Performance of Networked Systems

Lecture 5: Performance of Transport-Layer Protocols

Overview of today's lecture

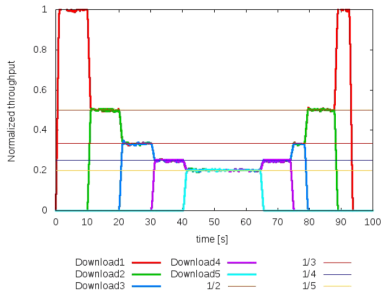
1. Recap of last week's lecture on Traffic Management in IP
2. Recap on latency and bandwidth
3. Performance at the transport layer
   - TCP Slow Start, Flow Control, Congestion Control, Congestion Avoidance, Fast Recovery and Fast Retransmit
5. Mathis' Square Root formula, PFTK Formula

Background reading material:

1. The book by Grigorik, chapters 2 and 3.
2. J. Padhye et al. (2000). Modelling TCP Reno Performance: a simple model and its empirical validation

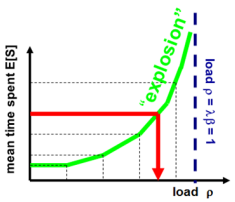
Wrap Up of Previous Lecture

Elastic traffic

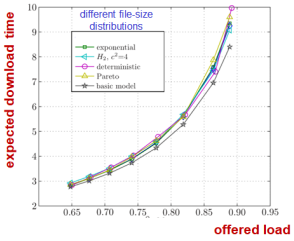


Processor Sharing models

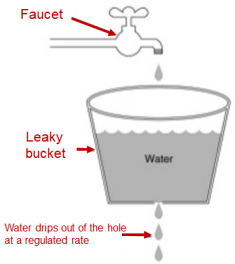
basic model (theory)



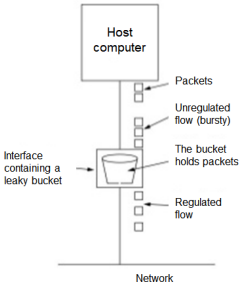
practice (lab setup)



Leaky buckets

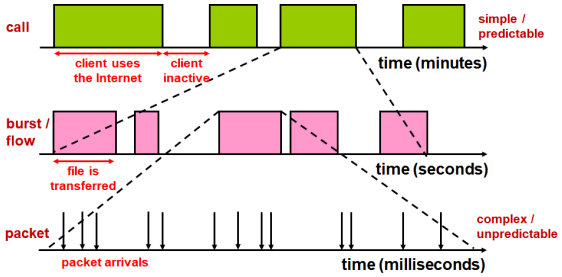


Leaky Bucket with water

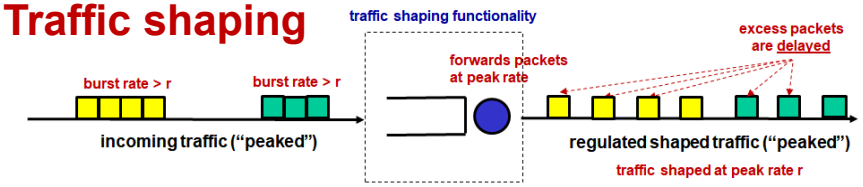


Leaky Bucket with packets

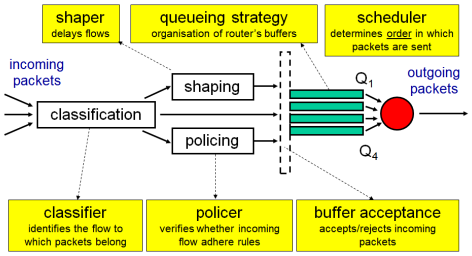
Time scales



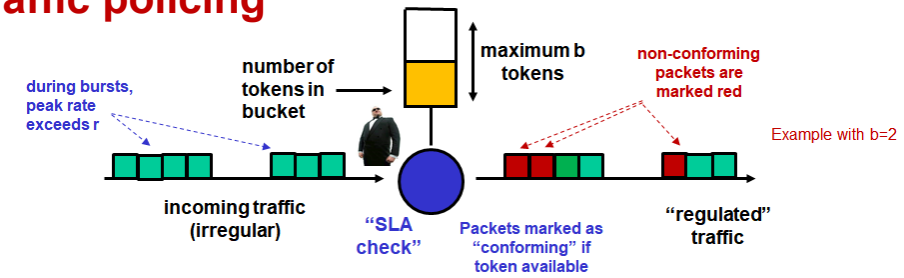
Traffic shaping



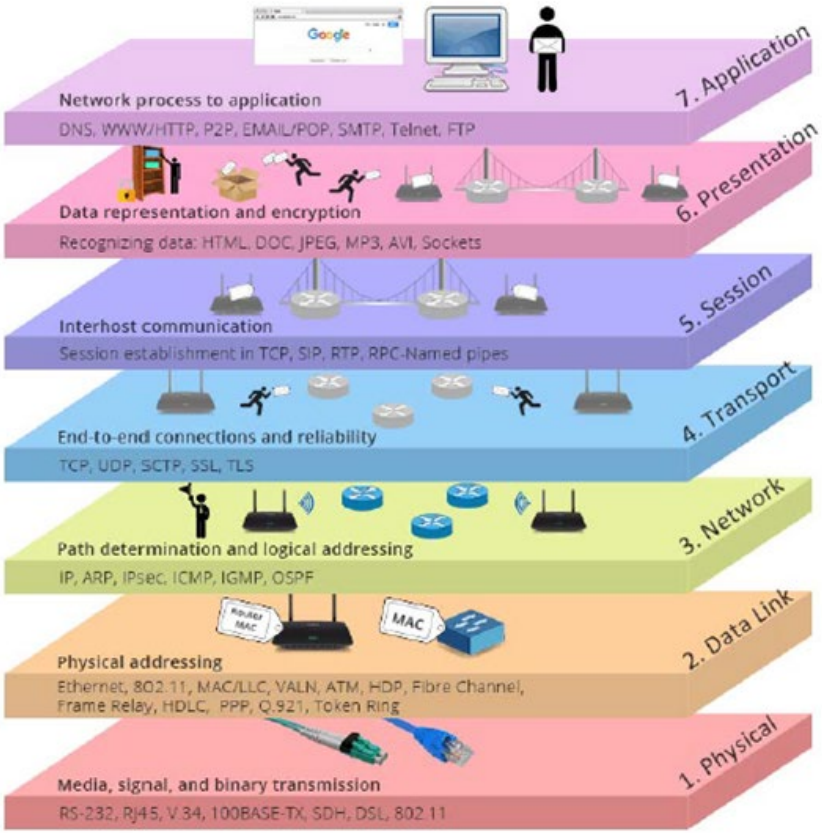
TM functionalities



Traffic policing



Course Structure



HTTP, FTP, MM applications

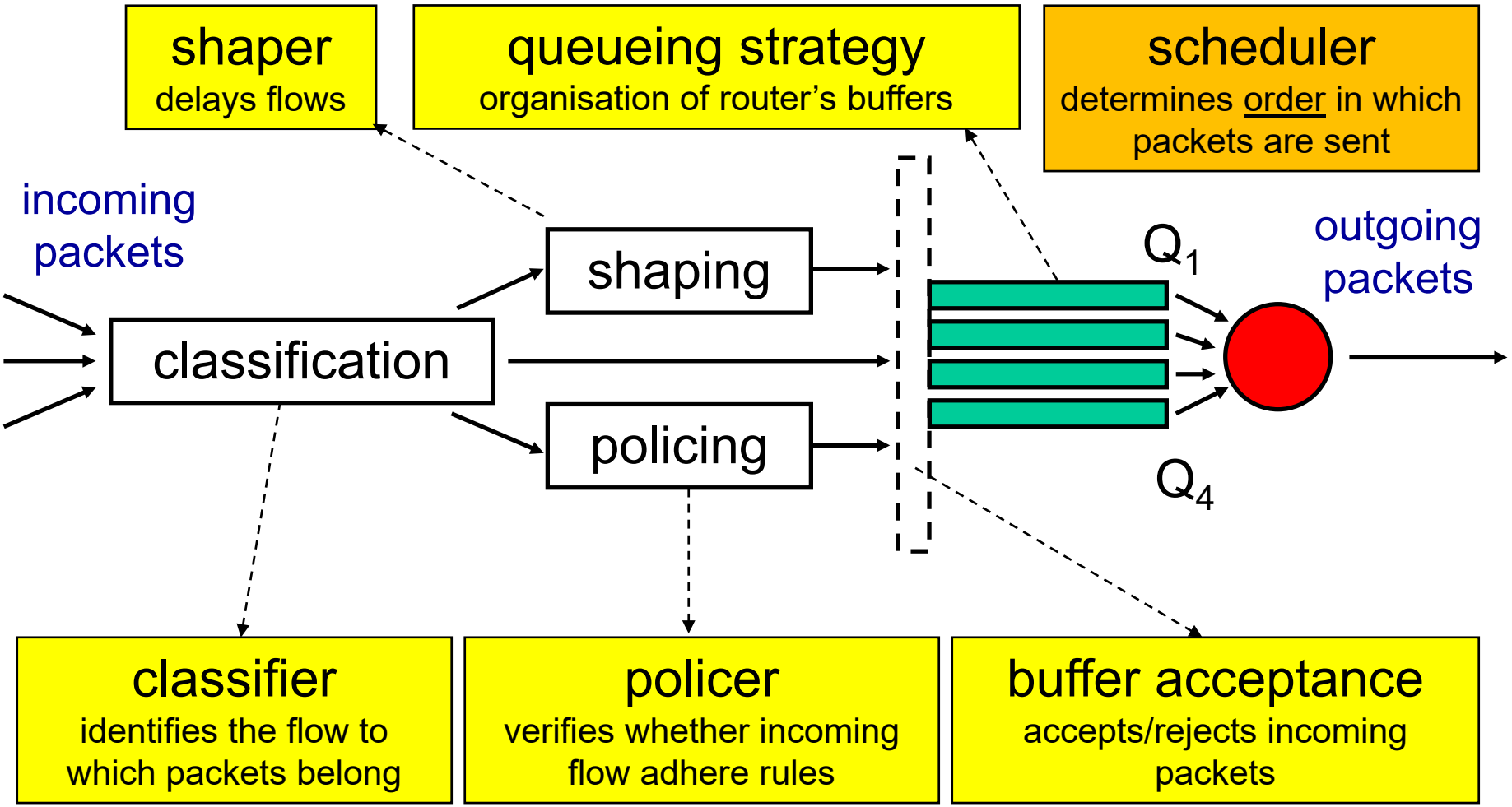
Transport Control ← today

Traffic Management last week

Medium Access next week's lectures

Idea: understand both **technology** and **theory**

QoS Functionalities of Routers

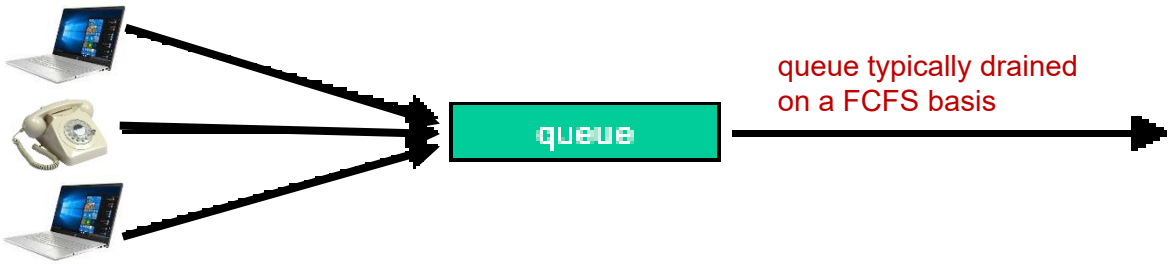


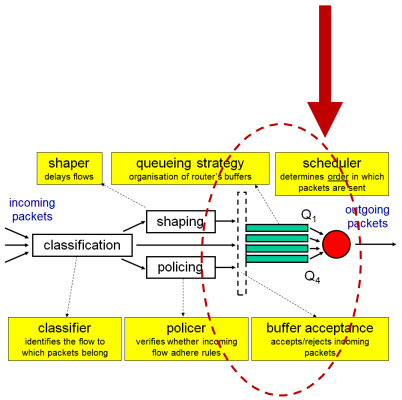
Idea: Routers can implements all kinds of QoS mechanisms

Queue Management

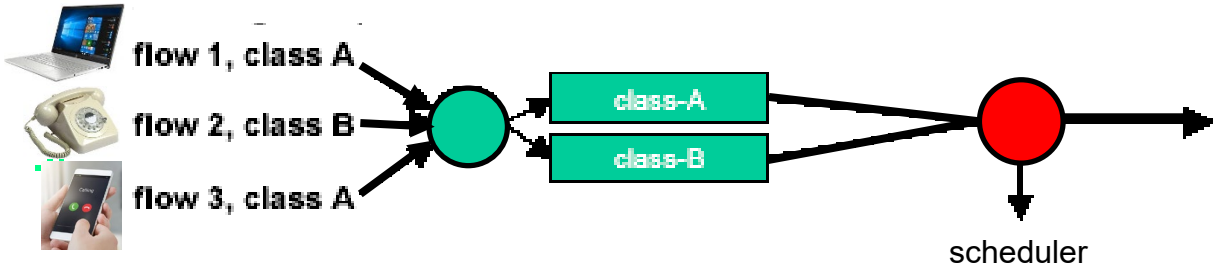
Many options to divide the buffer space in logical queues

1. One logical queue for all flows together



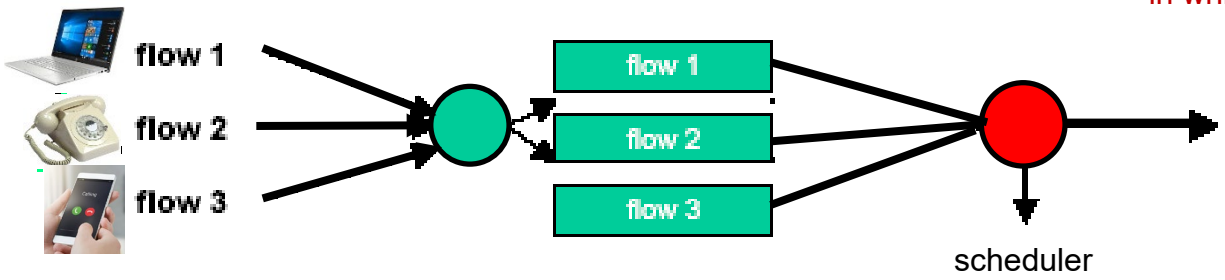


2. One logical queue for one class of flows

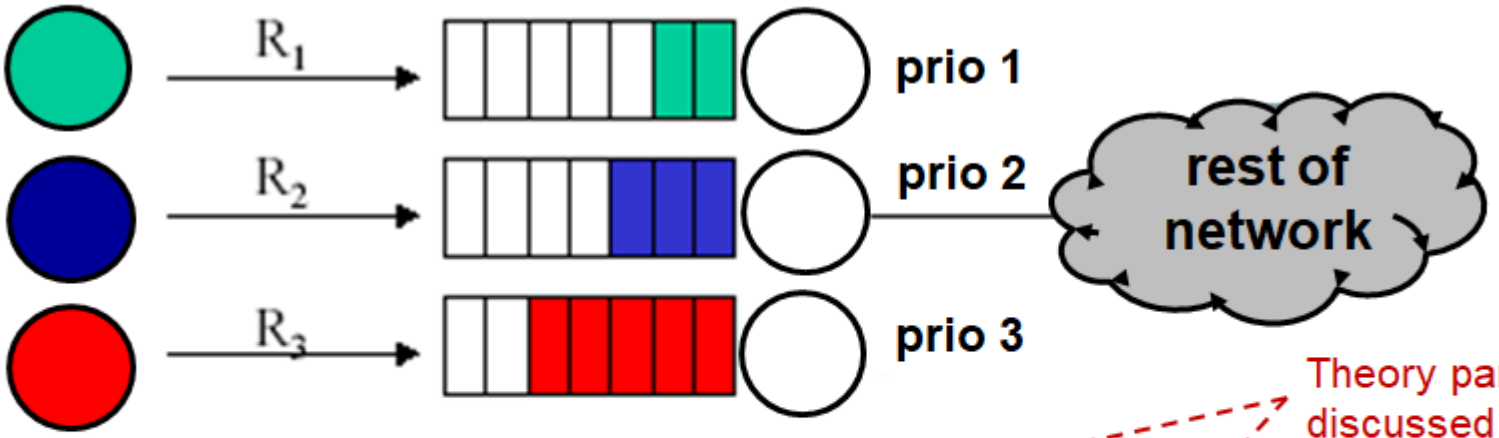


3. One logical queue for each individual flow

scheduler determines the order in which the queues are served



Priority Scheduling



Theory part to be discussed in later lecture

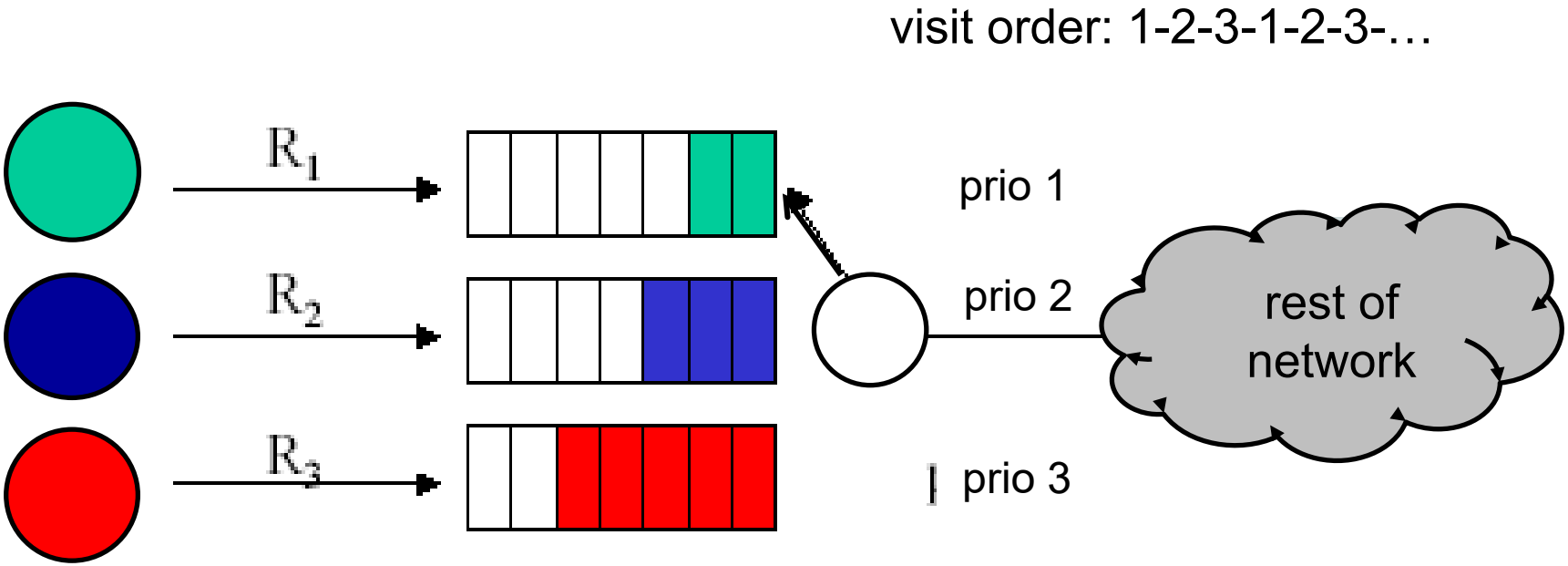
average waiting times

$$E[W_1] = \frac{\lambda E[X^2]}{1-\rho_1}$$

$$E[W_2] = \frac{\lambda E[X^2]}{(1-\rho_1)(1-\rho_1-\rho_2)}$$

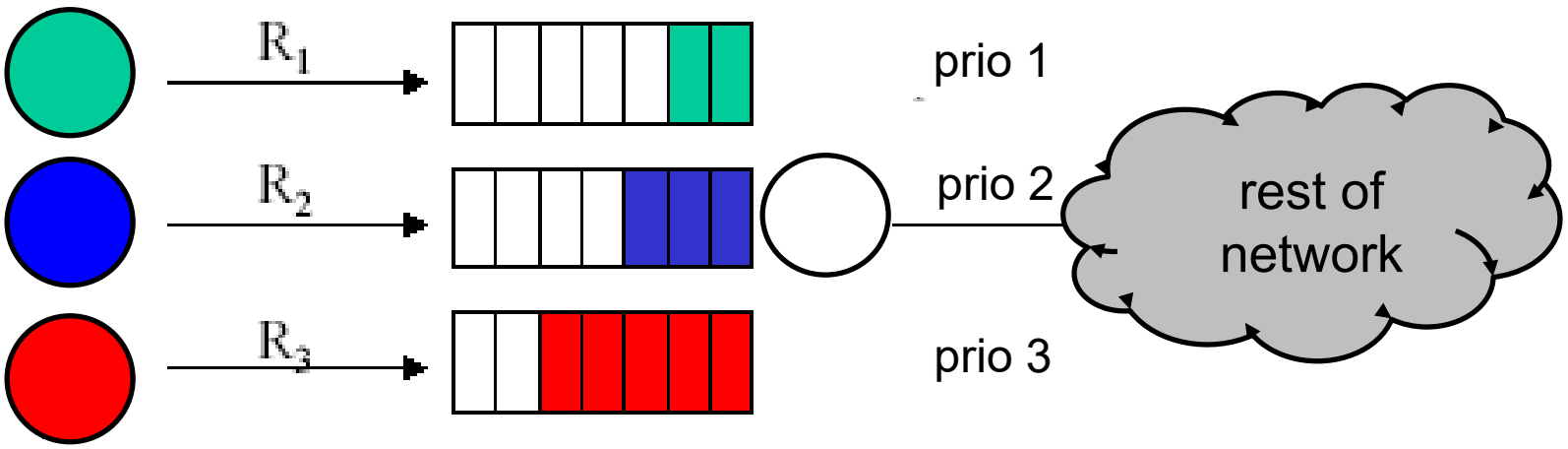
- **Priority scheduling**
  - multiple queues, served in order of priority
  - green packets high priority, blue medium, red low
  - risk of starvation for low-priority queues

Round-Robin Scheduling



- **Round Robin scheduling**
  - multiple queues, served in **cyclic** order: 1-2-3-1-2-3-…
  - ultimately fair, but may not be good for well-paying high-priority customers (inefficient)

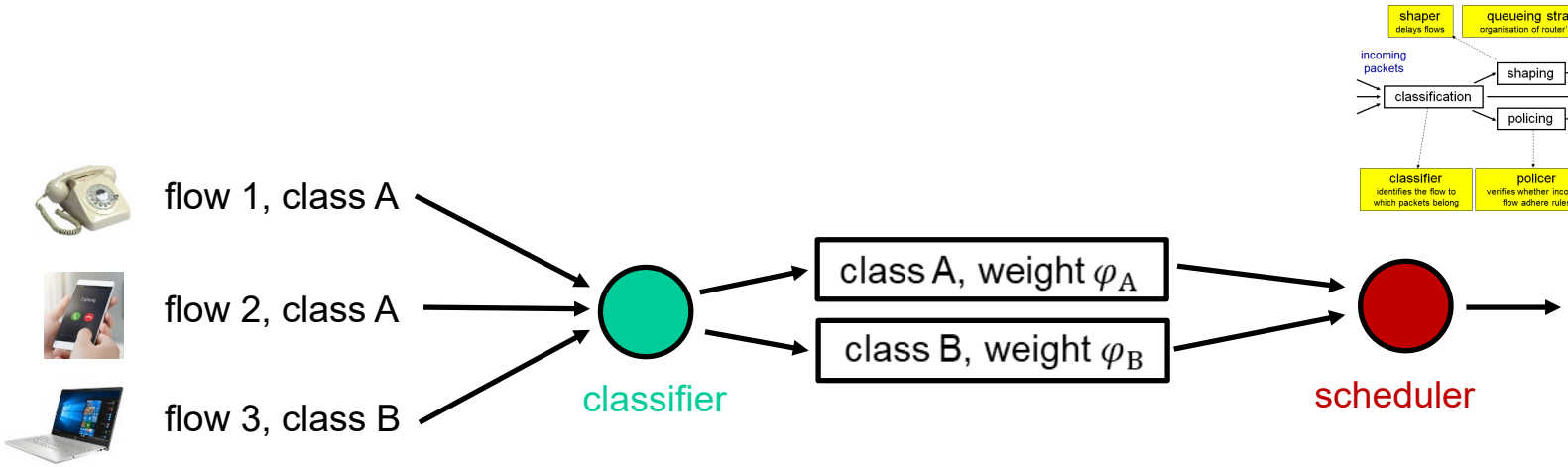
Weighted Fair Queueing



Example visit order: 1-1-1-2-2-3-…

- **Weighted Fair Queueing (WFQ) scheduling**
  - associate “weight” with each class, which determines the relative number of service units in each cycle

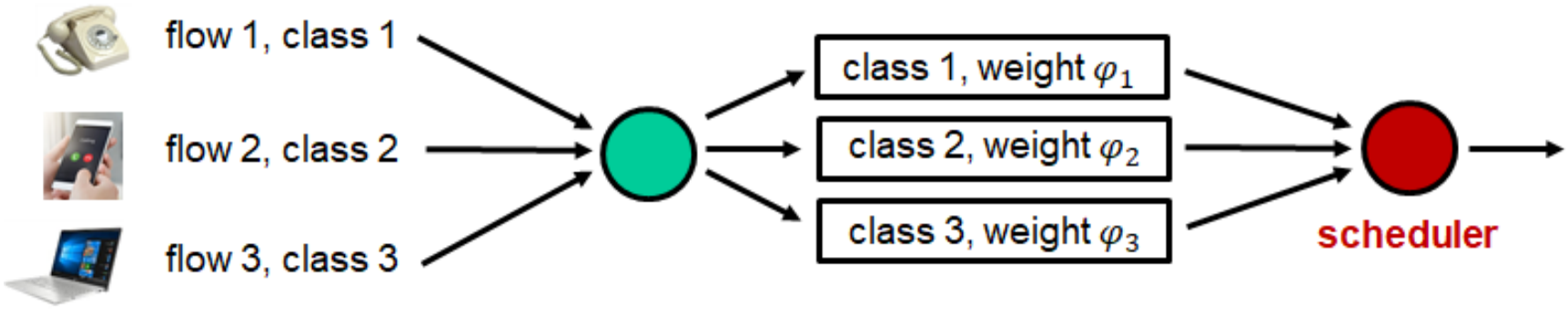
Generalized Processor Sharing



- **Generalized Processor Sharing scheduling**
  - "idealized" work-conserving scheduler
  - traffic flows are approximated as "fluid" flows
  - each flow **class** *k* has weight φ_k, and receives **at least fraction** φ_k of available bandwidth (relative priority guarantee)
  - within each class k, bandwidth is equally shared among all the flows in class k (in a Processor Sharing fashion)

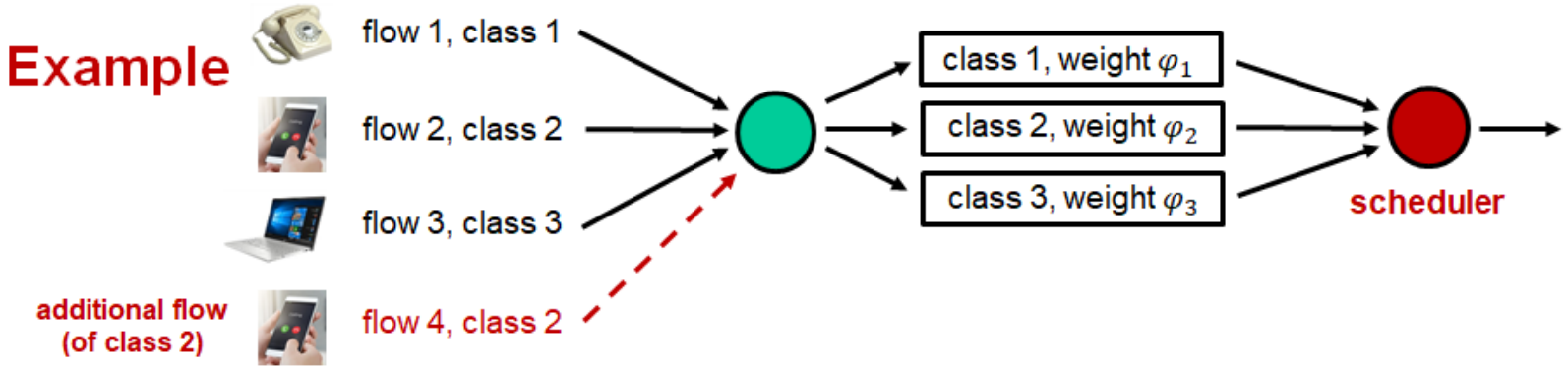
Generalized Processor Sharing

Example



class *k*	weight φ_k	# of flows in class *k*	Minimum fraction of BW for <u>each</u> flow in class *k*	Minimum fraction of BW for class *k* in total
1	2	1	2/(1*2+1*3+1*5) = 2/10	2/10
2	3	1	3/(1*2+1*3+1*5) = 3/10	3/10
3	5	1	5/(1*2+1*3+1*5) = 5/10	5/10

Generalized Processor Sharing

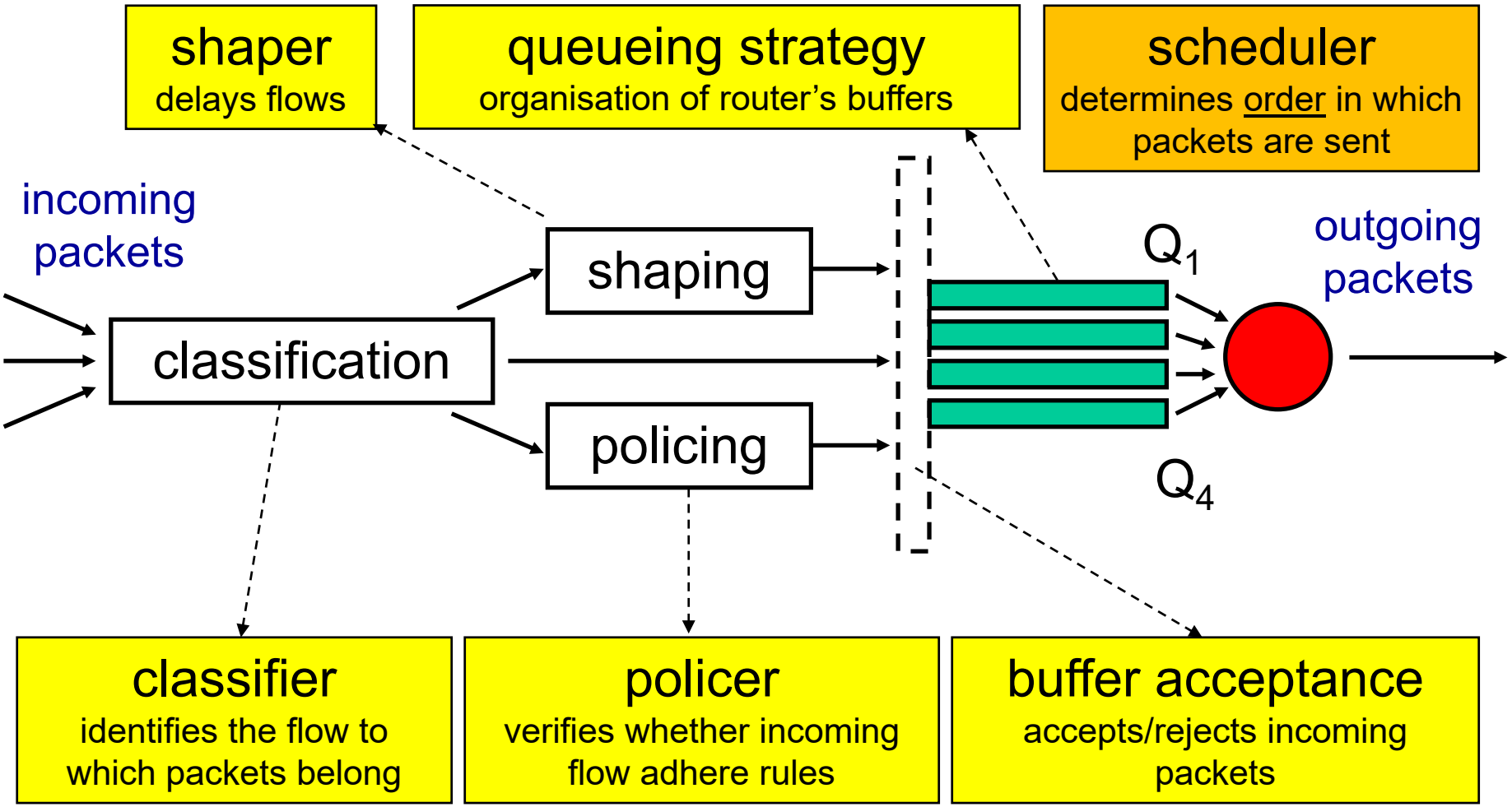


class *k*	weight φ_k	# of flows in class *k*	Minimum fraction of BW for <u>each</u> flow in class *k*	Minimum fraction of BW for class *k* <u>in total</u>
1	2	1	2/(1*2+1*3+1*5) = 2/10	2/10
2	3	1	3/(1*2+1*3+1*5) = 3/10	3/10
3	5	1	5/(1*2+1*3+1*5) = 5/10	5/10

class *k*	weight φ_k	# of flows in class *k*	Minimum fraction of BW for <u>each</u> flow in class *k*	Minimum fraction of BW for class *k* <u>in total</u>
1	2	1	2/(1*2+**2***3+1*5) = **2/13**	**2/13**
2	3	**2**	3/(1*2+**2***3+1*5) = **3/13**	**<u>6</u>/13**
3	5	1	5/(1*2+**2***3+1*5) = **3/10**	**5/13**

additional flow

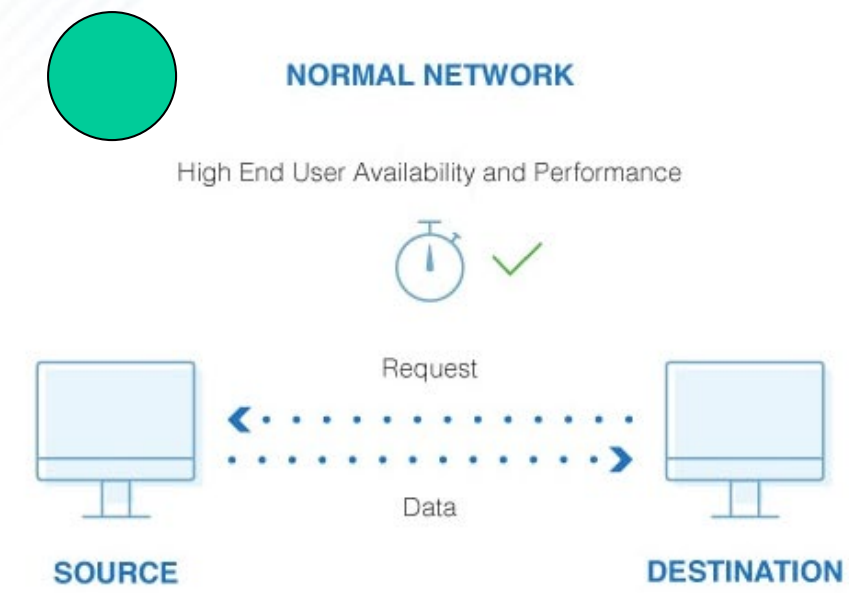
QoS Functionalities of Routers

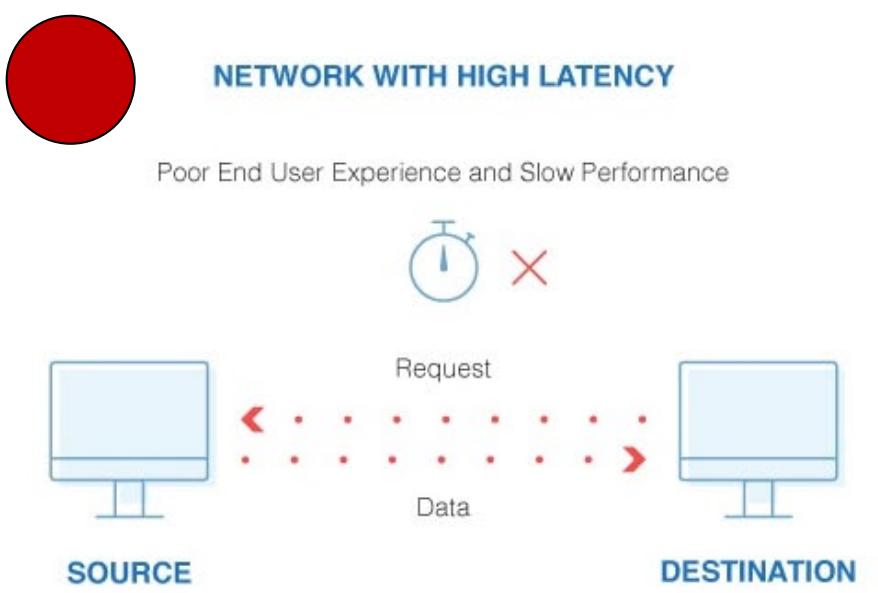


Idea: Routers can implements all kinds of QoS mechanisms

Recap on Latency and Bandwidth

Recap from First Lecture

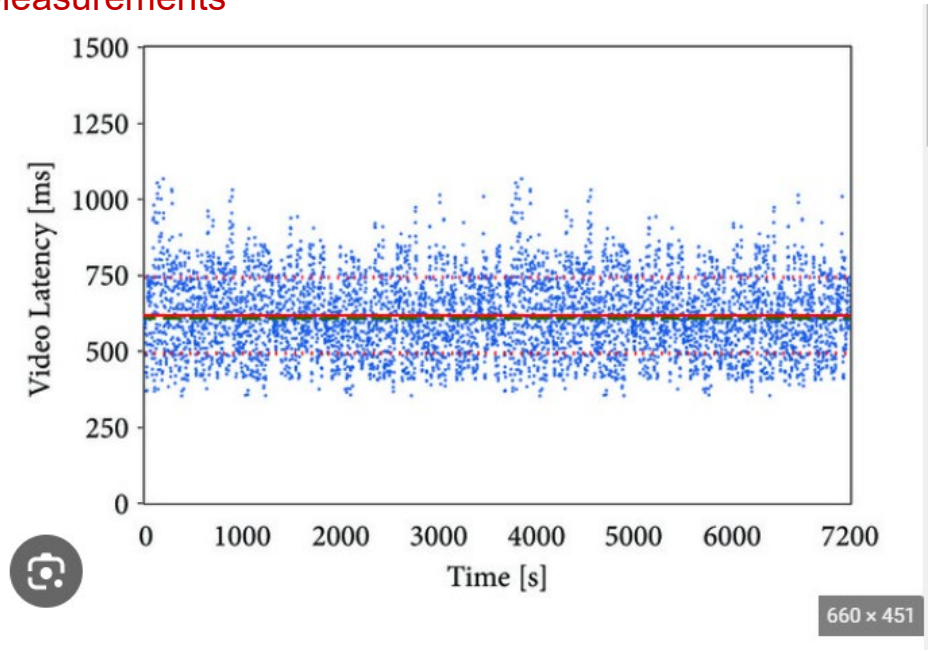


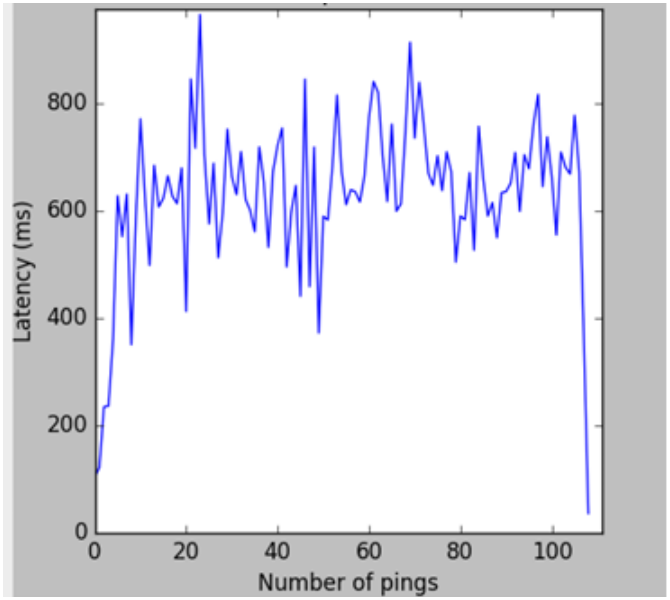


- Experiment to determine the bandwidth and latency of your network
- Use your phone to scan QR code
- Many apps to measure your latency and speed

Recap on Latency and Bandwidth

Measurements





Observation: bandwidth and latency differ across networks and devices

- Distance from device to the Internet service provider (Ziggo, KPN, ...)
- WiFi interference
- Network cable material (coaxial, optical, twisted, ...)
- Other devices on the network
- Possibly strong fluctuations over time

Traceroute

hop count

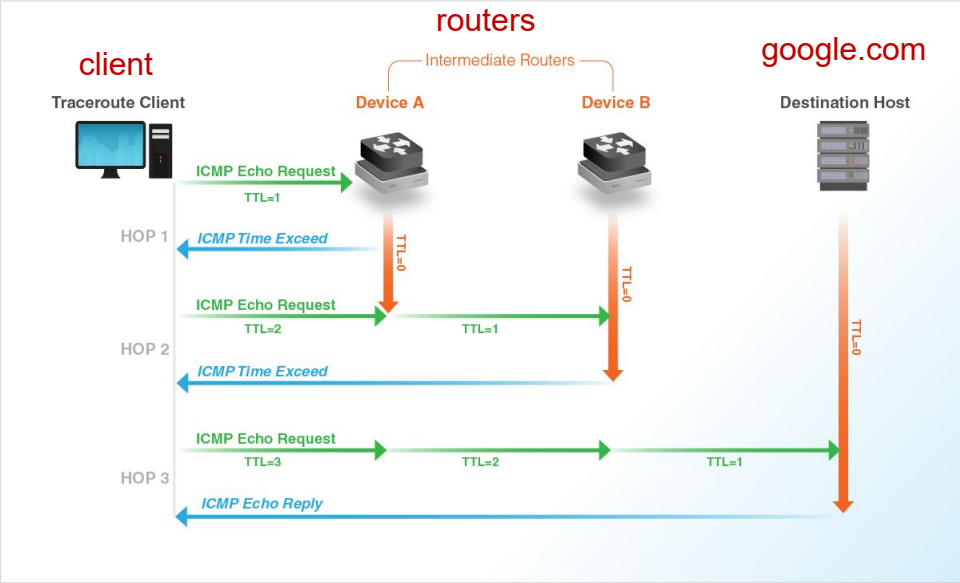
```
C:\>tracert google.com

Tracing route to google.com [172.217.17.110]
over a maximum of 30 hops:

  1     2 ms     2 ms     2 ms  router.home
  2     *        8 ms     9 ms  195.190.228.68
  3     *        *        *     Request timed out.
  4     9 ms    10 ms    11 ms  72.14.196.74
  5     9 ms    10 ms    10 ms  108.170.241.129
  6    32 ms     *       12 ms  108.170.236.223
  7    12 ms    11 ms    11 ms  ams15s29-in-f14.1e100

Trace complete.
```

RTT device → router | RTT time at host | RTT back to device

intermediate routers

client

google.com



TTL = Time-to-Live

- length of a path
- maximum # of hops

Traceroute **discovers the path of a packet to a certain host**

- Identifies the forwarding host (router)
- Determines the transit time delay to these hosts (‘ping’)
- Calculates the Round-rip Time (RTT)

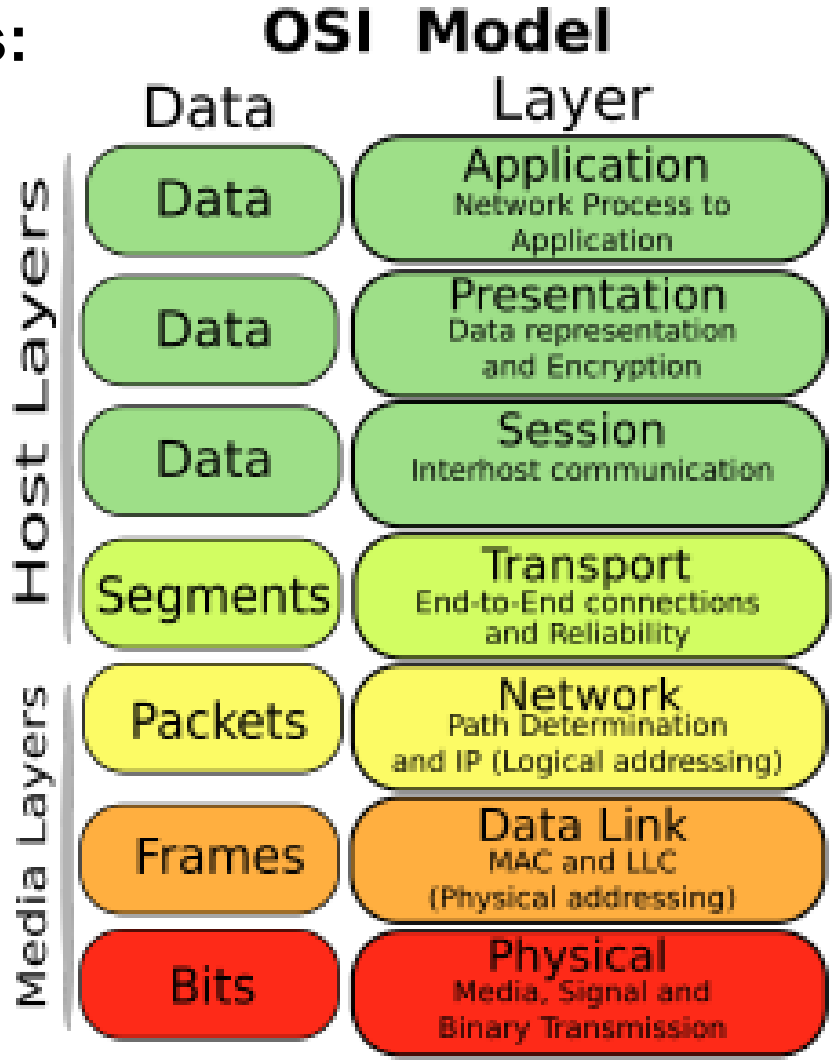
OSI Reference Model

Focus on the following layers:

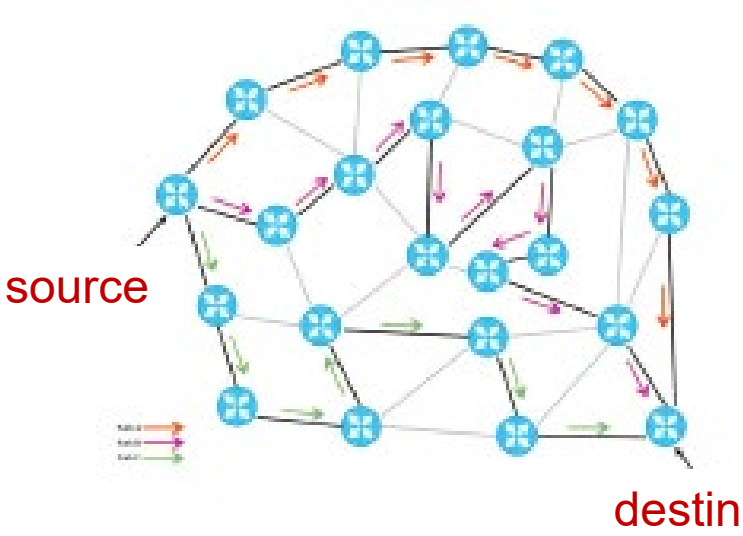
Application: HTTP and QUIC

Transport: TCP and UDP

Network: IP



Network Layer (Layer 3)



- Internet Protocol (IP) provides service of **routing packets**
- **'Best effort' delivery:** no guarantees about lost/duplicated packets
- **Consequence:** packet losses and/or delays
- **Observation:** no QoS guarantees

Transport Layer (Layer 4)

Transport Control Protocol

- Connection-oriented
- Reliable

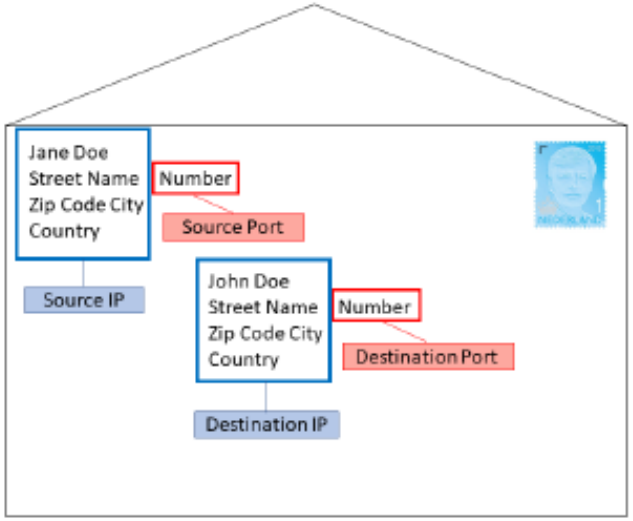
User Datagram Protocol

- Connectionless
- Unreliable

Used for time-sensitive applications:

- gaming
- playing videos
- Domain Name System (DNS) lookups

User Datagram Protocol (UDP)

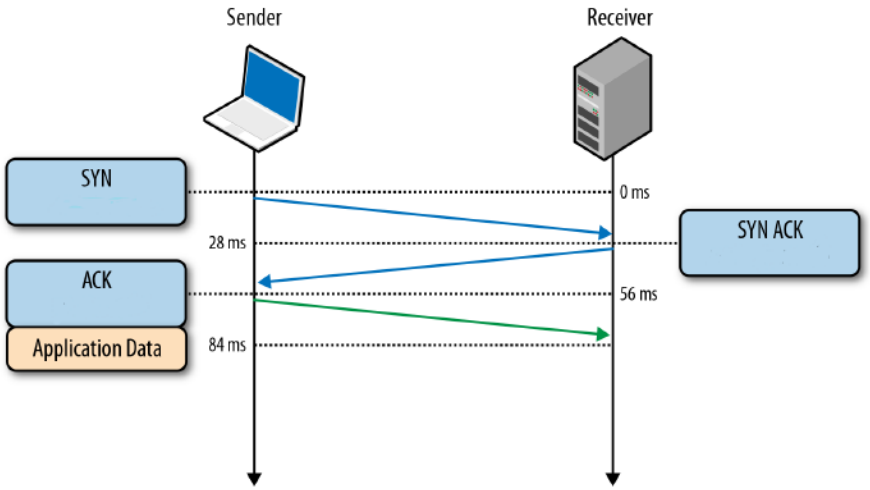


- Often referred to as **"Unreliable Datagram Protocol"**
- Datagram is an individual, self-contained packet
- **No sessions and no connection states** across packets
- With UDP, applications are in charge of data transfer protocol
- Why datagrams? Stream semantics not always needed (DNS, Skype, etc.)

Transmission Control Protocol ("TCP")

- TCP "packets" are called **segments**
- Implements the illusion of orderly transmission on top of the network layer
- **Connection-oriented**

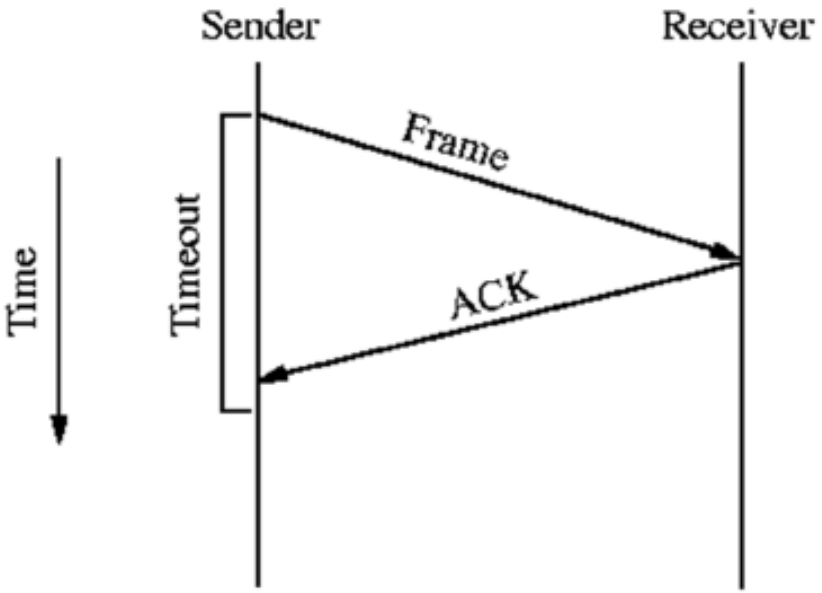
TCP Connection Setup



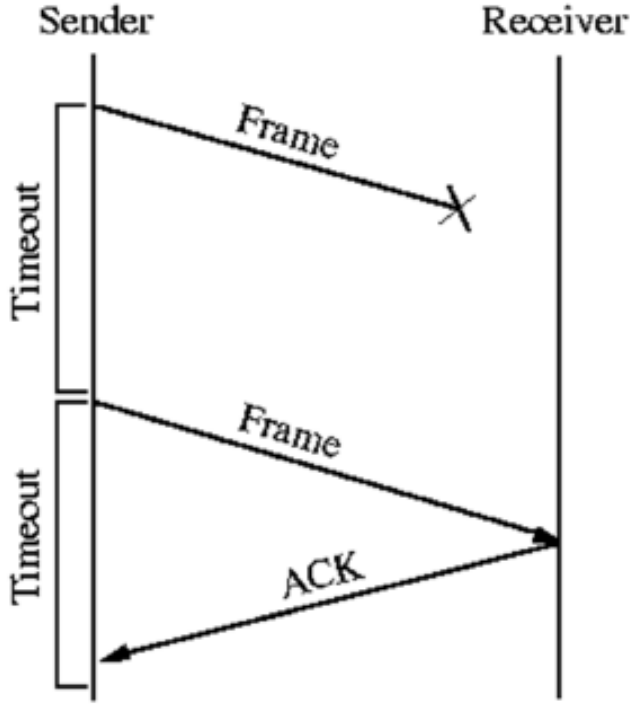
- (In)famous **"Three-way handshake"**
- Receiver sends ACK(nowledge) to confirm packet receipt

Transmission Errors

successful transmission

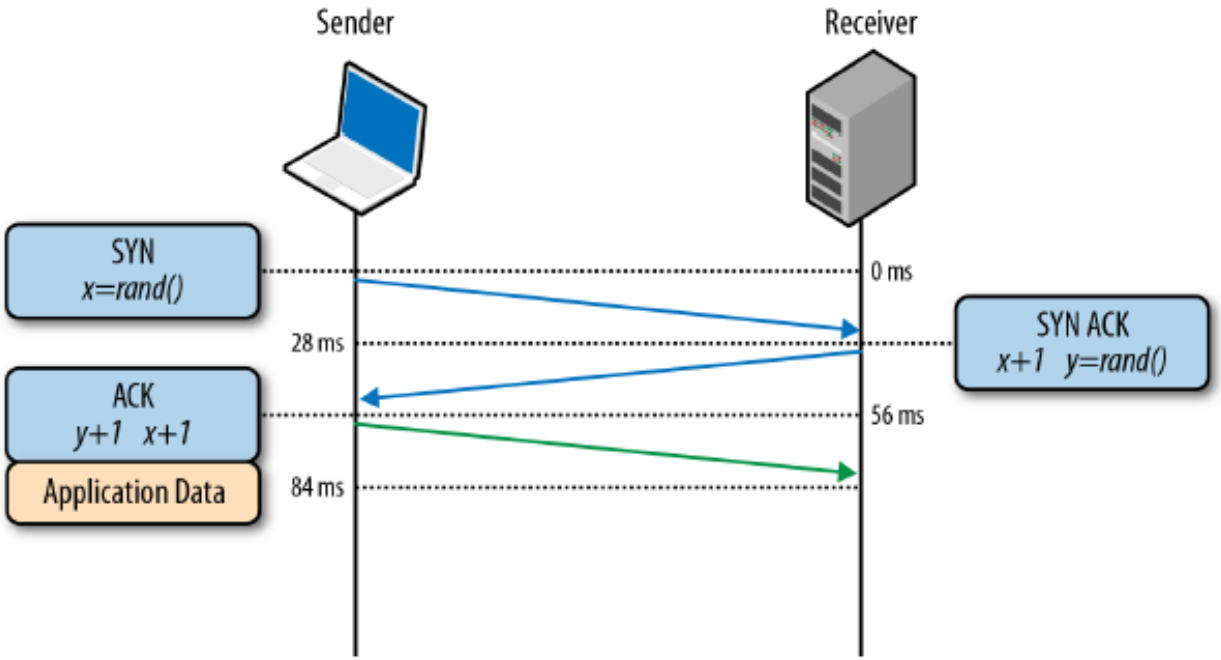


retransmision (due to a timeout)



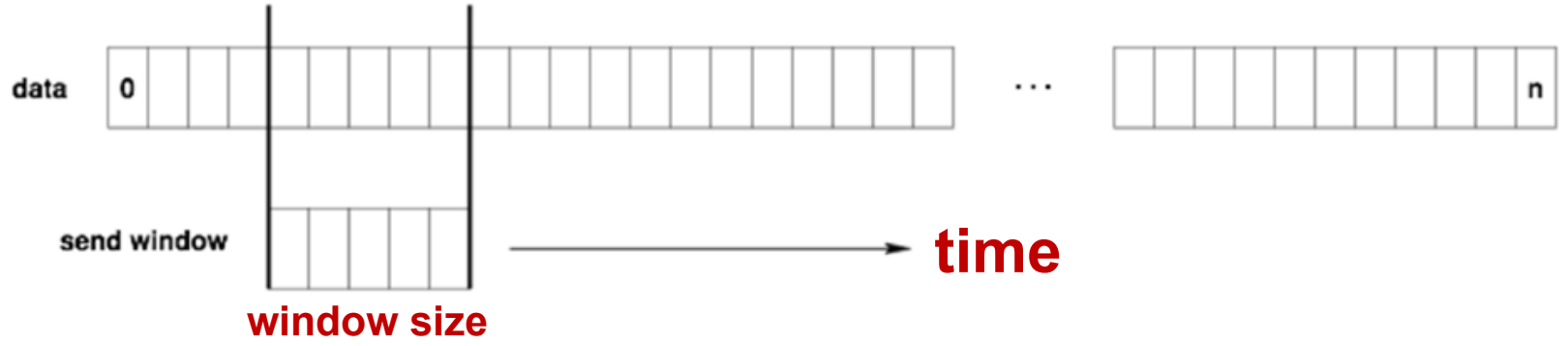
- **Basic principle:** confirm received packets by ACK
- If ACK does not arrive within a timeout, then retransmit

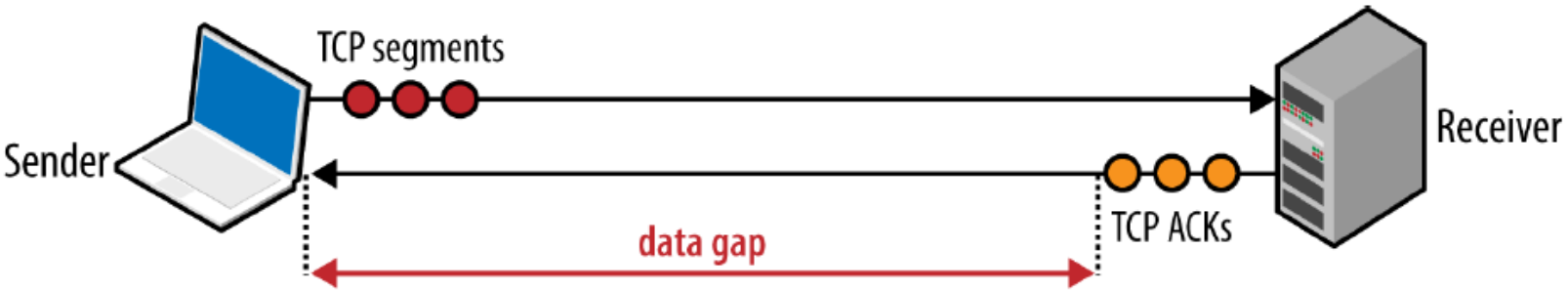
Packet Numbering



- Packets are identified with a **sequence number**
- Sender and receiver choose their first sequence number at random
- **Observation:** It is not efficient to just send only one packet
- **Question:** How to send multiple segments to the receiver?

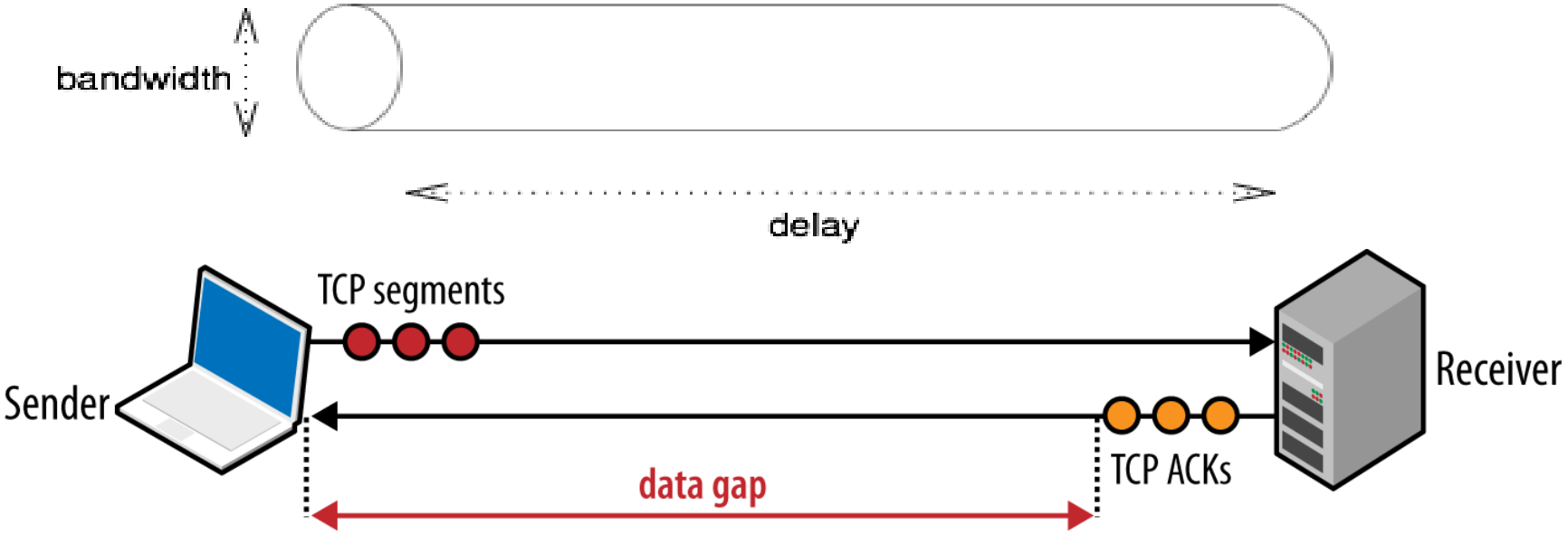
TCP Sliding Window Protocol





- Sender keeps (cyclic) **window of unacknowledged packets**
- Window **slides** along with transmission progress

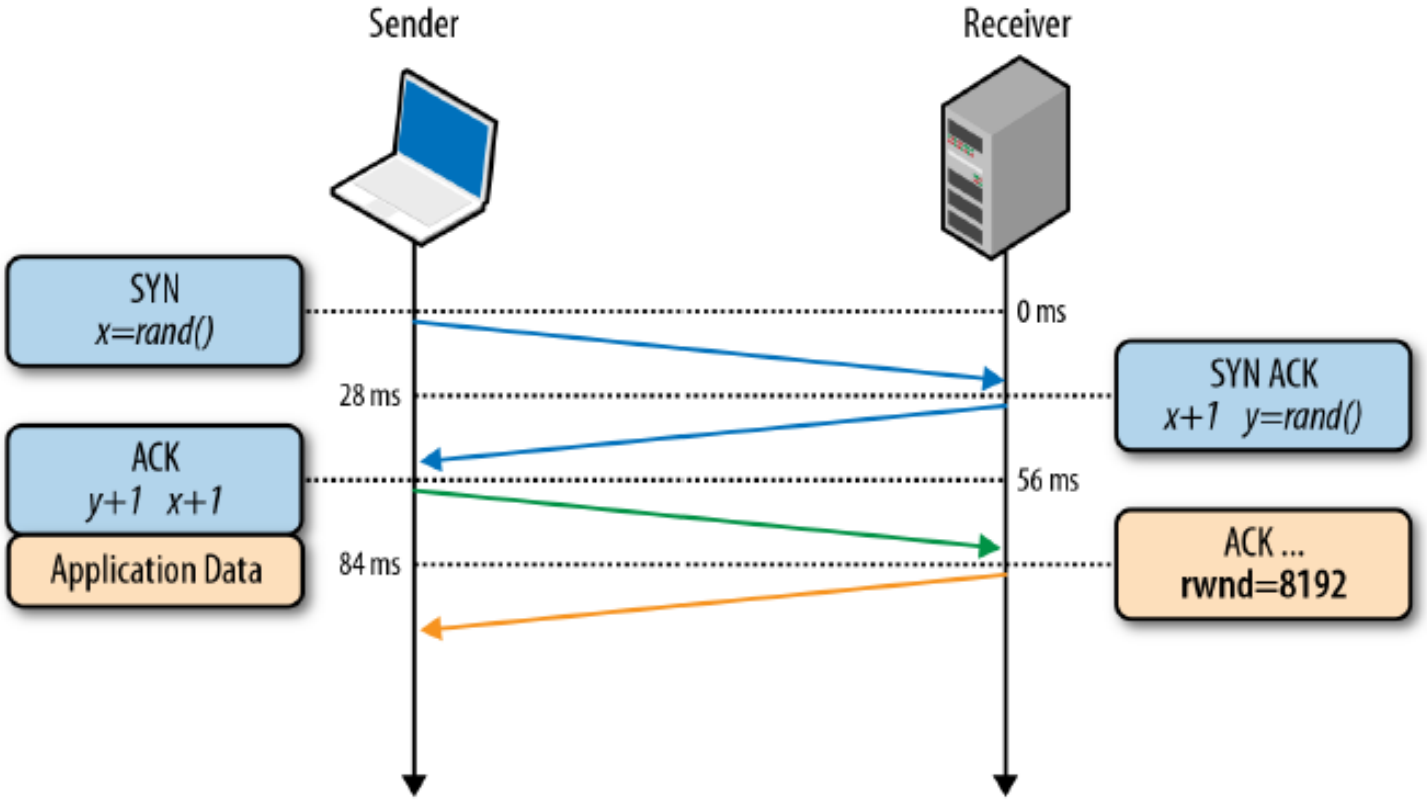
Bandwidth-Delay Product (BDP)



- **Flow control** in place to avoid overloading the receiver
- **Window size** = number of not yet acknowledged packets ("in flight")
- **BDP = bandwidth x RTT**
- "Every RTT time units, at most a BDP worth of data can be transmitted"

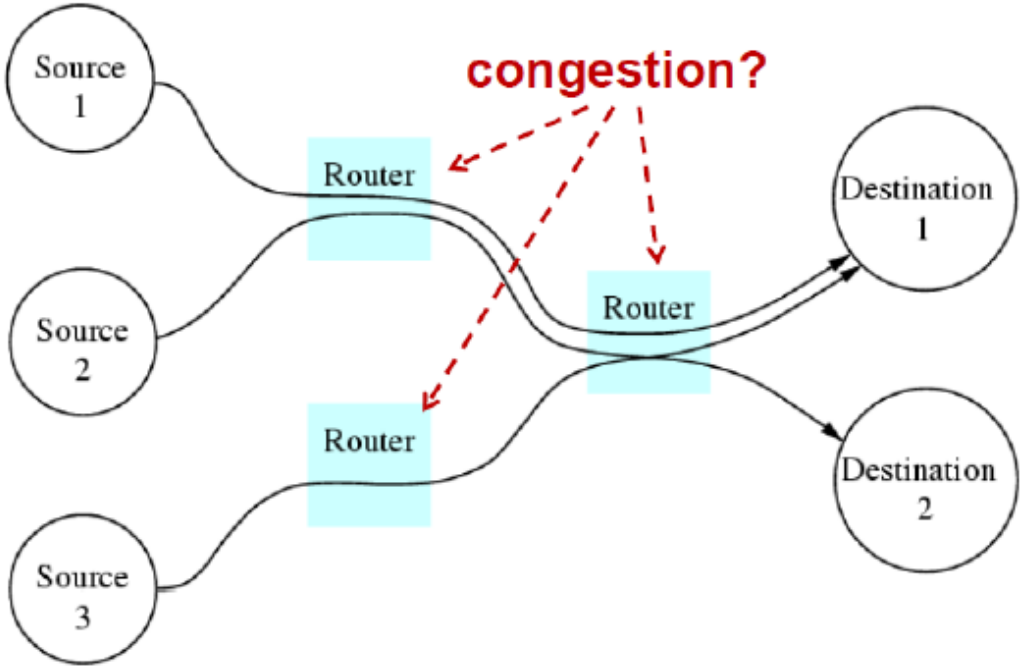
$$\text{maximum throughput of a TCP connection} = \frac{\text{maximum window size}}{\text{RTT}}$$

TCP Flow Control



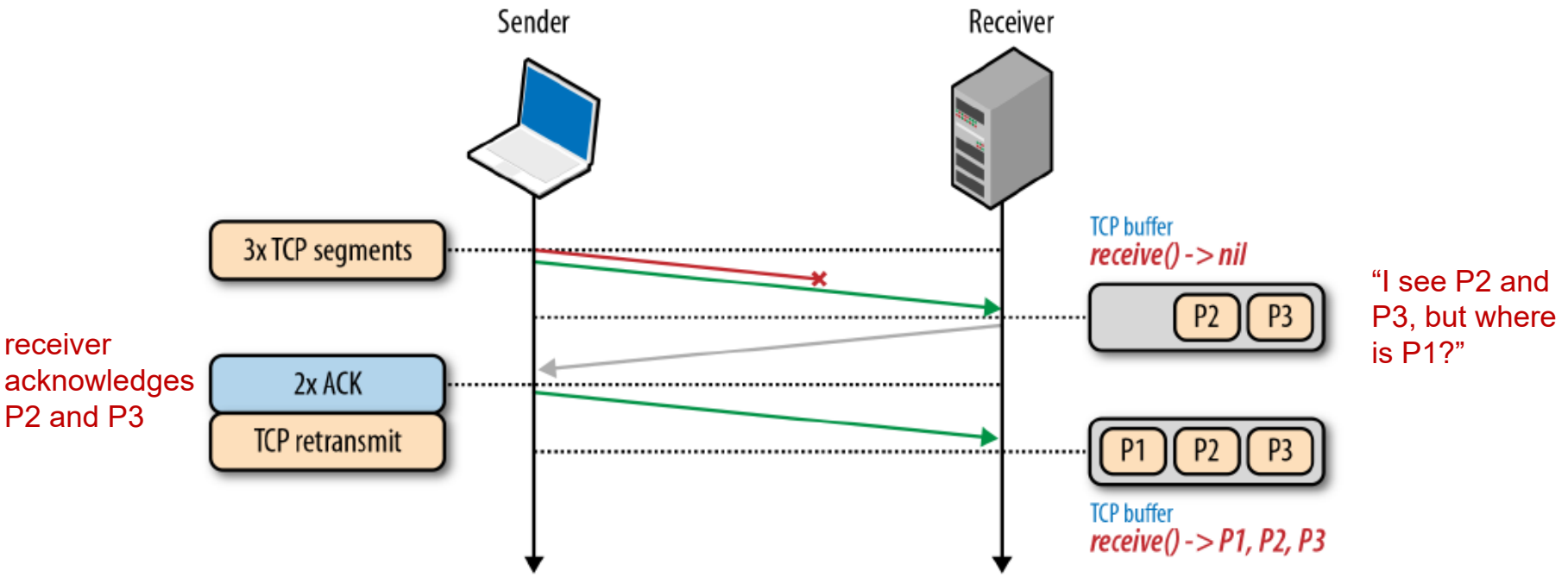
- Flow control is to **avoid overloading the receiver**
- We can send only as much as the receiver can handle

Network Congestion



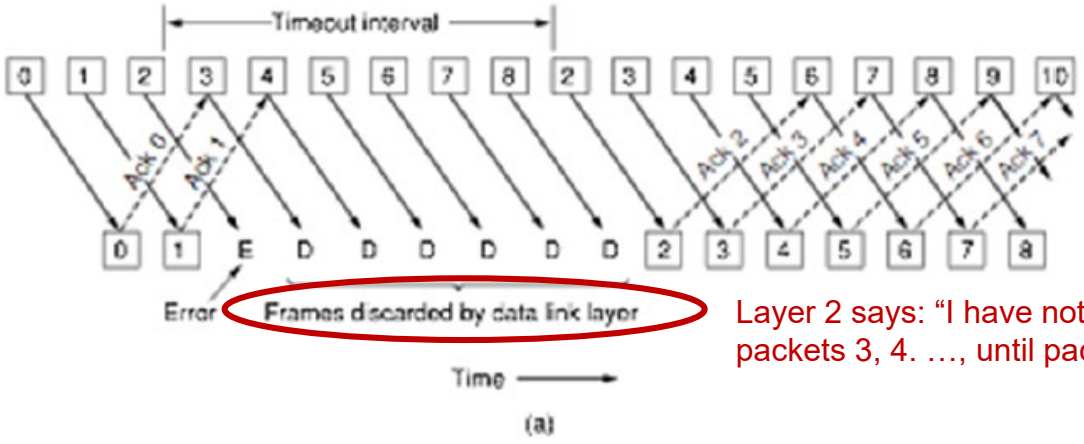
- Not only the receiver can be overloaded
- Congestion may happen **“somewhere in the network”**
- Sending might not notice that locally

Head-of-Line Blocking



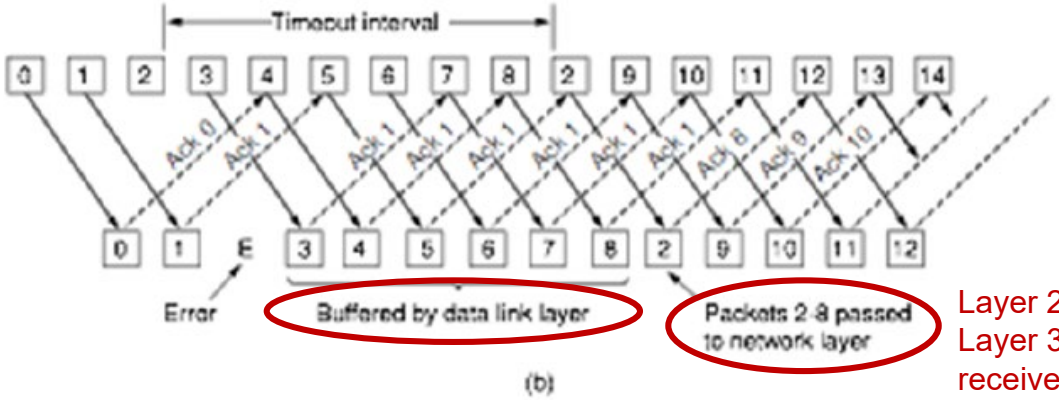
- The receiving process (application) simply **sees a delay in data delivery** when a packet has been lost ("packet 1")
- Data **behind** this packet has to wait until the missing packet has been retransmitted

TCP Cumulative Sliding Window Protocol



"naive way": frames thrown away

Layer 2 says: "I have not yet received packet 2, so I throw away packets 3, 4. …, until packet 2 has arrived"

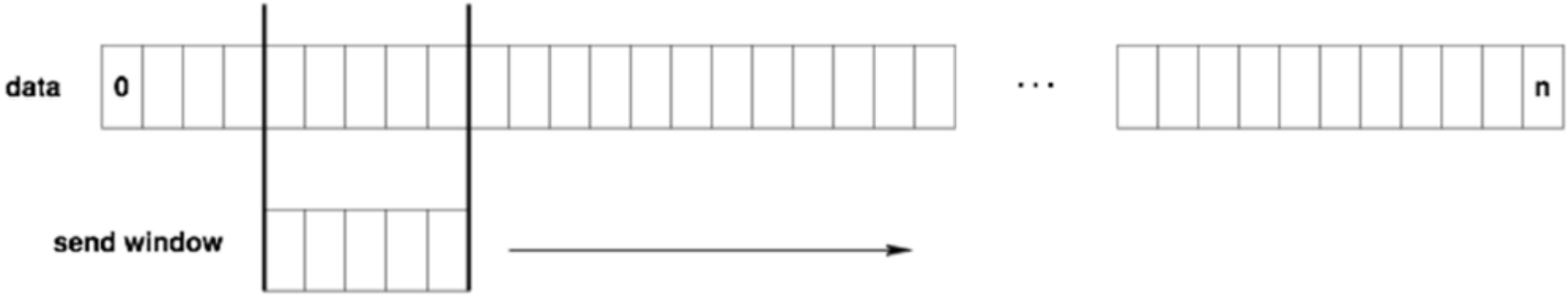


"smart way": frames buffered

Layer 2 says: "I buffer segments received"
Layer 3 says" "I acknowledge last segment I have received so far" (cumulative)

Cumulative ACK: confirms all packets up to n

TCP Sliding Window Protocol



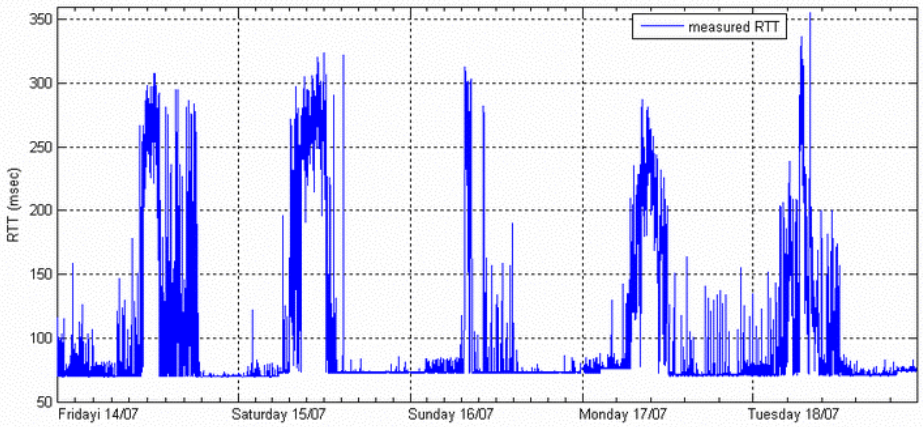
Serves three purposes:

1. Correction of transmission errors
2. **Flow control:** do not overload the receiver
3. **Congestion control:** do not overload the network

How to determine the "optimal" timeout time and sliding window size?

Adaptive Timeout Time

RTT measurements

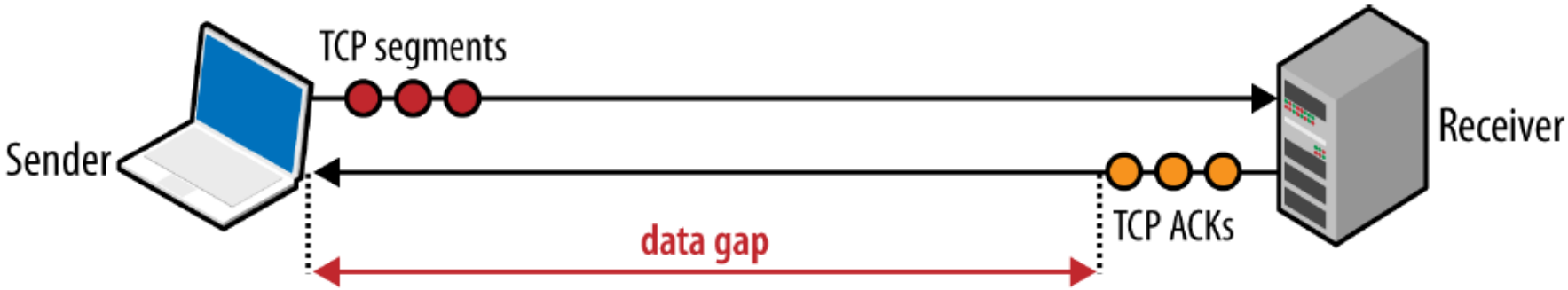


- RTT depends on distance and changing "network weather"
- **Implication:** sliding windows require an adaptive timeout parameter
- **Original TCP algorithm ("RTT changes over time")**

1. SampleRTT = T(ACK) – T(SEND)
2. EstimatedRTT = α Estimated RTT + (1 - α)SampleRTT
3. Timeout = 2 EstimatedRTT

TCP spec says: $0.8 \leq \alpha \leq 0.9$

TCP Sliding Window Size



- **Assumption:** sending should be able to transmit without interruption
- **Rule of thumb:** make send window big enough to bold a BDP worth of data

Link	BW	BDP
T1	1.5 Mbps	18 KB
Ethernet	10 Mbps	122 KB
FDDI	100 Mbps	1.2 MB
1G Ethernet	1000 Mbps	12 MB

- **Original TCP:** use 16 bits as send window
- **Now:** TCP header option for window scaling

TCP Congestion Control

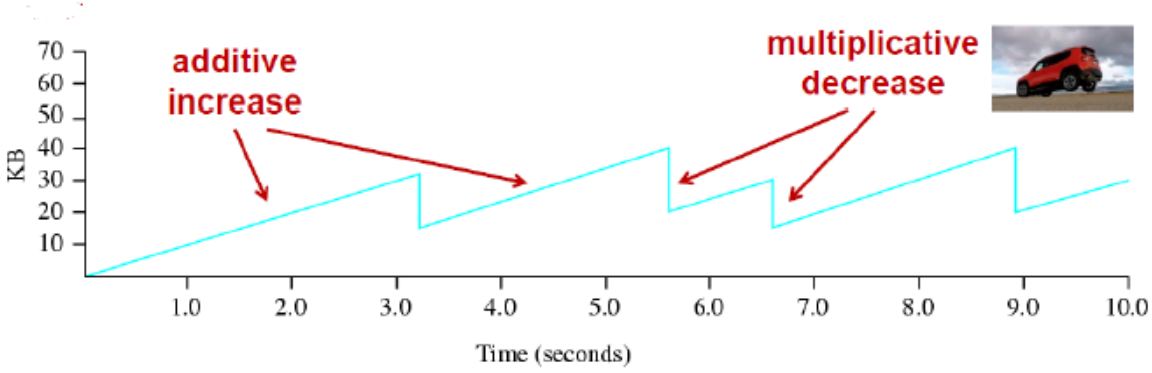
- Observing network capacity is hard

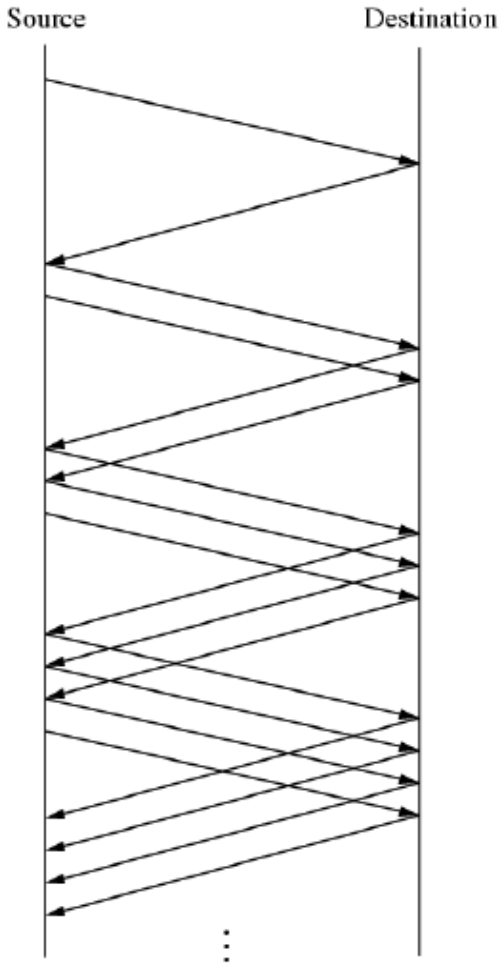
- TCP congestion control is
  - important for stability of the network
  - used to evaluate throughput and response times
  - used to improve network resource utilization

- Consists of **three parts**:
  - Slow Start (SS)
  - Congestion Avoidance (CA)
  - Fast Retransmit / Fast Recovery (FRR)

- Congestion control mechanisms: Tahoe, Reno, Vegas,…
  - we only discuss TCP Reno today

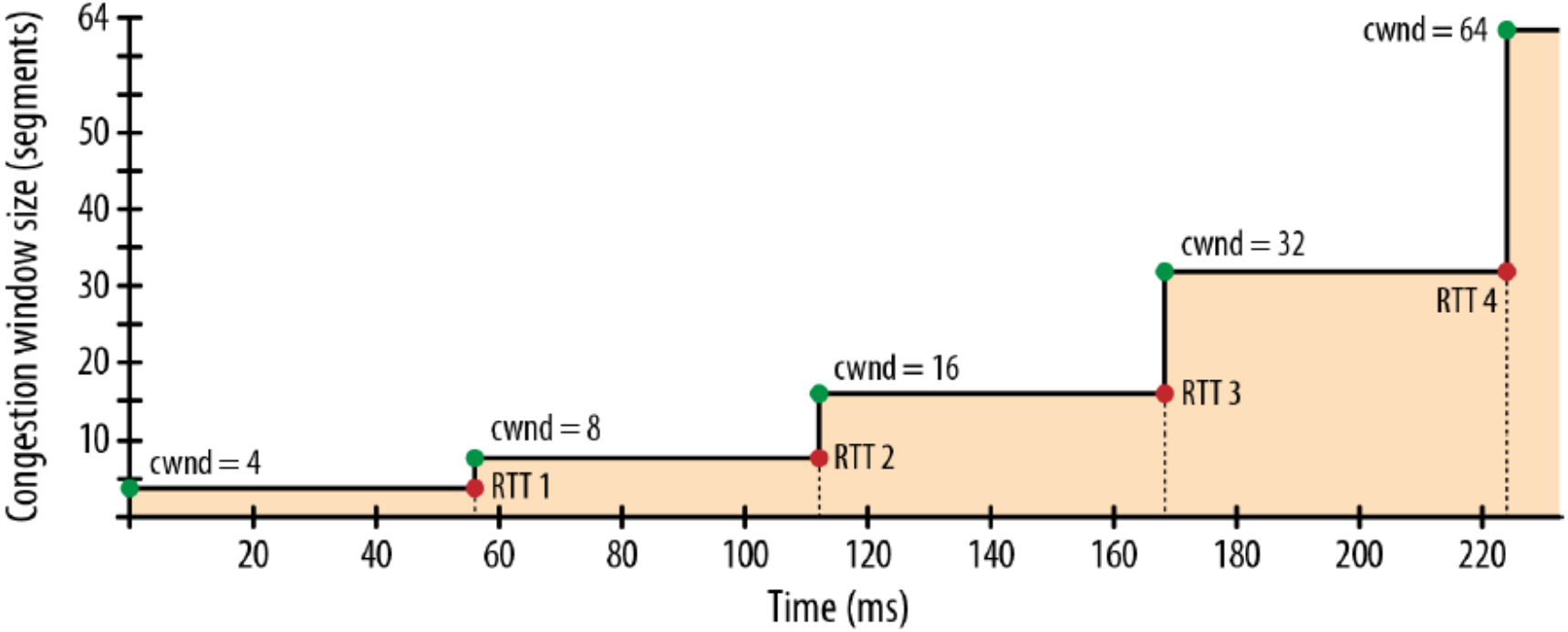
Additive Increase / Multiplicative Decrease (AIMD)





- CW large when no congestion, small otherwise
- How to **recognize congestion?** Timeouts!
- Timeout → CW := CW / 2 (minimum MSS)
- Else → CW := CW + increment
- Operates a TCP stream in “stable state”

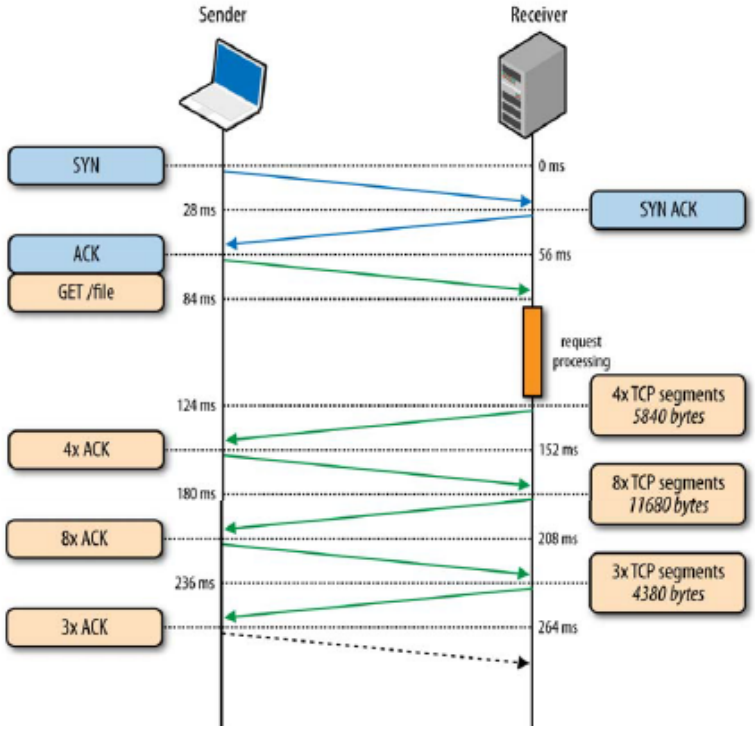
TCP Slow Start (SS)



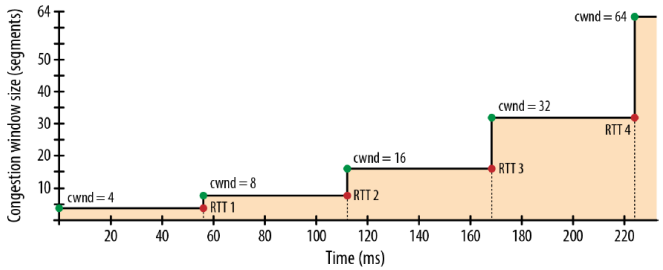
- How to initialize the CW (segments)?
- AIMD takes too long
- Advertised Window could be too aggressive
- Start with (for example) CW = 4, doubles every RTT

TCP SS Example: New York- London

Parameter	Value
File size	20 KB
RTT	56 ms
Bandwidth	5 Mbps
Receive window	64 KB
MSS	1460 bytes
Initial CW	4 segments
Server processing	40 ms



Net result: file transfer 264 ms over a new TCP connection

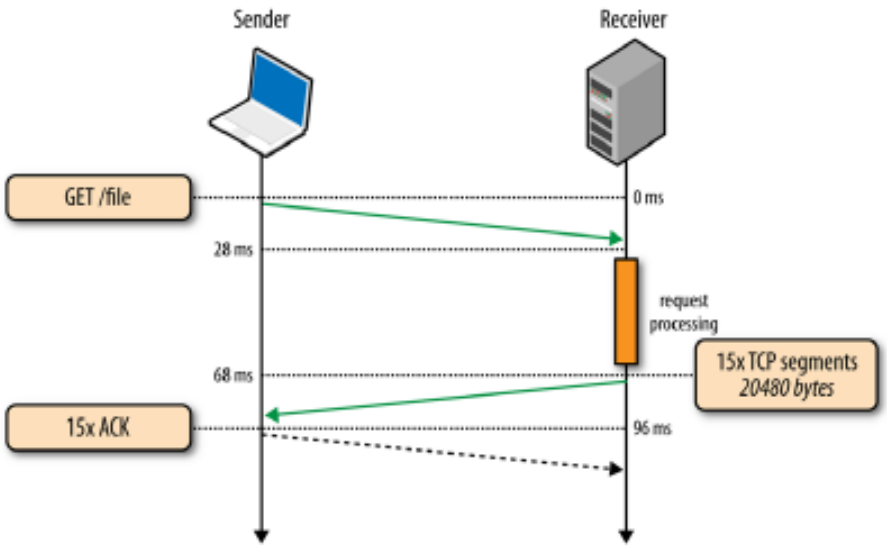


Problem: HTTP and TCP almost always in SS

Observation previous example: bandwidth did not show up at all!

Revised TCP specs: initial CW size increased to 16 segments

Goal: saving RTTs for HTTP traffic



More: reusing TCP connections leads to more reduction

Net results: reduction by 264 ms – 96 ms = 168 ms!