TÉCNICO LISBOA

Redes de Computadores

LEIC-A

3 – Transport Layer (part 2)

Prof. Paulo Lobato Correia

IST, DEEC – Área Científica de Telecomunicações

1

TÉCNICO LISBOA

Objectives

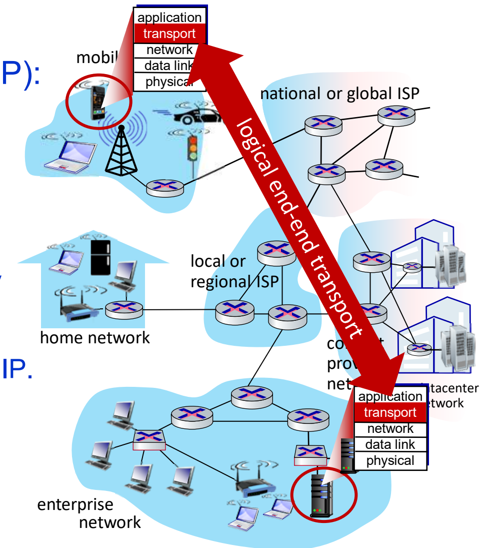
Application
Transport
Network
Link
Physical

- Understand the principles behind transport layer services:
  - Multiplexing/demultiplexing;
  - Reliable data transfer;
  - Flow control;
  - Congestion control.
- Transport layer protocols in the Internet:
  - UDP: connectionless transport;
  - TCP: connection-oriented transport with congestion control;
  - QUIC: Quick UDP Internet Connections.

4

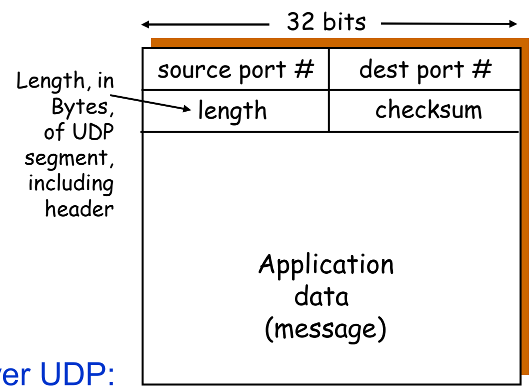
Internet Transport Layer Protocols

- Reliable, in-order delivery (TCP):
  - Connection setup;
  - Flow control;
  - Congestion control.
- Unreliable, unordered delivery (UDP):
  - Simple extension of “best-effort” IP.
- Services not available:
  - Delay guarantees;
  - Bandwidth guarantees.



UDP: User Datagram Protocol

- Often used for streaming multimedia applications:
  - Loss tolerant;
  - Rate sensitive.
- Other UDP uses
  - DNS;
  - SNMP;
  - QUIC.
- To have reliable transfer over UDP:
  - Reliability added at application layer;
  - Application-specific error recovery!

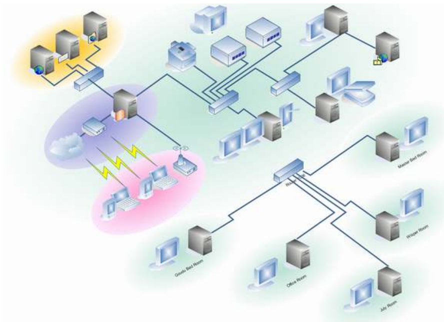


UDP segment format

TÉCNICO LISBOA

Outline

- Transport-layer services;
- Multiplexing and demultiplexing;
- Connectionless transport: UDP;
- Principles of reliable data transfer;
- Connection-oriented transport: TCP
  - Connection management;
  - Segment structure;
  - Reliable data transfer;
  - Flow control;
- Principles of congestion control
- TCP congestion control
- QUIC



TÉCNICO LISBOA

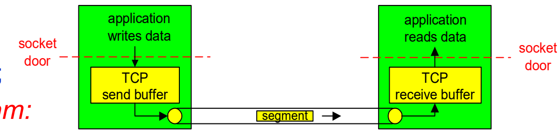
TCP: Overview

*Some RFCs: 675, **793**, **1122**, 1323, 1379, 1948, 2018, **5681**, 6247, **6298**, 6824, **7323**, 7424, …*

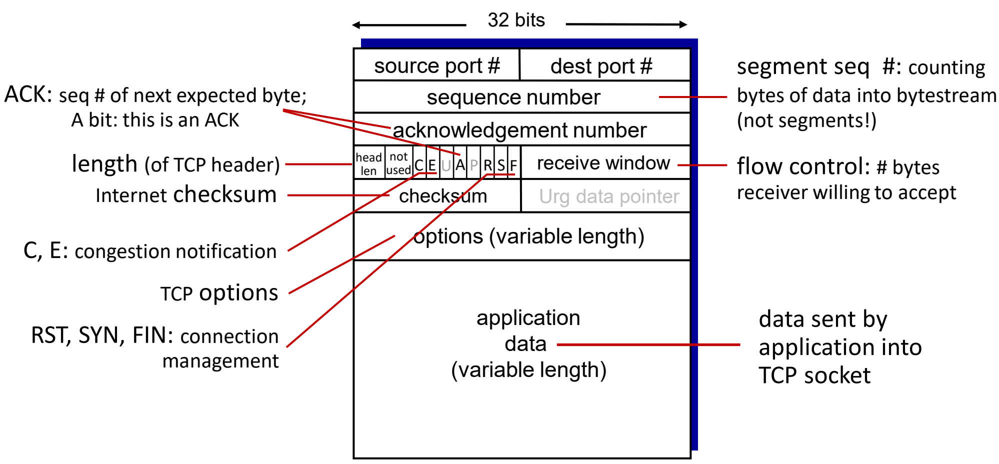
- RFC 793: "Transmission Control Protocol", STD 7 (*Sept 1981*) - fundamental TCP specification document
- RFC 1122: "Requirements for Internet Hosts - Communication Layers" (*Oct 1989*) - updates and clarifies RFC 793, fixes specification bugs and oversights. **Mandates that a congestion control mechanism must be implemented**.
- RFC 5681: "TCP Congestion Control" (*Aug 2009*) - defines **congestion avoidance and control mechanism for TCP**. RFCs 2001 and 2581 are conceptual precursors of RFC 5681.
- RFC 6298: "Computing TCP's Retransmission Timer" (*June 2011*)
- RFC 6691: "TCP Options and Maximum Segment Size (MSS)" (*July 2012*)
- RFC 7323: "TCP Extensions for High Performance" (*Sept 2014*)
- RFC 7414 - A Roadmap for Transmission Control Protocol (TCP) Specification Documents

TCP: Overview

- Point-to-point:
  - One sender, one receiver;
- Reliable, in-order *byte steam:*
  - No “message boundaries”;
- Sliding window:
  - TCP congestion and flow control set window size;
- *Send & receive buffers;*
- Full duplex data:
  - Bi-directional data flow in same connection;
  - **MSS**: maximum segment size;
- Connection-oriented:
  - Handshaking (exchange of control messages) initializes sender and receiver state before data exchange;
- Flow control:
  - Sender will not overwhelm receiver.



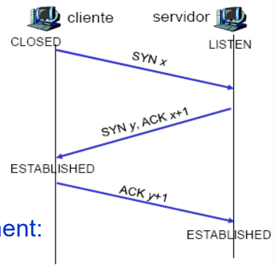
TCP segment structure



TCP Connection Management

Recall:

- TCP sender and receiver establish “connection” before exchanging data segments;
- Initialize TCP variables: seq. numbers, buffers, flow control information (e.g. **`RcvWindow`**);
- *Client:* initiates connection;
- *Server:* contacted by client.



Three-way handshake:

Step 1: Client sends TCP **SYN** segment to server:

- Specifies initial client seq. number (x);
- No data included.

Step 2: Server receives SYN and replies with **SYN ACK** segment:

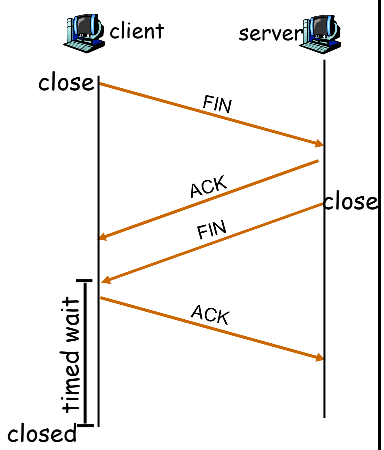
- Server allocates buffers;
- Specifies server initial seq. number (y).

Step 3: Client receives SYN ACK, replies with **ACK** segment, which **may already contain data**.

TCP Closing a Connection

Client and server each close their side of connection.

Example: client closes socket (*server could start*) :



Step 1: Client end system sends TCP **FIN** control segment to server;

Step 2: Server receives FIN, replies with **ACK**. Closes connection, sends **FIN**.

Step 3: Client receives FIN, replies with **ACK**.

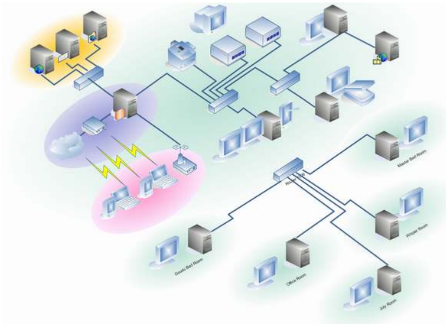
- Enters “**timed wait**” - will respond with ACK to received FINs.

Step 4: Server, receives ACK. Connection closed.

TÉCNICO LISBOA

Outline

- Transport-layer services;
- Multiplexing and demultiplexing;
- Connectionless transport: UDP;
- Principles of reliable data transfer;
- Connection-oriented transport: TCP
  - Connection management;
  - Segment structure;
  - Reliable data transfer;
  - Flow control;
- Principles of congestion control
- TCP congestion control
- QUIC



TÉCNICO LISBOA

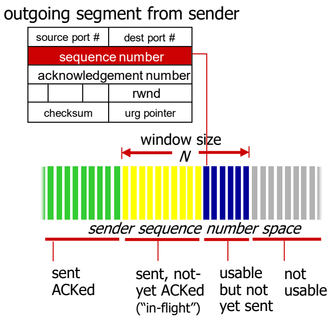
TCP Sequence Numbers, ACKs

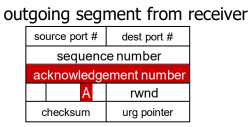
Sequence numbers:

- Byte stream – use **"number" of first byte** in segment's data

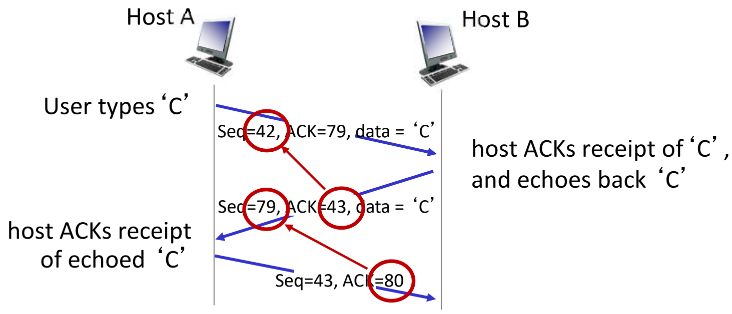
Acknowledgements:

- Seq **# of next byte expected** from other side
- Cumulative ACK





TCP Sequence Numbers, ACKs



simple telnet scenario

TCP Round Trip Time *and* Timeout

How to set TCP timeout value?

- Longer than RTT:
  - But RTT varies…
- Too short: premature timeout – unnecessary retransmissions.
- Too long: slow reaction to segment loss.

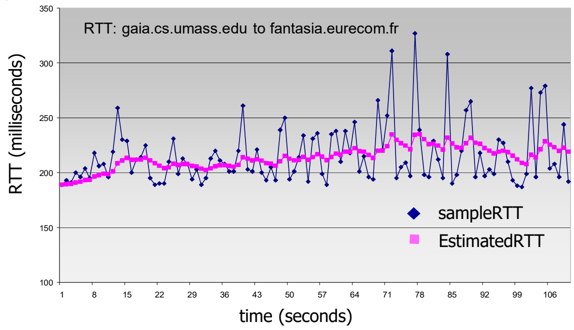
How to estimate RTT?

- `SampleRTT`: measured time between segment transmission and ACK receipt.
  - Ignore retransmissions;
- `SampleRTT` varies;
  Smoother RTT estimation (`EstimatedRTT`):
  - **Average several recent measurements, not just current `SampleRTT`.**

TCP Round Trip Time, Timeout

```
EstimateRTT = (1- α)*EstimatedRTT + α*SampleRTTd
```

- Exponential weighted moving average (EWMA)
- Influence of past sample decreases exponentially fast
- Typical value: α = 0.125



TCP Round Trip Time, Timeout

- timeout interval: **EstimatedRTT** plus "safety margin"
  - large variation in **EstimatedRTT:** want a larger safety margin

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

estimated RTT "safety margin"

- **DevRTT**: EWMA of **SampleRTT** deviation from **EstimatedRTT**:

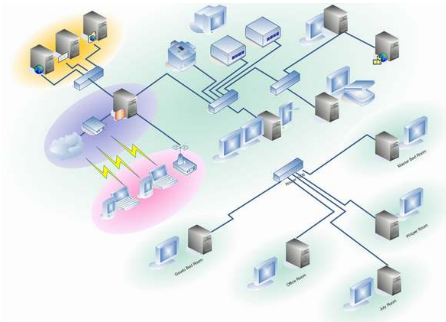
```
DevRTT = (1-β)*DevRTT + β*|SampleRTT-EstimatedRTT|
```

(typically, β = 0.25)

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

Outline

- Transport-layer services;
- Multiplexing and demultiplexing;
- Connectionless transport: UDP;
- Principles of reliable data transfer;
- Connection-oriented transport: TCP
  - Connection management;
  - Segment structure;
  - Reliable data transfer;
  - Flow control;
- Principles of congestion control
- TCP congestion control
- QUIC



TCP Reliable Data Transfer

- TCP creates a reliable data transfer (rdt) service on top of the unreliable service offered by IP;
- TCP uses:
  - **Sliding window**;
  - **Cumulative ACKs**;
  - A single **retransmission timer**;
- Retransmissions are triggered by:
  - Timeout events;
  - Duplicate ACKs.

Initially, let's consider a simplified TCP sender:

- Ignore duplicate ACKs
- Ignore flow control and congestion control.

TCP Sender Events

Data received from the application layer:

- Create segment;
- Sequence number is the **number** of the **first data byte** in the segment;
- Start timer (if not yet running – timer is for the oldest unacked segment);
- Expiration interval: `TimeOutInterval`

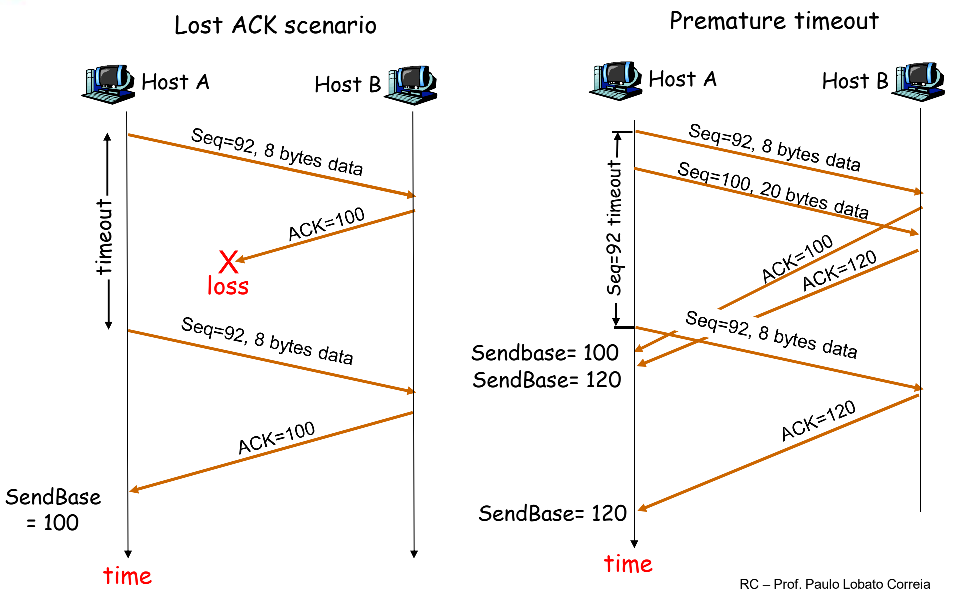
Timeout:

- Retransmit the segment that caused the timeout;
- Restart the timer

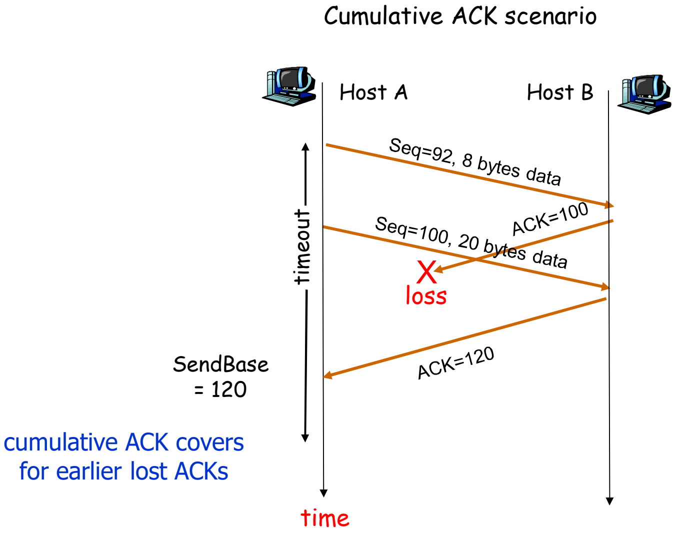
ACK received:

- If previously unacked, segments are acknowledged:
  - Update what is known to be ACKed;
  - Start timer if there are other outstanding segments.

TCP: Retransmission Scenarios

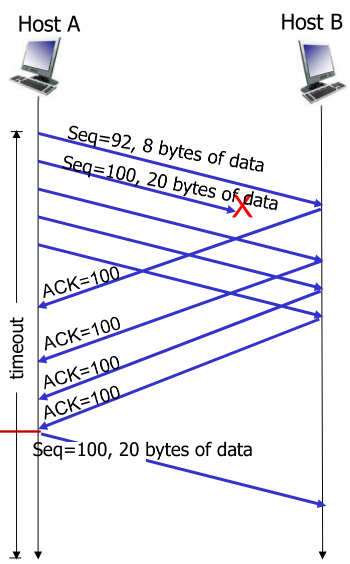


TCP: Retransmission Scenarios



TCP Fast Retransmit

- Time-out period is relatively long:
- **Detect lost segments via duplicate ACKs.**
  - Sender often sends many data segments;
  - If a segment is lost → many duplicate ACKs.
- If sender receives 3 additional ACKs, assumes the segment (after the ACKed data) was lost:
  - **Fast retransmit**: resends segment before timer expiration.



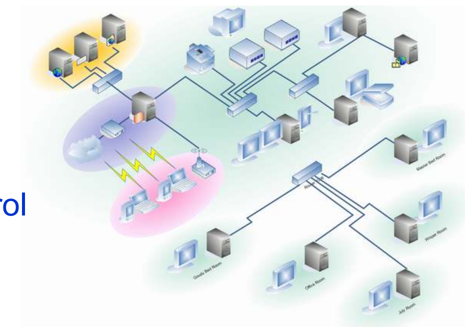
Receipt of three duplicate ACKs indicates 3 segments received after a missing segment – lost segment is likely. Retransmit!

TCP ACK Generation *[RFC 1122, RFC 2581]*

Event at Receiver	TCP Receiver action
Arrival of **in-order** segment with expected seq #. All data up to expected seq # already ACKed	***Delayed ACK***. **Wait up to 500ms** for next segment. If no next segment, send ACK
Arrival of **in-order** segment with expected seq #. One **other segment has ACK pending**	Immediately **send** single **cumulative ACK**, ACKing both in-order segments
Arrival of **out-of-order** segment higher-than-expect seq. # . **Gap detected**	Immediately **send** ***duplicate ACK***, indicating seq. # of next expected byte
Arrival of segment that **partially or completely fills gap**	Immediately **send ACK**, provided that segment starts at lower end of gap

Outline

- Transport-layer services;
- Multiplexing and demultiplexing;
- Connectionless transport: UDP;
- Principles of reliable data transfer;
- Connection-oriented transport: TCP
  - Connection management;
  - Segment structure;
  - Reliable data transfer;
  - Flow control;
- Principles of congestion control
- TCP congestion control
- QUIC



TÉCNICO LISBOA

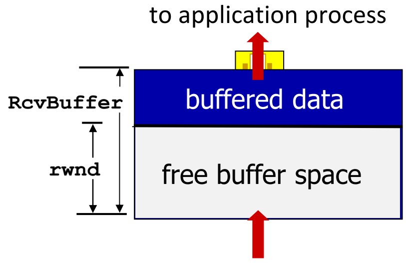
TCP Flow Control

Goal: ensure that sender won't overflow the receiver's buffer by transmitting too fast.

TCP has a receive buffer:

Application process may be slow at reading from buffer

Speed-matching: adjust send rate to the receiving application processing rate

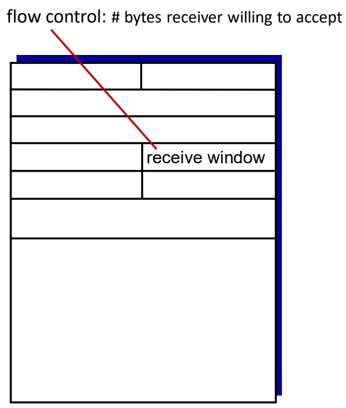


TCP receiver-side buffering

TÉCNICO LISBOA

TCP Flow Control

- TCP receiver "advertises" free buffer space in **`rwnd`** field in TCP header
  - **`RcvBuffer`** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **`RcvBuffer`**
- Sender limits amount of unACKed ("in-flight") data to received **`rwnd`**
  - Guarantees receive buffer will not overflow



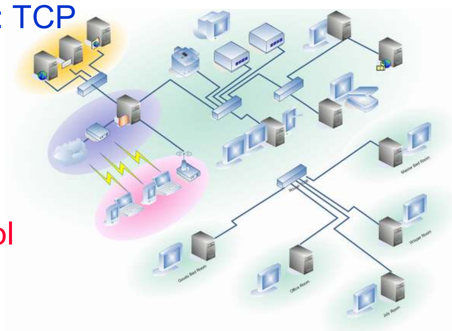
TCP segment format

Test applet at:
https://media.pearsoncmg.com/aw/ecs_kurose_compnetwork_7/cw/content/interactiveanimations/flow-control/index.html

TÉCNICO LISBOA

Outline

- Transport-layer services;
- Multiplexing and demultiplexing;
- Connectionless transport: UDP;
- Principles of reliable data transfer;
- Connection-oriented transport: TCP
  - Connection management;
  - Segment structure;
  - Reliable data transfer;
  - Flow control;
- Principles of congestion control
- TCP congestion control



RC – Prof. Paulo Lobato Correia 34

34

TÉCNICO LISBOA

Principles of Congestion Control

congestion control:
too many senders,
sending too fast

Congestion:

- Informally:
  "Too many sources sending
  too much data too fast
  for the *network* to handle";
- Different from flow control!
- Manifestations/consequences:
  - **Lost packets** (*buffer overflow at routers*);
  - **Long delays** (*queueing in router buffers*).
- A top-10 problem!

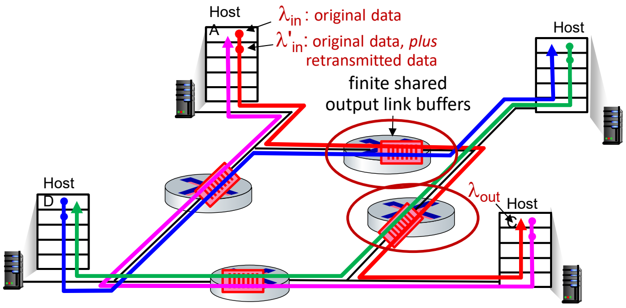
RC – Prof. Paulo Lobato Correia 35

35

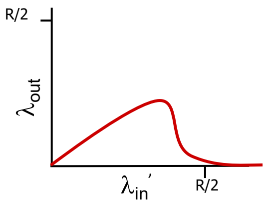
Causes/Costs of Congestion

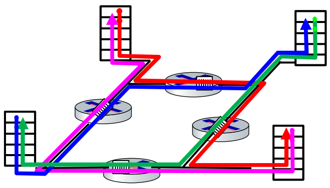
- *four* senders
- *multi-hop* paths
- timeout/retransmit

As red λ_{in}' increases, all arriving blue pkts at upper queue are dropped, blue throughput → 0



Causes/Costs of Congestion



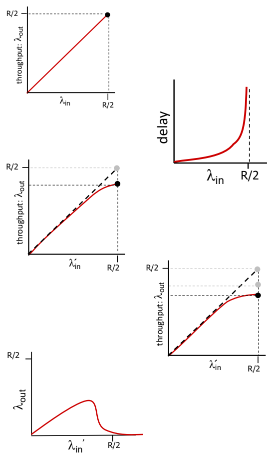


Another “cost” of congestion:

- when packet dropped, any upstream transmission capacity and buffering used for that packet was wasted!

Causes/costs of Congestion: Insights

- Throughput can never exceed capacity
- Delay increases as capacity approached
- Loss/retransmission decreases effective throughput
- Un-needed duplicates further decreases effective throughput
- Upstream transmission capacity / buffering wasted for packets lost downstream



Approaches Towards Congestion Control

Two approaches towards congestion control:

Network-assisted congestion control:

- Routers provide feedback to end systems:
  - Bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM);
  - Router may inform sender explicitly of supported rate.

End-end congestion control:

- No explicit feedback from network;
- **Congestion inferred from end-system observed loss and delay**;
  - It's the approach taken by TCP.

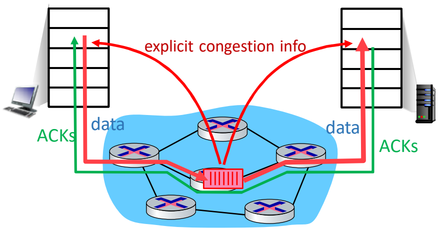
TÉCNICO LISBOA

Approaches Towards Congestion Control

Network-assisted congestion control:

- Routers provide *direct* feedback to sending/receiving hosts with flows passing through congested router
- May indicate congestion level or explicitly set sending rate

- **TCP ECN**, ATM, DECbit protocols



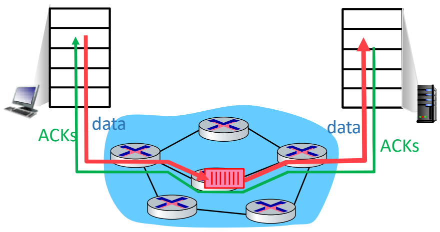
TÉCNICO LISBOA

Approaches Towards Congestion Control

End-end congestion control:

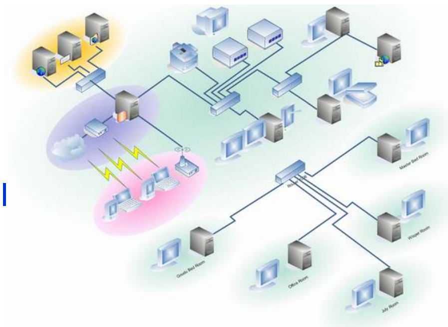
- No explicit feedback from network
- Congestion *inferred* from observed loss, delay

- Approach taken by TCP



Outline

- Transport-layer services;
- Multiplexing and demultiplexing;
- Connectionless transport: UDP;
- Principles of reliable data transfer;
- Connection-oriented transport: TCP
  - Connection management;
  - Segment structure;
  - Reliable data transfer;
  - Flow control;
- Principles of congestion control
- TCP congestion control
- QUIC

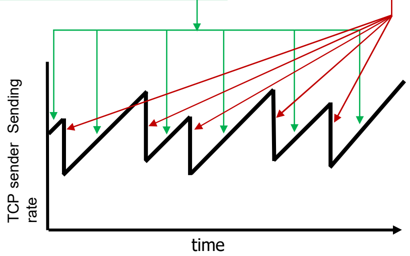


TCP Congestion Control: AIMD

- *Approach:* senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss event

Additive Increase
Increase sending rate by 1 maximum segment size every RTT until loss detected

Multiplicative Decrease
Cut sending rate in half at each loss event



AIMD sawtooth behavior: *probing* for bandwidth

TCP AIMD: more

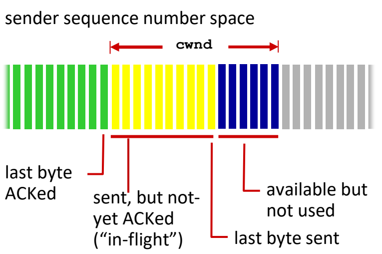
Multiplicative decrease detail – sending rate is:

- Cut in half on loss detected by triple duplicate ACK (TCP Reno)
- Cut to 1 MSS (maximum segment size) when loss detected by timeout (TCP Tahoe)

Why AIMD?

- AIMD – a distributed, asynchronous algorithm – has been shown to:
  - Optimize congested flow rates network wide!
  - Have desirable stability properties

TCP Congestion Control: Details



TCP sending behavior:

- *Roughly:* send `cwnd` bytes, wait RTT for ACKS, then send more bytes

$$\text{TCP rate} \approx \frac{\texttt{cwnd}}{\text{RTT}} \text{ bytes/sec}$$

- TCP sender limits transmission: `LastByteSent- LastByteAcked ≤ cwnd`
- `cwnd` is dynamically adjusted in response to observed network congestion (implementing TCP congestion control)

typically `cwnd` < `rwnd`

TCP Congestion Control: Details

How does sender perceive congestion?

- Loss events:
  - Timeout (TCP Tahoe);
  - 3 duplicate ACKs (TCP Reno).

- TCP sender reduces rate (**`cwnd`**) after loss event.

Three mechanisms:

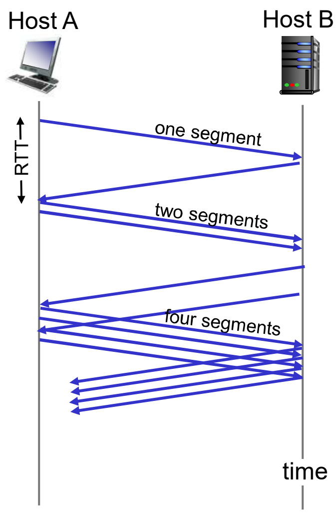
- AIMD (additive increase, multiplicative decrease);
- Slow start;
- Conservative after timeout events.

TCP Slow Start

- When connection begins, rate increases exponentially until first loss event:
  - Initially **`cwnd`** = 1 MSS
  - Double **`cwnd`** every RTT;
  - This is done by incrementing **`CongWin`** of MSS **for every ACK received**:

    `cwnd = cwnd + MSS`

- Slow start: The **initial rate is slow** but it **grows exponentially** fast.

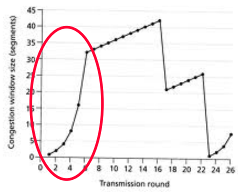


TCP Slow Start

- When connection begins, **`cwnd`** = 1 MSS
  - Example: MSS = 500 bytes & RTT = 200 msec
  - Initial rate (~MSS/RTT) = 20 kbps
  - Too slow ...
- Available bandwidth may be >> MSS/RTT
  - It is desirable to quickly ramp up to a respectable rate.
- **Slow Start**:
  - **When connection begins,**

    rate increases exponentially fast

    until first loss event or cwnd = Thr.



Loss Events

- After *3 duplicate ACKs* (→ **Congestion Avoidance**):
  - **`cwnd`** is cut in half;
  - Window then grows linearly.
- But after *timeout* (→ **Slow Start**):
  - **`cwnd`** is set to 1 MSS;
  - The window then grows exponentially up to a threshold, then grows linearly.

Philosophy:

- 3 duplicate ACKs indicate that the network is still capable of delivering some segments;
- A timeout indicates a “more alarming” congestion scenario.

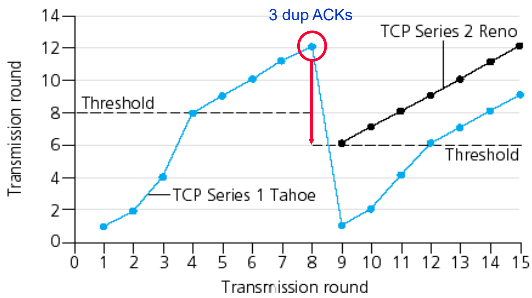
TÉCNICO LISBOA

Loss Events

Q: When should the exponential increase switch to linear?
A: When **cwnd** gets to 1/2 of its value before timeout.

Implementation:

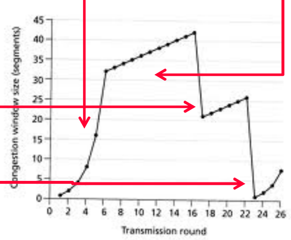
- Variable **Threshold**;
- At loss event, **Threshold** is set to 1/2 of **cwnd** just before loss event.



TÉCNICO LISBOA

Summary: TCP Congestion Control

- When **cwnd** is below **Threshold**, sender is in slow-start phase: the window grows exponentially.
- When **cwnd** is above **Threshold**, sender is in congestion-avoidance phase: window grows linearly.
- When a triple duplicate ACK occurs: **Threshold** set to **cwnd/2** and **cwnd** set to **Threshold**.
- When timeout occurs: **Threshold** set to **cwnd/2** and **cwnd** is set to 1 MSS.



TCP Sender Congestion Control

Fast Recovery state

When detecting **three duplicated ACKs** and before moving to congestion avoidance state, *try to speed up recovery* by fast resend of unacknowledged data:

- **cwnd** is increased by 1 MSS for every duplicate ACK received for the missing segment that caused TCP to enter the **fast recovery state**;

When receiving ACK for the missing segment:

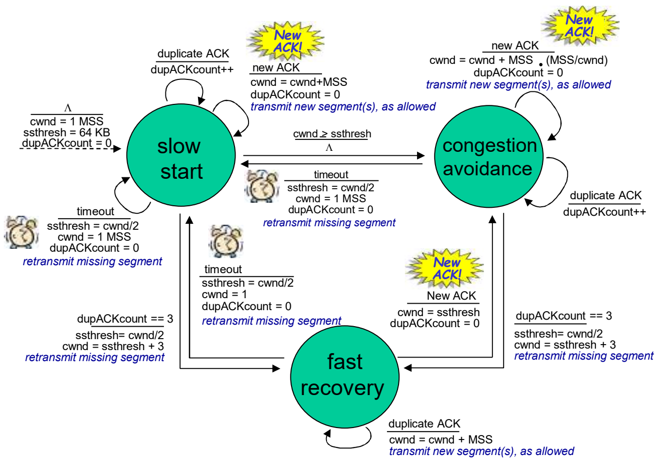
- Move to **congestion avoidance** state (after deflating `cwnd`).

Fast recovery is a recommended, but not required, component of TCP [RFC 5681].

TCP Sender Congestion Control

State	Event	TCP Sender Action	Comment
Slow Start (SS)	ACK receipt for previously unacked data	cwnd = cwnd **+ MSS**, If (cwnd > Threshold) state = "**Congestion Avoidance**"	**Double cwnd every RTT**
Congestion Avoidance (CA)	ACK receipt for previously unacked data	cwnd = cwnd **+ MSS*(MSS/cwnd)**	Additive increase: **cwnd increases 1 MSS every RTT**
SS or CA	Loss event detected by **triple duplicate ACK**	Threshold = cwnd/2, cwnd = Threshold + 3 MSS, State = "**Fast Recovery**"	Enter **fast recovery**, implementing **multiplicative decrease**. cwnd will not drop below 1 MSS.
Fast Recovery (FR)	ACK receipt for the previously unacked data	cwnd = Threshold, dupACKcount=0 State = "**Congestion Avoidance**"	Exit fast recovery
FR	duplicate ACK	cwnd = cwnd + MSS	FR – increase cwnd
SS, CA or FR	**Timeout**	Threshold = cwnd/2, **cwnd = 1 MSS**, State = "**Slow Start**"	Enter **Slow Start**
SS or CA	Duplicate ACK	dupACKcount ++	cwnd and Threshold not changed

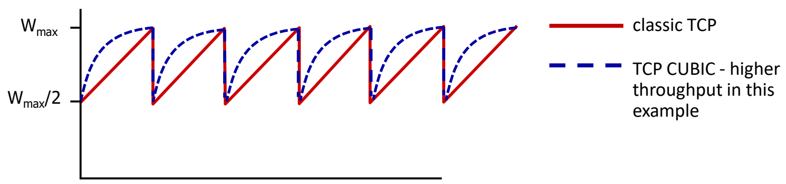
TCP Sender Congestion Control



67

TCP CUBIC

- Is there a better way than AIMD to “probe” for usable bandwidth?
- Insight/intuition:
  - W_{max}: sending rate at which congestion loss was detected
  - congestion state of bottleneck link probably (?) hasn’t changed much
  - after cutting rate/window in half on loss:
    - **start towards W_{max} *faster*, but then approach W_{max} *slowly***

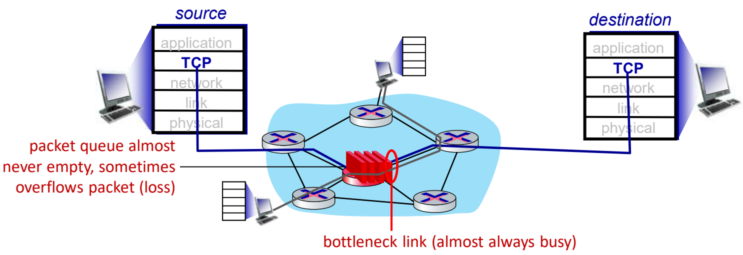


TCP CUBIC - default in Linux, most popular TCP for popular Web servers

68

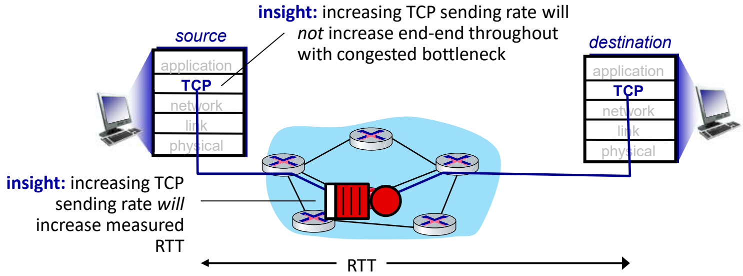
TCP and Congested "bottleneck link"

- TCP (classic, CUBIC) increase TCP's sending rate until packet loss occurs at some router's output: the *bottleneck link*



TCP and Congested "bottleneck link"

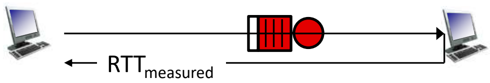
- TCP (classic, CUBIC) increase TCP's sending rate until packet loss occurs at some router's output: the *bottleneck link*
- Understanding congestion: useful to focus on congested bottleneck link



Goal: *"keep the end-end pipe just full, but not fuller"*

Delay-based TCP Congestion Control

Keeping sender-to-receiver pipe "just full enough, but no fuller": keep bottleneck link busy transmitting, but avoid high delays/buffering



$$\text{measured throughput} = \frac{\text{\# bytes sent in last RTT interval}}{RTT_{measured}}$$

Delay-based approach:

- RTT_{min} - minimum observed RTT (uncongested path)
- uncongested throughput with congestion window `cwnd` is $cwnd/RTT_{min}$

if measured throughput "very close" to uncongested throughput
    increase `cwnd` linearly   /* since path **not congested** */

else if measured throughput "far below" uncongested throughput
    decrease `cwnd` linearly   /* since path is **congested** */

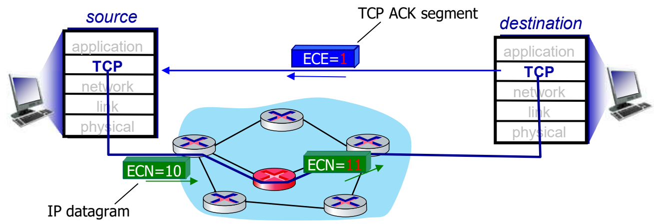
Delay-based TCP Congestion Control

- Congestion control without inducing/forcing loss
- Maximize throughput ("keeping the just pipe full… ") while keeping delay low ("…but not fuller")
- A number of deployed TCPs take a delay-based approach:
  - BBR deployed on Google's (internal) backbone network

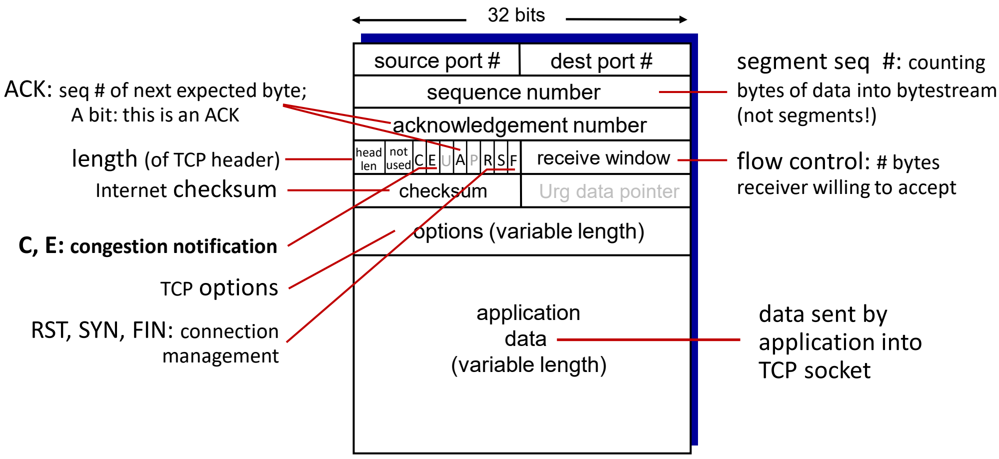
Explicit congestion notification (ECN)

TCP deployments often implement *network-assisted* congestion control:

- two bits in IP header (ToS field) marked *by network router* to indicate congestion
  - *policy* to determine marking chosen by network operator
- congestion indication carried to destination
- destination sets ECE bit on ACK segment to notify sender of congestion
- involves both IP (IP header ECN bit marking) and TCP (TCP header C,E bit marking)



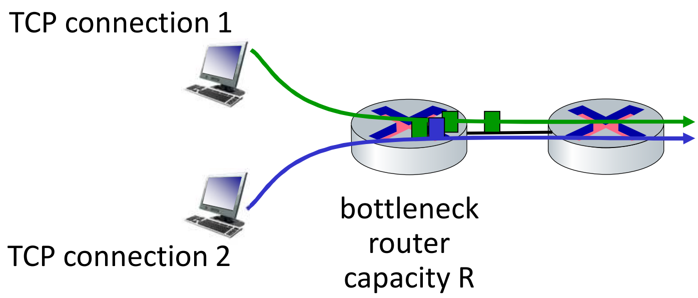
TCP segment structure



TCP Fairness

Fairness goal:

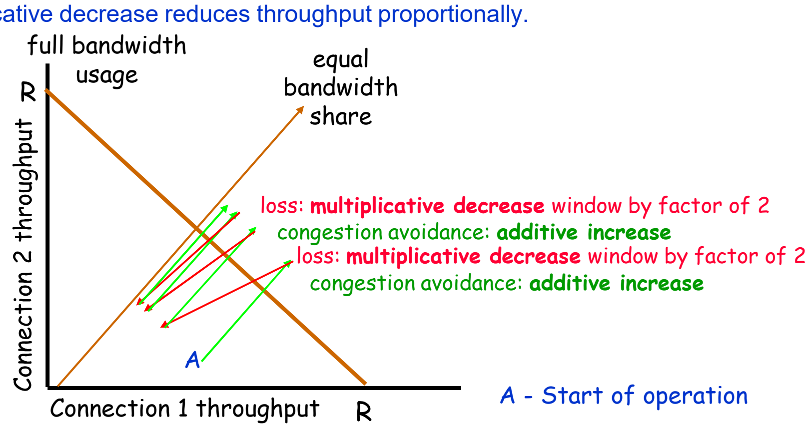
For k TCP sessions sharing the same bottleneck link, of bandwidth R, each should have average rate of R/k.



Is TCP Fair?

Two competing sessions (both have a large amount of data to send, in CA mode):

- Additive increase gives slope of 1, as throughput increases;
- Multiplicative decrease reduces throughput proportionally.



https://media.pearsoncmg.com/aw/ecs_kurose_compnetwork_7/cw/content/interactiveanimations/tcp-congestion/index.html

TÉCNICO LISBOA

Fairness

Fairness and parallel TCP connections

- Application can open parallel connections between 2 hosts;
- Web browsers do this.
- Example: link of rate R supporting 9 connections;
  - New app asks for 1 TCP, gets rate R/10
  - Another new app asks for 10 TCPs, getting R/2 !

Fairness and UDP

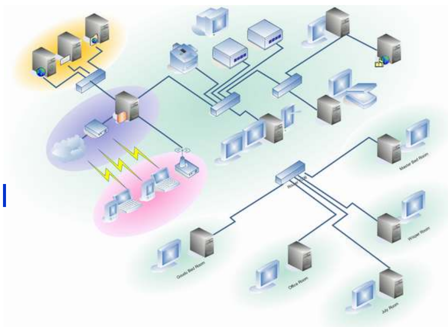
- Multimedia applications often do not use TCP:
  - Do not want rate restricted by congestion control.
- Instead use UDP:
  - Send audio/video at constant rate and tolerate packet loss.

There is no “Internet police” to check the usage of congestion control.

TÉCNICO LISBOA

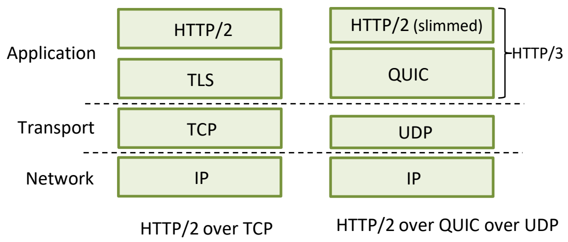
Outline

- Transport-layer services;
- Multiplexing and demultiplexing;
- Connectionless transport: UDP;
- Principles of reliable data transfer;
- Connection-oriented transport: TCP
  - Connection management;
  - Segment structure;
  - Reliable data transfer;
  - Flow control;
- Principles of congestion control
- TCP congestion control
- QUIC



QUIC: Quick UDP Internet Connections

- Application-layer protocol, on top of UDP
  - Increase performance of HTTP
  - Deployed on many Google servers, apps (e.g., Chrome, mobile YouTube app)



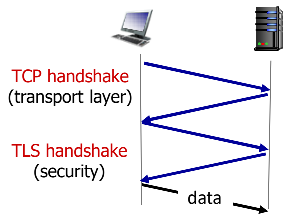
QUIC: Quick UDP Internet Connections

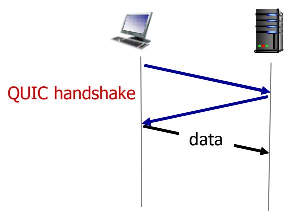
QUIC:

- **Error and congestion control:**
  "Readers familiar with TCP's loss detection and congestion control will find algorithms here that parallel well-known TCP ones."
  [*from QUIC specification*]
- **Connection establishment:**
  Reliability, congestion control, authentication, encryption, state – all established in just one RTT

- Multiple application-level "streams" multiplexed over single QUIC connection
  - Separate reliable data transfer, security
  - Common congestion control

QUIC: Connection Establishment

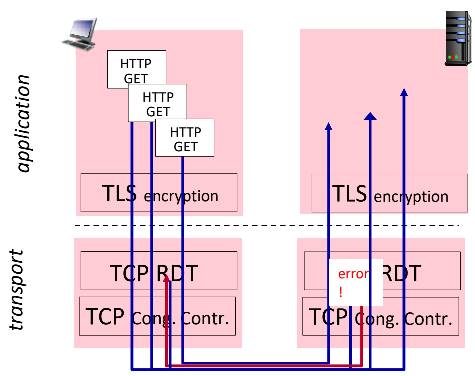




TCP (reliability, congestion control state)
+
TLS (authentication, crypto state)
- 2 serial handshakes

QUIC: reliability, congestion control, authentication, crypto state
- 1 handshake

QUIC Streams: Parallelism, no HOL Blocking



(a) HTTP 1.1

Chapter 3: Summary

- Principles behind transport layer services:
  - Multiplexing and demultiplexing;
  - Reliable data transfer;
  - Flow control;
  - Congestion control.
- Instantiation and implementation in the Internet:
  - UDP;
  - TCP;
  - QUIC.