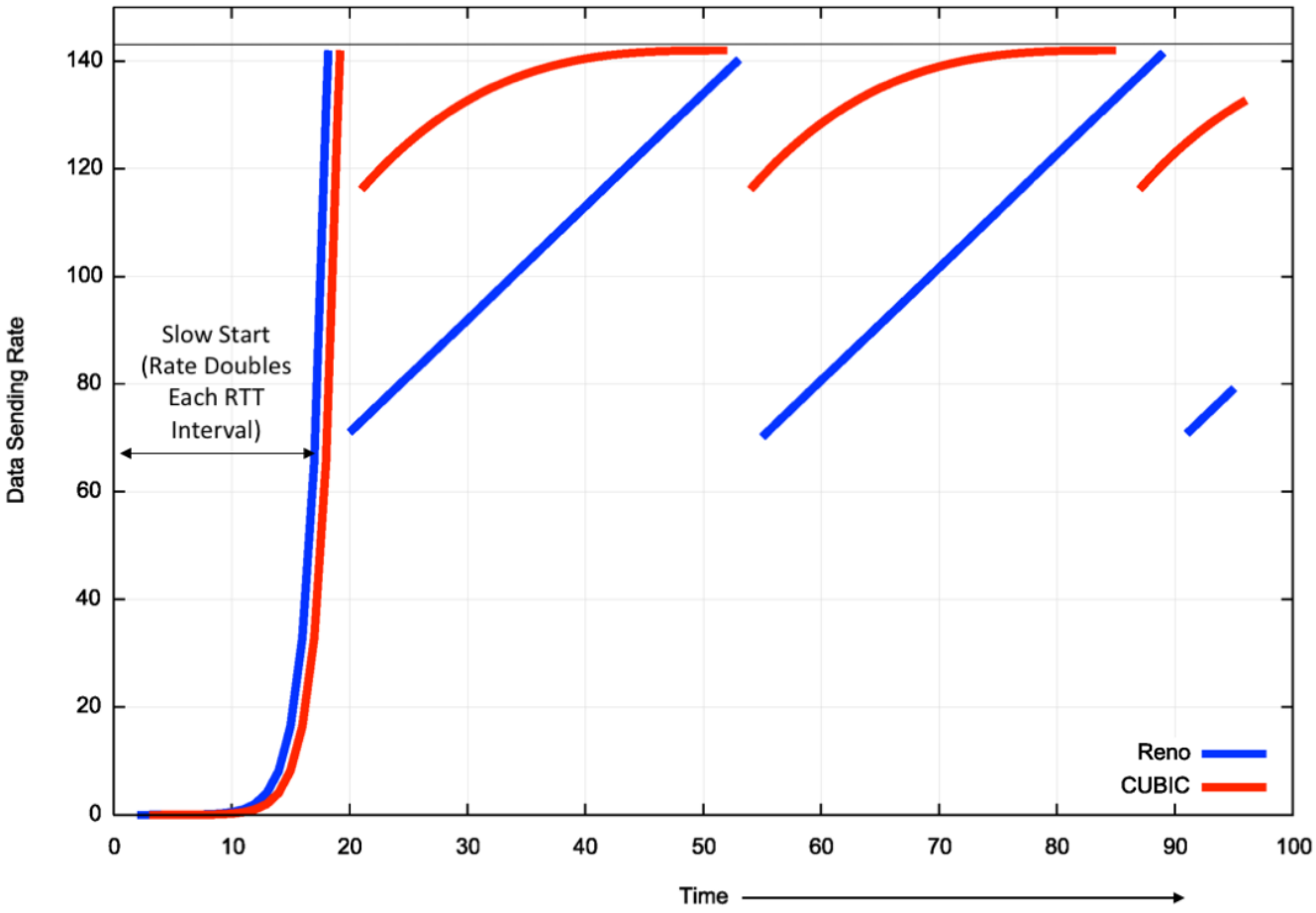
- TCP Cubic



© S. Ha, I. Rhee, and L. Xu, "CUBIC: A New TCP - Friendly High-Speed TCP Variant, ACM SIGOPS, 2008"

TCP

- TCP Cubic



TCP

- TCP Cubic
- Advantages:
  - Real-time dependent (since it's not based on ACKs)
    => Fairness among flows
  - Scalability and stability
    - Fast increase to Wmax and longer stay there
  - Drawbacks:
    - The speed to react (if saturation point has increased far beyond the last one)



TCP

Beyond the End-to-End Argument

- The way routers decide to drop packets impacts the functioning of TCP
- Advanced techniques can be implemented inside the network

Random Early Detection

- RED manages router queues and drops packets **randomly** based on a queue threshold

  => Faster senders will see a packet drop

- Only the affected TCP senders will enter Slow Start or Congestion Avoidance, slowing the network down before the actual congestion

TCP

- Explicit Congestion Notification
  - ECN is also based on a queue threshold parameter
  - As opposed to RED, ECN only marks packets instead of dropping them
  - Routers mark 2 bits in the IP header (Type of Service field) to signal whether congestion is occurring
  - Through cross-layer mechanisms, TCP can learn this information and reduce the congestion window
  - ECN avoids packet drops and reduces the delay created by retransmissions



TCP

- Explicit Congestion Notification
  - ECN is also based on a queue threshold parameter
  - As opposed to RED, ECN only marks packets instead of dropping them
  - Routers mark 2 bits in the IP header (Type of Service field) to signal whether congestion is occurring
  - Through cross-layer mechanisms, TCP can learn this information and reduce the congestion window
  - ECN avoids packet drops and reduces the delay created by retransmissions

How does TCP react?



TCP

- QUIC – Quick UDP Internet Connections
  - draft-ietf-quic-transport-23
  - User space implementation of a transport protocol
  - Released by Google in 2013
  - From January 2017, implemented in the Chrome browser and the Google Search and You Tube applications
  - Currently transports between 5% and 10% of the Internet traffic
  - 5% reduction in search time
  - 15% reduction in video rebuffering



TCP

- QUIC – Quick UDP Internet Connections
  - draft-ietf-quic-transport-23
  - User space implementation of a transport protocol
  - Released by Google in 2013
  - From January 2017, implemented in the Chrome browser and the Google Search and You Tube applications
  - Currently transports between 5% and 10% of the Internet traffic
  - 5% reduction in search time
  - 15% reduction in video rebuffering

HOW?



TCP

- QUIC – Goals
  - Reduce latency - make the Web faster!
  - Rapid experimentation
    - TCP modifications: 5-15 years to be implemented
    - QUIC: just a few weeks : Update Chrome + Server (no kernel involved!)
  - Open source development in Chromium

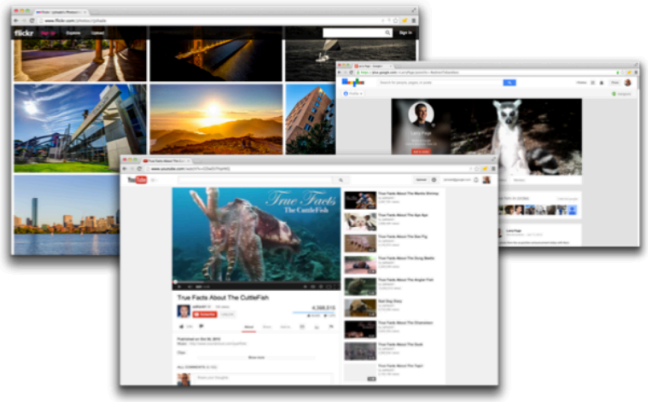


TCP

- QUIC – Motivations -> Web

Google

Google Developers Live: QUIC



1990

single, static page
one resource
one domain

2014

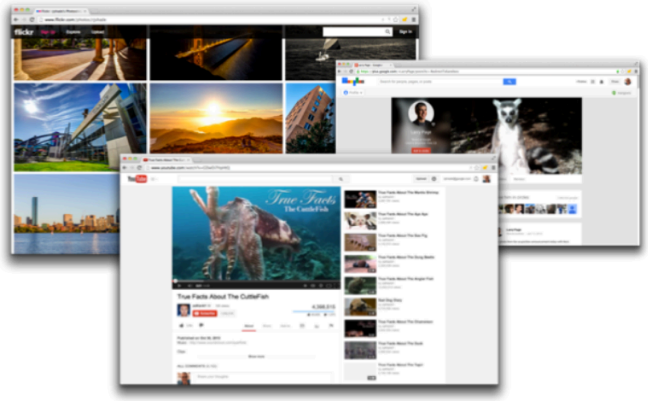
1,200 KB
80 resources
30 domains

data from httparchive.org

- QUIC – Motivations -> Web

Google

Google Developers Live: QUIC



1990

single, static page
one resource
one domain

2014

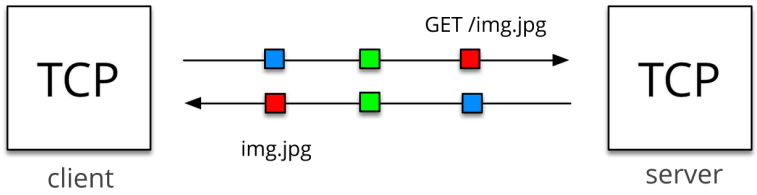
1,200 KB
80 resources
30 domains

Individual HTTP Request for each resource?

data from httparchive.org

TCP

- QUIC – Motivations -> Web



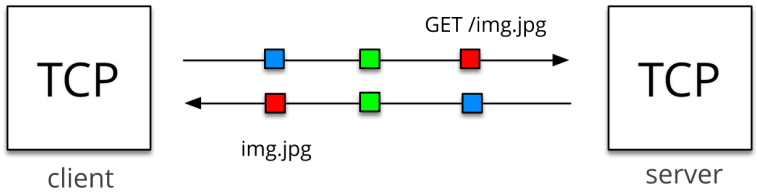
HTTP pipelining



Google Developers Live: QUIC

TCP

- QUIC – Motivations -> Web



HTTP pipelining

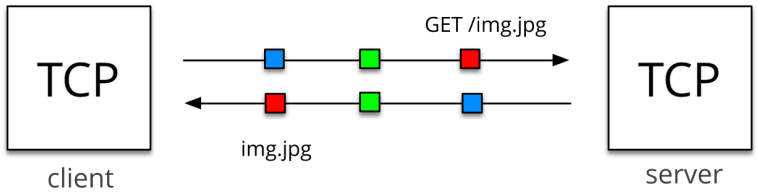
What happens if the first request takes a long time to process?



Google Developers Live: QUIC

TCP

- QUIC – Motivations -> Web



HTTP pipelining

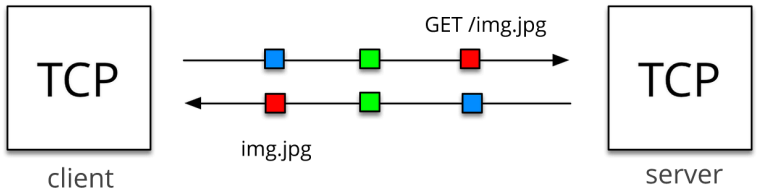
What happens if the first request takes a long time to process?

Head of line blocking!

Google Developers Live: QUIC

TCP

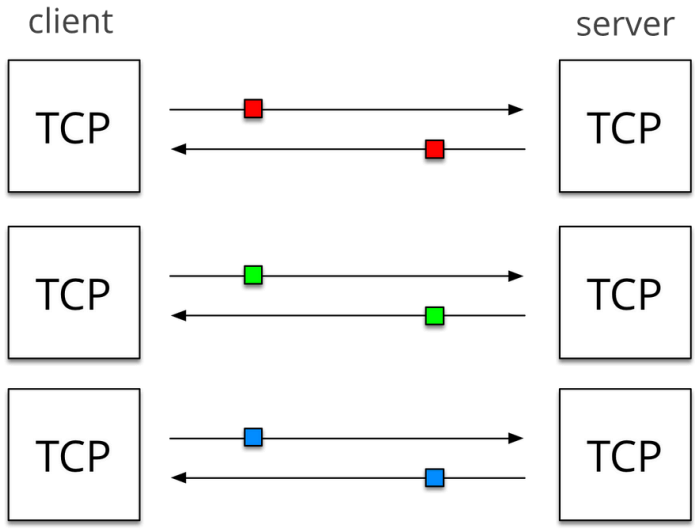
- QUIC – Motivations -> Web



HTTP pipelining

What happens if the first request takes a long time to process?

Head of line blocking!

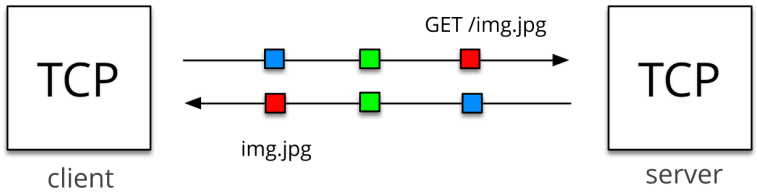


Multiple TCP connections

Google Developers Live: QUIC

TCP

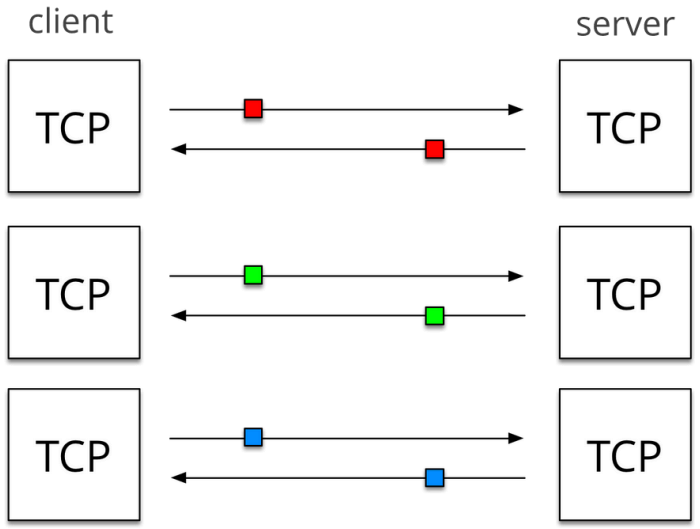
- QUIC – Motivations -> Web



HTTP pipelining

What happens if the first request takes a long time to process?

Head of line blocking!



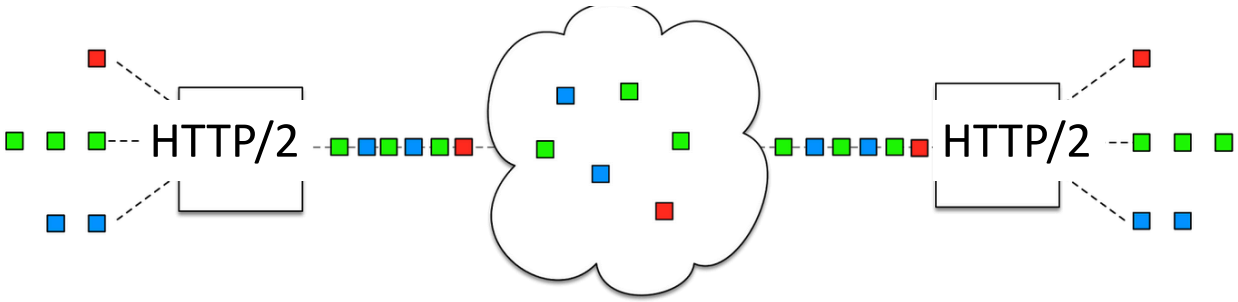
Multiple TCP connections

How many?

Google Developers Live: QUIC

TCP

- QUIC – Motivations -> Web



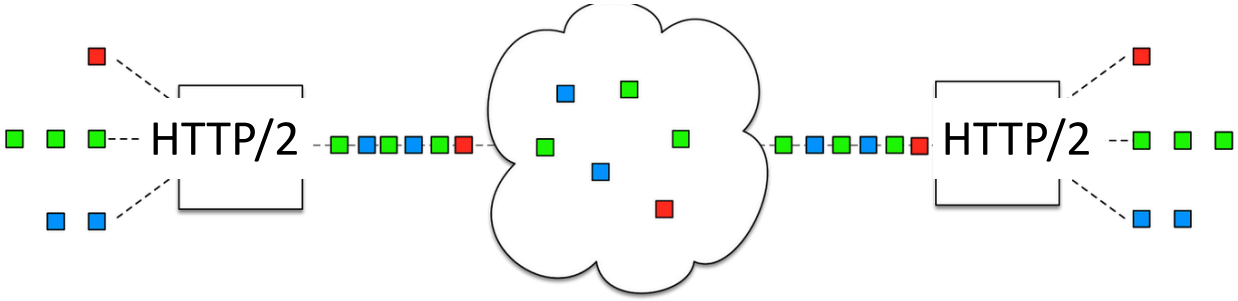
HTTP/2 multiplexing over TCP



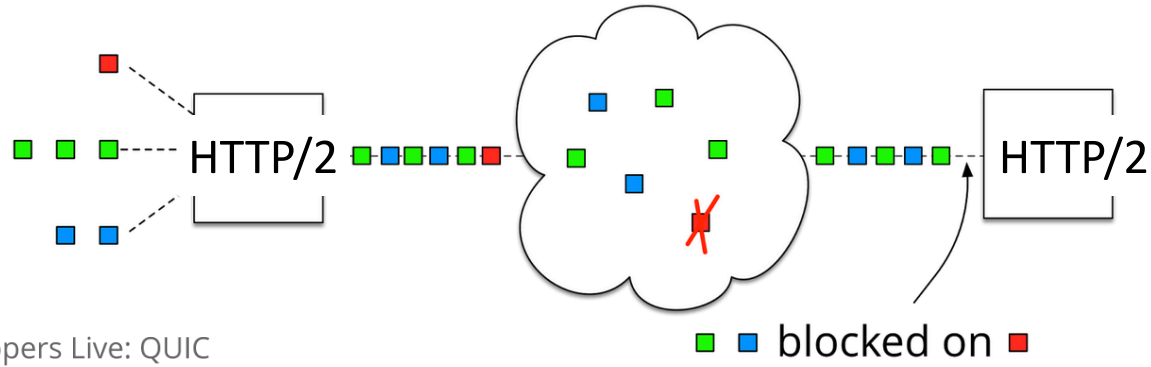
Google Developers Live: QUIC

TCP

- QUIC – Motivations -> Web



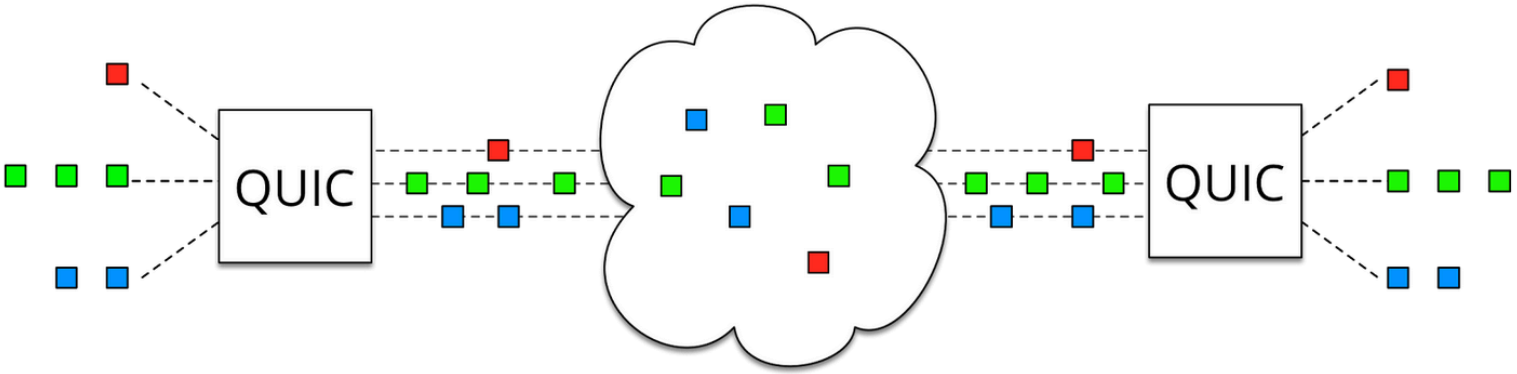
HTTP/2 multiplexing over TCP



Google Developers Live: QUIC

TCP

- QUIC – Motivations -> Web

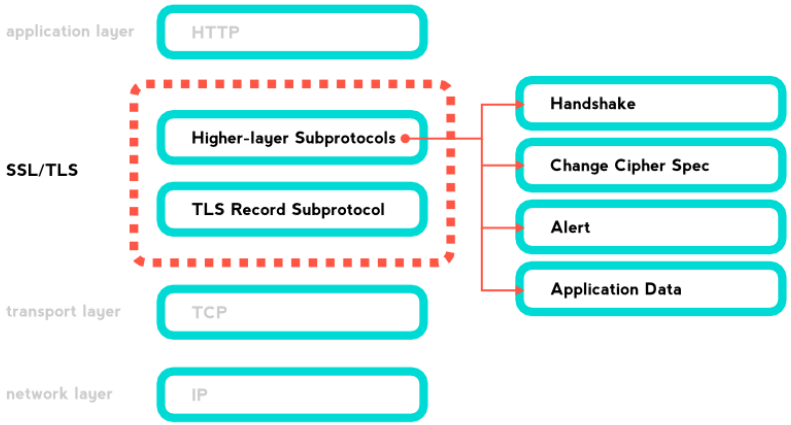


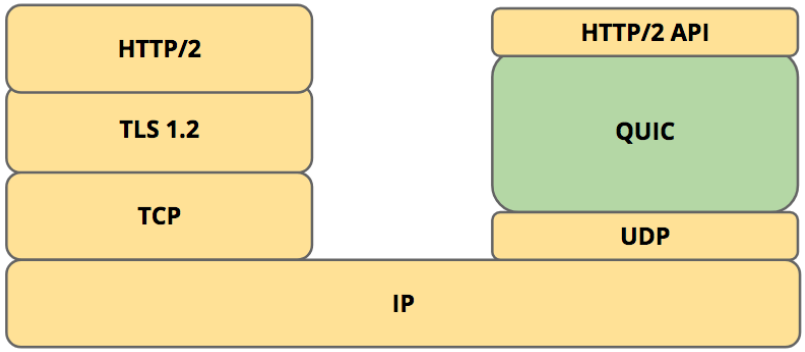
QUIC: multiplexing over UDP

Google Developers Live: QUIC

TCP

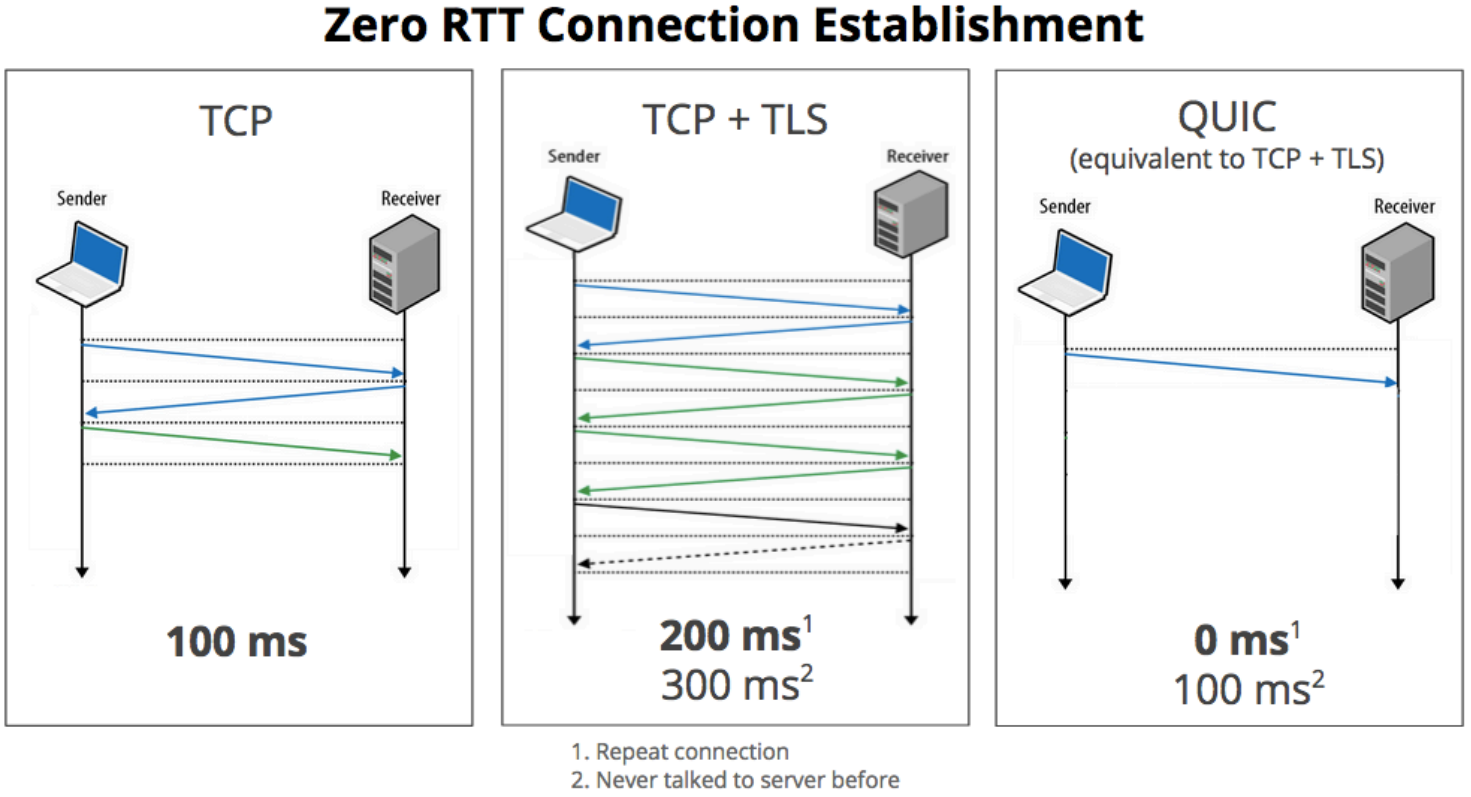
- QUIC – Motivations
  - Classic functioning: HTTP/2 – TLS – TCP
  - Transport Layer Security (TLS) – extra overhead
  - Data transmitted after 2xRTT





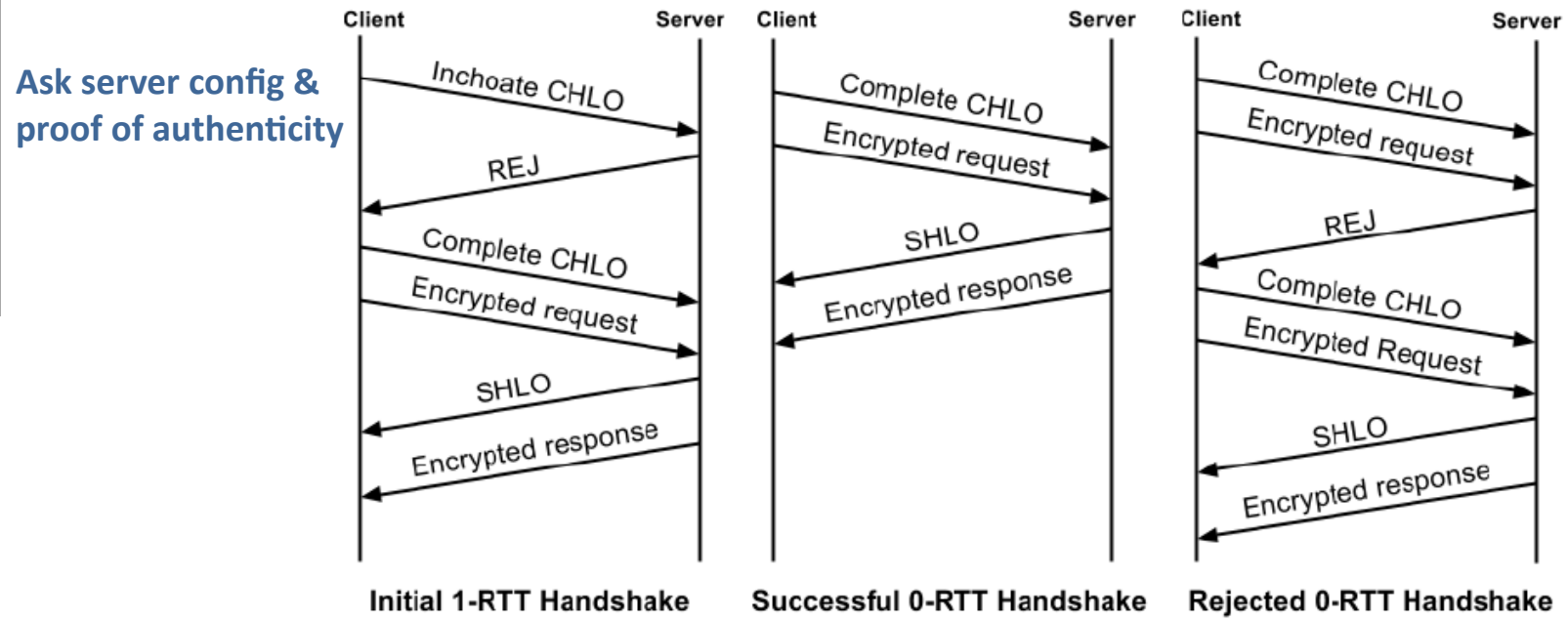
TCP

- QUIC – Motivations



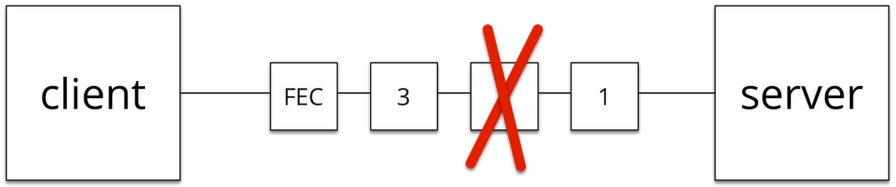
TCP

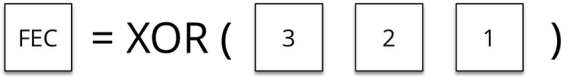
- QUIC – Principles
  - Save the context of already known servers/clients
  - Results in 0-RTT connection in 85% of the cases



- QUIC – Principles

Forward Error Correction





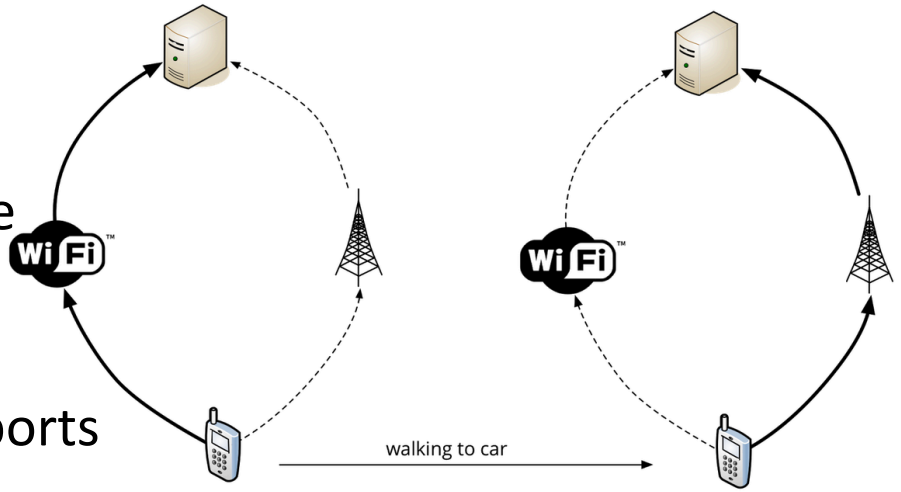
Google Developers Live: QUIC

TCP

- QUIC – Principles

“Parking lot problem”

Long-lived TCP connection:
- Big congestion window
- No handshake needed anymore



TCP - defined by IP addresses & ports
QUIC - unique identifier (64 bits)
=> seamless transition between networks

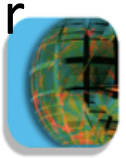
Google Developers Live: QUIC

TCP

- QUIC – Congestion Control

  - Incorporates TCP best practices
    - TCP Cubic
    - Fast Retransmit, etc.
  - Better signaling than TCP
    - Retransmitted packets consume new sequence number
      - No retransmission ambiguity
      - Prevents loss of retransmission from causing RTO
    - More verbose ACK (256 NACK ranges vs. 3 SACK ranges)





Google Developers Live: QUIC

TCP

- QUIC – Sum-up
  - Multiple streams transmitted over the same connection (similar mechanisms in TCP)
  - Streams are controlled both independently (flow controlled) and per connection
  - A large part of the transport layer information is encrypted
  - Unique identifier, even for retransmissions, easing RTT estimation
  - TCP congestion control mechanisms
  - Chromium's network stack opens both a QUIC and traditional TCP connection at the same time, which allows it to fallback with zero latency
  - More details: IETF drafts, research papers, Google developers live: https://www.youtube.com/watch?v=hQZ-0mXFmk8



TCP

- QUIC – Problems
  - Not always better performance than TCP (e.g. when a lot of packets are delivered in disorder, “perfect conditions”, mobile)
  - 2x CPU consumption compared with TCP
  - Some middle-boxes block UDP
  - TCP unfriendly



TCP

- Back to TCP: Implementation
  - At the transport layer, an active application is identified by the 5-tuple:

    (protocol, $@IP_{source}$, $Port_{source}$, $@IP_{dest}$, $Port_{dest}$)

  - A client needs to know $@IP_{server}$ and $Port_{server}$ in order to send a connection request

  - The $Port_{client}$ can be allocated dynamically by the operating system



TCP

- Socket
  - Application programming interface (API) for communication between processes
  - When processes are run on different machines, a socket becomes the basis of network communications
  - Support for both TCP and UDP
  - Bidirectional communication using functions such as *read()/write()* or *send()/recv()*
  - A socket is represented as a file handler in Unix systems



TCP

- Socket API
  - A series of libraries
    - sys/socket.h – core socket functions and data structures
    - netinet/in.h – IP, TCP and UDP data structures
    - sys/un.h – data structures for local communications
    - arpa/inet.h – functions for manipulating IP addresses
    - netdb.h – functions for translating protocol and host names



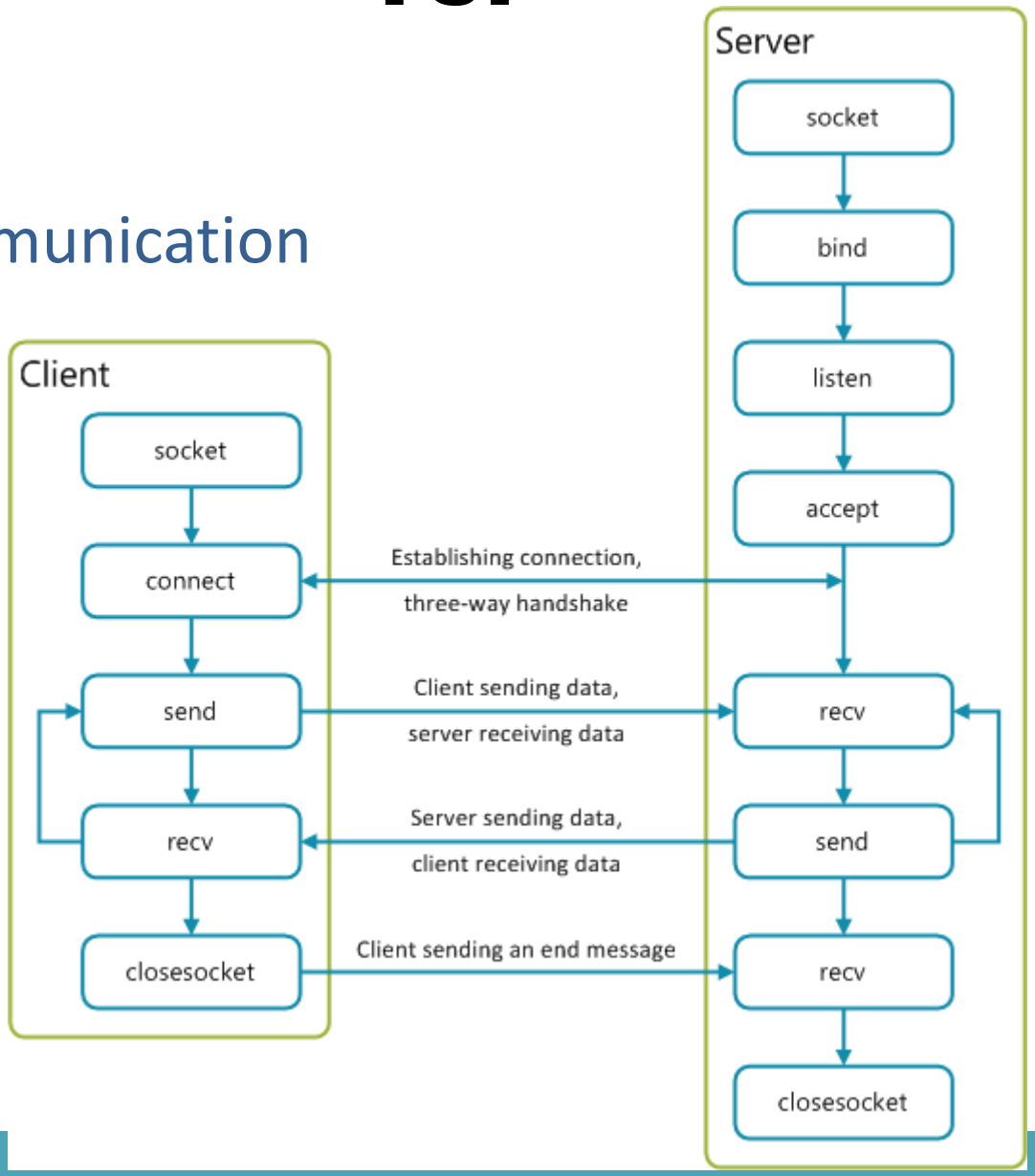
TCP

- Socket API
  - A series of functions
    - socket()
    - bind()
    - listen()
    - connect()
    - accept()
    - send() / write()
    - recv() / read()
    - close()
    - select()
    - poll()



TCP

- TCP communication



TCP

- UDP communication