Communication Services and Security
Network Congestion Control

Cèsar Fernández

Departament d'Informàtica
Universitat de Lleida

Curs 2022 - 2023

Contents

Objectives

- To remember TCP operation procedures
- To detail TCP flow control mechanisms
- To understand TCP related congestion control mechanisms
- To study other congestion control mechanisms

TCP Operation

Introduction

- A **end to end** communication requires:
  - To ensure message delivering
  - As well as their ordering
  - To support different message sizes
  - To allow the receiver to stablish a **flow control** over the message delivery rate
  - To allow multiple applications communicate through the same machine
- Low architecture communication layers don't avoid the following problems:
  - Packet loss
  - No order or duplicate packet delivery
  - A maximum packet size to be delivered
  - No time limit for packet delivery
- To create high layer protocols able to meet those operational conditions is a challenge

TCP Operation

TCP highlights

- Connection oriented
- Full duplex
- Reliable
- Ordered delivery
- Byte stream based. We call **segment** the TCP data unit
- Flow control (avoids receiver overload)
- Congestion control (avoids network overload)
- Multiplexing

TCP Operation

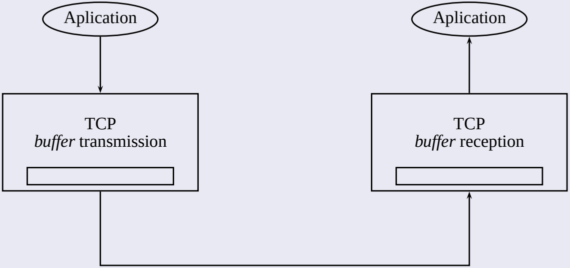
TCP highlights

- TCP implements a **byte stream based sliding window** protocol
- Requires **connection** and **disconnection** phases
- TCP is able to adjust the sliding window size to the RTT (Round Trip Time). As well as the **delay** $\times$ **capacity** product
- TCP assumes a **maximum segment lifetime** (MSL) to 120 seconds
- End to end **acknowledgment**. Routers don't speak TCP

TCP (Transport Control Protocol) highlights

Byte stream oriented sliding window

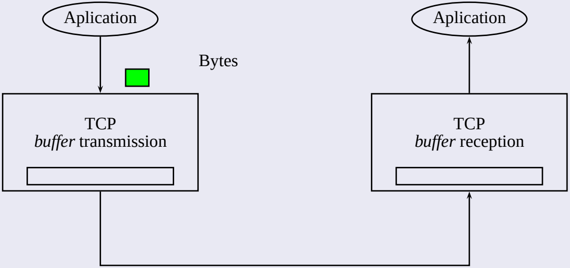
TCP operation outline byte stream based



TCP (Transport Control Protocol) highlights

Byte stream oriented sliding window

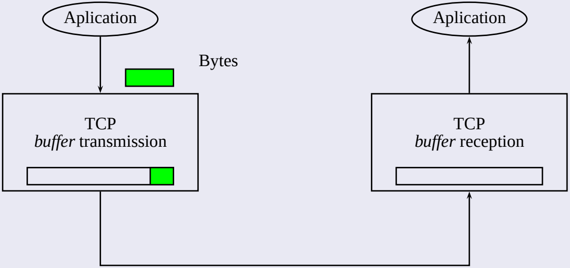
TCP operation outline byte stream based



TCP (Transport Control Protocol) highlights

Byte stream oriented sliding window

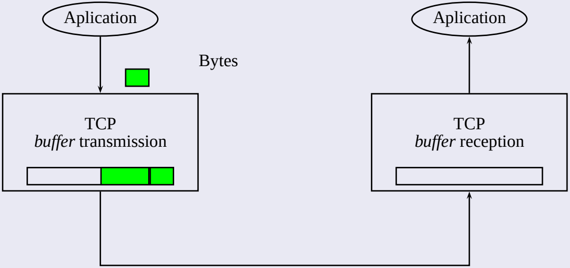
TCP operation outline byte stream based



TCP (Transport Control Protocol) highlights

Byte stream oriented sliding window

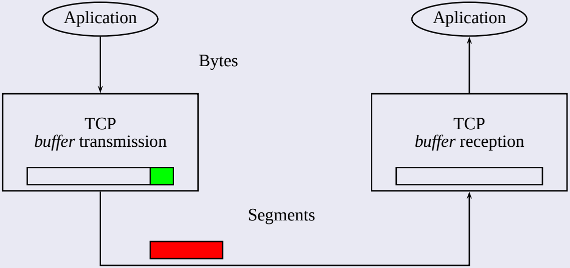
TCP operation outline byte stream based



TCP (Transport Control Protocol) highlights

Byte stream oriented sliding window

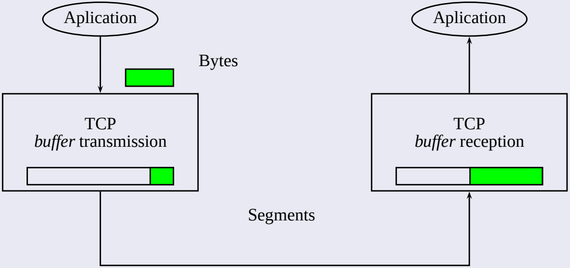
TCP operation outline byte stream based



TCP (Transport Control Protocol) highlights

Byte stream oriented sliding window

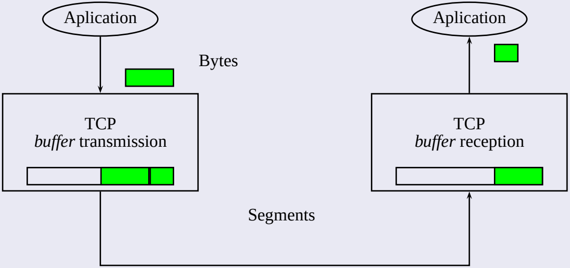
TCP operation outline byte stream based



TCP (Transport Control Protocol) highlights

Byte stream oriented sliding window

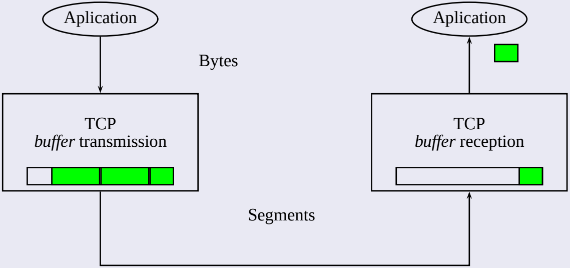
TCP operation outline byte stream based



TCP (Transport Control Protocol) highlights

Byte stream oriented sliding window

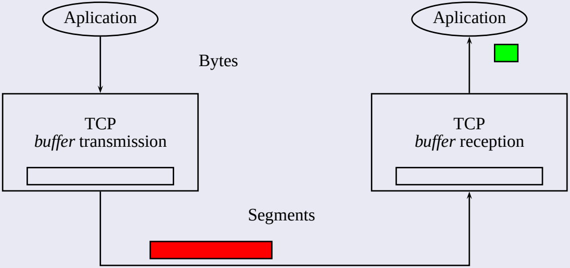
TCP operation outline byte stream based



TCP (Transport Control Protocol) highlights

Byte stream oriented sliding window

TCP operation outline byte stream based



TCP Operation

Maximum Segment Size (MSS)

Usually:

$$MSS = MTU - \mathrm{IP}_{overhead} - \mathrm{TCP}_{overhead}$$

MTU (Maximum Transfer Unit), depends on LAN *Ethernet: 1500 bytes*

TCP Operation

When to transmit ?

TCP triggers a segment transmission when (3 possible causes):

- The transmission buffer reaches *MSS* bytes
- Explicitly indicated (**push**) by the transmission application. I.e. `telnet`
- A timer expires

TCP Operation

Segment format

<table>
<tr><td>0</td><td>4</td><td>10</td><td>16</td><td>31</td></tr>
<tr><td colspan="3">SrcPort</td><td colspan="2">DstPort</td></tr>
<tr><td colspan="5">SequenceNum</td></tr>
<tr><td colspan="5">Acknowledgement</td></tr>
<tr><td>HdrL</td><td>000000</td><td>Flags</td><td colspan="2">AdvertisedWindow</td></tr>
<tr><td colspan="3">Checksum</td><td colspan="2">UrgPtr</td></tr>
<tr><td colspan="5">Options</td></tr>
<tr><td colspan="5">Data</td></tr>
</table>

TCP Operation

Segment format

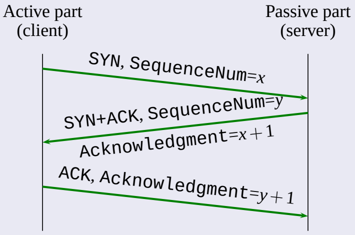
- `Acknowledgment`, `SequenceNum` and `AdvertisedWindow` are involved in flow and congestion control mechanisms. Indicate a value in number of bytes
- `Flags` consists of 6 bits;

`SYN`, `FYN`	To start and to end a connection
`RESET`	Connection aborted
`PUSH`	Indicates to the receiver buffer emptying
`URG`	Such a segment have urgent data, starting at `UrgPtr`
`ACK`	Acknowledgement

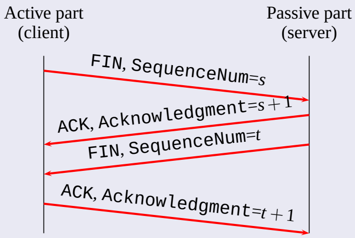
TCP Operation

Setting and ending a connection

- There is always an active and a passive part
- The connection establishment phase is known as **three-way handshake**



Connection setting
Three-way handshake

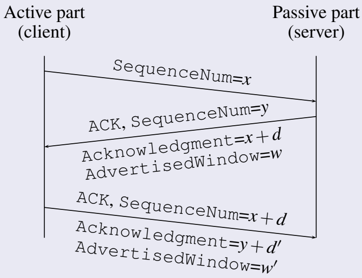


Connection ending

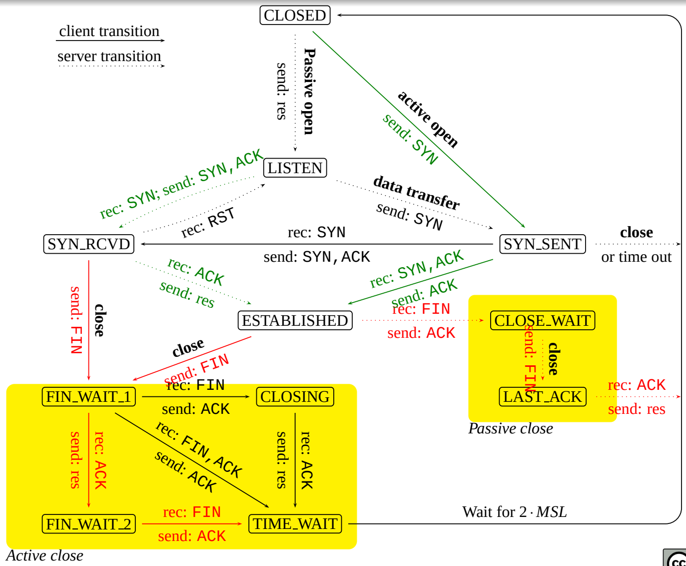
TCP Operation

Setting and ending a connection

- There is always an active and a passive part
- The connection establishment phase is known as **three-way handshake**



TCP Operation



TCP State Diagram

TCP: Flow control

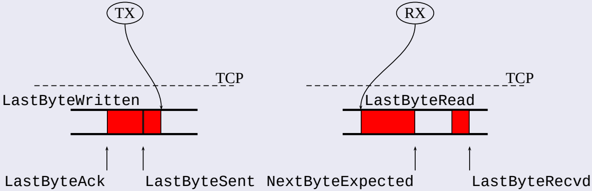
TCP sliding window

The sliding window procedure for TCP allows:

- Segment delivery guarantee
- Segment order guarantee
- Transmission and reception rate synchronization (flow control). Using `AdvertisedWindow`
- Objective: define an `EffectiveWindow`

TCP: Flow control

How delivery and order is guaranteed ?



- 3 pointers for TX and 3 for RX required
- Following inequalities for TX hold (obvious)

  $\texttt{LastByteAck} \le \texttt{LastByteSent}$

  $\texttt{LastByteSent} \le \texttt{LastByteWritten}$

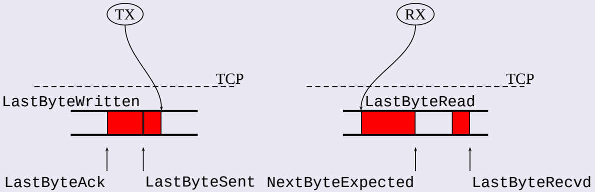
- And for RX

  $\texttt{LastByteRead} < \texttt{NextByteExpected}$

  $\texttt{NextByteExpected} \le \texttt{LastByteRcvd + 1}$

TCP: Flow control

How delivery and order is guaranteed ?



- Both buffers have a maximum size: `MaxSendBuffer` and `MaxRcvBuffer`
- At TX: `LastByteWritten` - `LastByteAck` $\leq$ `MaxSendBuffer`
- At RX: `LastByteRcvd` - `LastByteRead` $\leq$ `MaxRcvBuffer`
- To avoid RX saturation

$$\texttt{AdvertisedWindow} = \texttt{MaxRcvBuffer} - (\texttt{LastByteRcvd} - \texttt{LastByteRead})$$

- TX must collaborate

$$\texttt{EffectiveWindow} = \texttt{AdvertisedWindow} - (\texttt{LastByteSent} - \texttt{LastByteAck})$$

TCP: Flow control

Wrapping the counters

- TCP has 32 bits for `SequenceNum`. Problem: a lost segment must be waited for *MSL* seconds (typically 120 s). During this time lapse, counters must not be wrapped

Rate	Available time
E1 (2048 Kbps)	4.66 hours
Ethernet (10 Mbps)	57.3 minutes
FDDI (100 Mbps)	5.7 minutes
STS-3 (155 Mbps)	3.7 minutes
STS-12 (622 Mbps)	55 seconds

- TCP has 16 bits for `AdvertisedWindow`. Problem: Rx acknowledges up to 65536 KBytes. Assuming a 100 ms *RTT*:

Bandwidth	Delay $\times$ Bandwidth (bytes)
E1 (2048 Kbps)	25.6 KBytes
Ethernet (10 Mbps)	125 KBytes
FDDI (100 Mbps)	1.25 MBytes
STS-3 (155 Mbps)	1.93 MBytes

TCP: Flow control

Adaptive retransmission

TCP has a timer for non-acknowledged segment retransmission. Ideally, such a value should be *RTT*. But Internet delay can be highly variable

Original algorithm (RFC 793)

- *RTT* is measured for each sent segment
- *RTT* is updated according to:

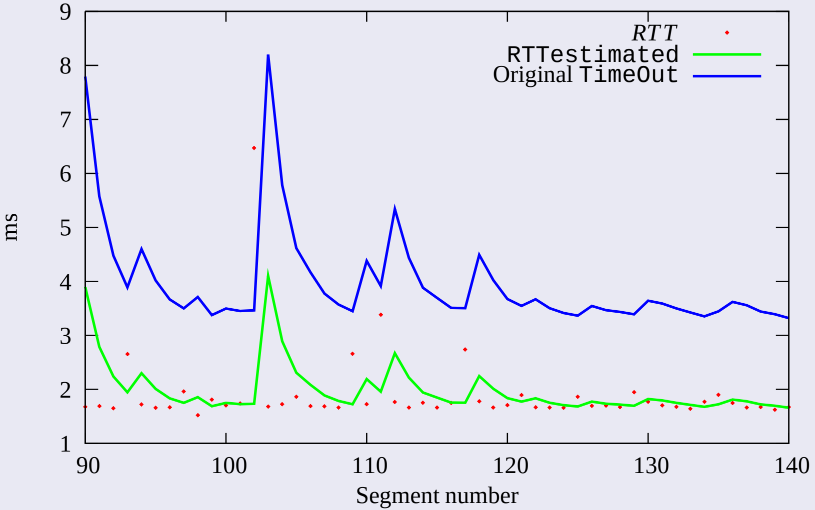
$$\mathtt{RTTestimated} = \alpha \cdot \mathtt{RTTestimated} + (1 - \alpha) \cdot RTT$$

- α between 0.8 and 0.9 acts as a low-pass filter
- We take

$$\mathtt{TimeOut} = 2 \cdot \mathtt{RTTestimated}$$

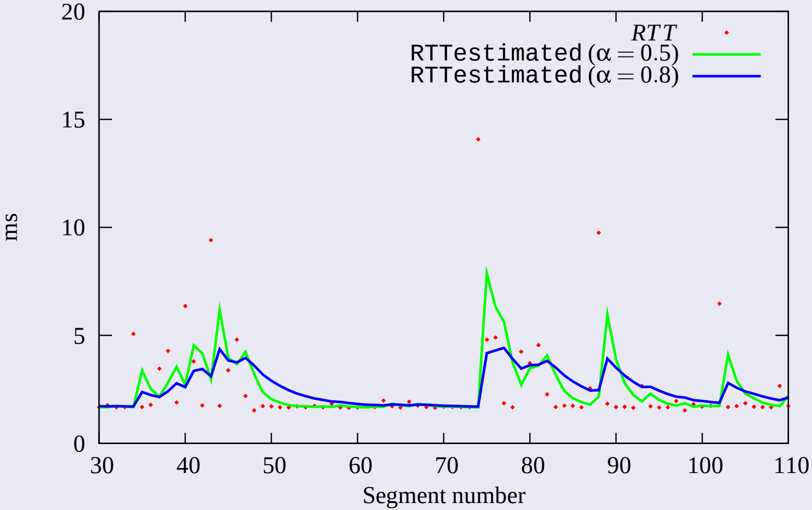
TCP: Flow control

Numerical example ($\alpha = 0.8$)



TCP: Flow control

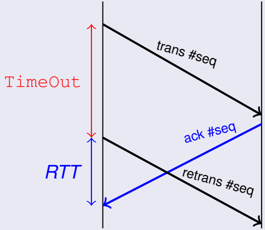
Numerical example



TCP: Flow control

Karn/Partridge algorithm

- Original problem pitfall: *RTT* underestimated when `TimeOut` is shorter than *RTT*



- Solution:
  - Measuring *RTT* only when no retransmission
  - Double `TimeOut` at each retransmission
- As **congestion** is the main cause of retransmissions, one acts aggressively

TCP: Flow control

Jacobson/Karels algorithm

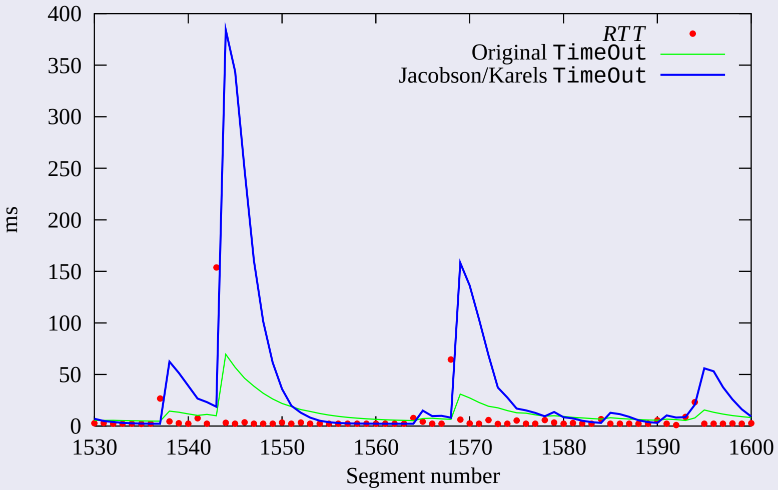
- The **variance** of measured *RTT* is considered
- When variance is **small**, take `RTTestimated` as a better choice than double `TimeOut`
- When variance is **high**, don't rely on `RTTestimated`

$$
\begin{aligned}
\texttt{Diff} &= RTT - \texttt{RTTestimated} \\
\texttt{RTTestimated} &= \texttt{RTTestimated} + \delta \cdot \texttt{Diff} \\
\texttt{Deviation} &= \texttt{Deviation} + \delta \left(|\texttt{Diff}| - \texttt{Deviation} \right) \\
\texttt{TimeOut} &= \mu \cdot \texttt{RTTestimated} + \phi \cdot \texttt{Deviation}
\end{aligned}
$$

typically, $0 < \delta < 1$, $\mu \simeq 1$ and $\phi \simeq 4$

TCP: Flow control

Numerical example ($\delta = 0.5$)



TCP: Flow control

Implementation aspects

- Adaptive algorithms use integer arithmetic
- Counters are scaled by 2^n:
  - $\texttt{t\_srtt\_} = \texttt{RTTestimated} << T\_SRTT\_BITS$
  - $\texttt{t\_rttvar\_} = \texttt{Deviation} << T\_RTTVAR\_BITS$
  - $T\_SRTT\_BITS = 3$
  - $T\_RTTVAR\_BITS = 2$
- Default measure accuracy is 0.5 seconds

TCP: Flow control

Scaling `RTTestimated`

- Remember:

$$\texttt{RTTestimated} = \alpha \cdot \texttt{RTTestimated} + (1-\alpha) \cdot RTT$$

- Taking $\alpha = 7/8$, results:

$$\texttt{RTTestimated} = 7/8 \cdot \texttt{RTTestimated} + 1/8 \cdot RTT$$

- From code (`ns-2.35/tcp/tcp-rfc793edu.cc`)

$$\begin{aligned} \texttt{delta} &= RTT - (\texttt{t\_srtt\_} >> T\_SRTT\_BITS) \\ \texttt{t\_srtt\_} &+= \texttt{delta} \end{aligned}$$

Turns to be:

$$\texttt{RTTestimated} = \texttt{t\_srtt\_} >> T\_SRTT\_BITS$$

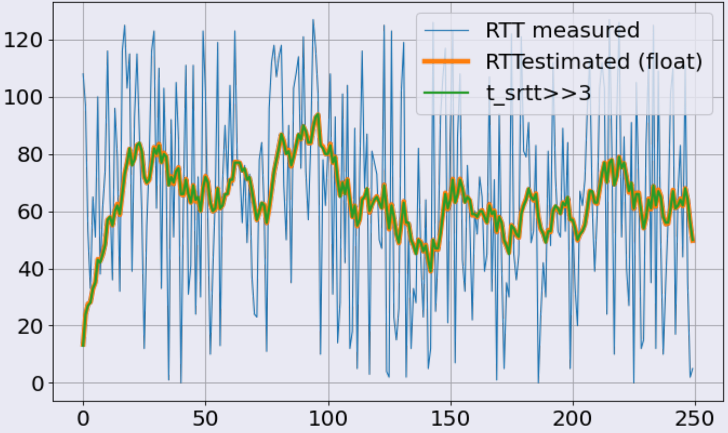
TCP: Flow control

Scaling `RTTestimated`. Numerical example

`RTT`	`t_srtt>>3`	`RTTestimated` (as float)
108	13	13.5
97	24	23.9
53	27	27.6
33	28	28.2
65	33	32.8
51	35	35.1
100	43	43.2
38	42	42.6
61	45	44.9
74	48	48.5
116	57	57.0
64	58	57.8
36	55	55.1
96	60	60.2
79	62	62.6
32	59	58.7

TCP: Flow control

Scaling `RTTestimated`. Numerical example



Codi: rtt.py

TCP: Flow control

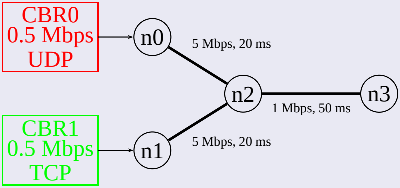
Scaling `Deviation`

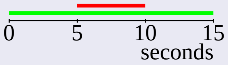
- Rembember Jacobson/Karels
- Same idea, but 1/4 factor scale
- Question. Specify the operations

TCP: Flow control

Simulation example

- Topology:



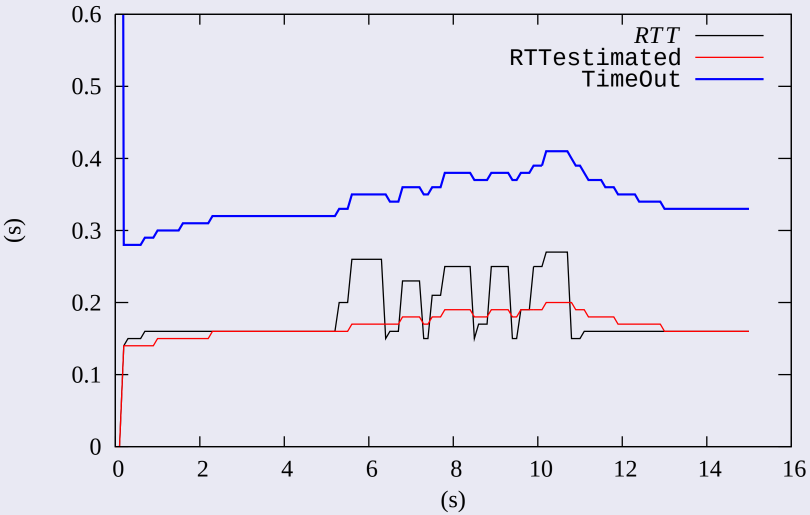


- 3 simulations (NS code: sim1.tcl), none uses exponential backoff:
  1. Original adaptive algorithm (RFC793)
  2. Including Karn/Partridge
  3. Including Jacobson/Karels time estimation
- Summary results:

Simulation	Lost segments
1	57
2	54
3	48

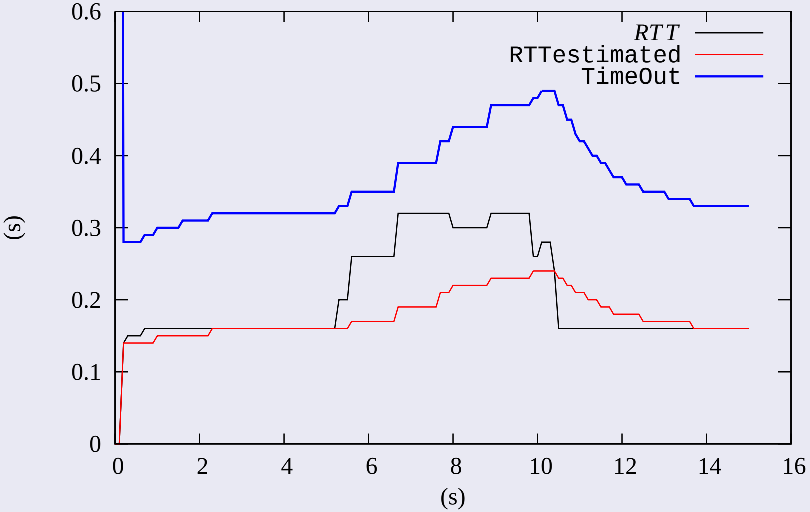
TCP: Flow control

Simulated example: (Original, `tcpTick`=10ms)



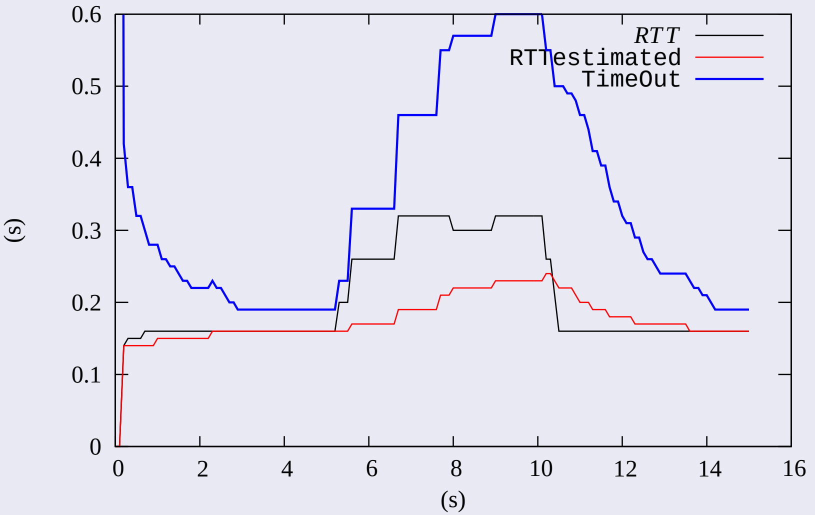
TCP: Flow control

Simulated example: (Karn/Partridge, `tcpTick`=10ms)



TCP: Flow control

Simulated example: (Jacobson/Karels, `tcpTick`=10ms)



Congestion control

Introduction

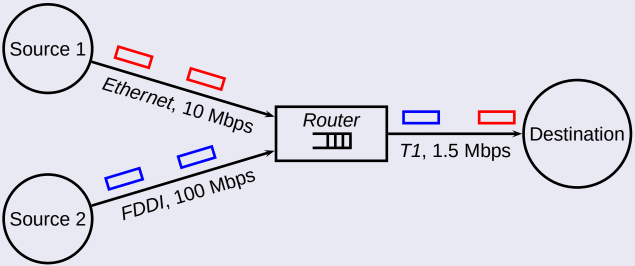
- **Resource allocation:** set of procedures that allows a network element (usually a **router**) to decide about how to assign the available resources (**bandwidth** and **buffers size**)
- **Congestion control:** set of procedures devoted to avoid or correct congestion at the network. Some congestion control mechanisms also includes resource allocation policies
- What is the difference between congestion control and **flow control** ?
    - Congestion control operates at network and/or transmitter
    - Flow control does the same at receiver

Congestion control

Network model

Highlight 3 aspects of the network model before describing resource allocation:

1. **Packet switching.** Links and routers deal with discrete information units (packets)



Analogy: medium access control mechanisms (local area networks) and resource allocation mechanisms (between links and routers)

Congestion control

Network model

Highlight 3 aspects of the network model before describing resource allocation:

2. **Non-connected traffic flow**. We'll focus on non-connected traffic. In a connected network model, resources may be allocated before transmission (at connection phase)
   Even though, routers can distinguish among non-connected traffic flows. Such an identification can be:
   - Implicit, determined by addresses
   - Explicit, determined by the source

Congestion control

Network model

Highlight 3 aspects of the network model before describing resource allocation:

3. **Service model**. One can ask to the network:
    - A given **quality of service** (QoS), telling the source the parameters to be accomplished (bandwidth, delay, . . .)
    - Nothing (**best-effort**)

Congestion control

Taxonomy

According to:

- **Who** takes the decisions. **Routers** or **hosts**. May be both
- **When** are taken
    - Hosts made **reservations** and routers decide
    - **Implicit feedback** from hosts or **explicit** (router decided)
- **How** decisions are taken. Based on **window** (as TCP) or **rate sharing**

Congestion control

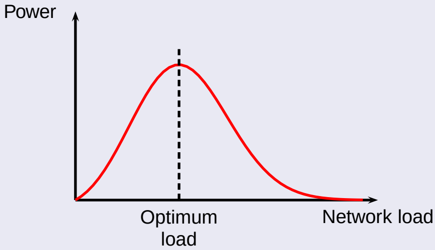
Performance indicators

One can define some indicators in order to determine the **efficiency** and the **fairness** of a given resource allocation method

- **Efficiency**. Defined as

$$\text{Power} = \frac{\text{Throughput}}{\text{Delay}}$$

Objective: maximize it



Congestion control

Performance indicators

- **Fairness**. Assuming n receiving streams at $x_1, x_2, \cdots, x_n$ bps, one can assign a fairness index as follows:

$$f(x_1, x_2, \cdots, x_n) = \frac{\left(\sum_{i=1}^{n} x_i\right)^2}{n \sum_{i=1}^{n} x_i^2}$$

Note that $0 < f(x_1, x_2, \cdots, x_n) \leq 1$. Fairness is maximum when $x_1 = x_2 = \cdots = x_n$
Assuming $n - 1$ identical streams (1 bps) and a stream at $1 + \Delta$ bps,

$$f(1, 1, \cdots, 1 + \Delta) = \frac{n^2 + 2n\Delta + \Delta^2}{n^2 + 2n\Delta + n\Delta^2}$$

Denominator takes is $(n-1)\Delta^2$ units larger than numerator

Service policies

Introduction

Unregarding resource allocation mechanisms, a router must employ a **queue policy** (or service policy) that decides:

- Which packets must be transmitted
- Which packets must be dropped if queue overflows

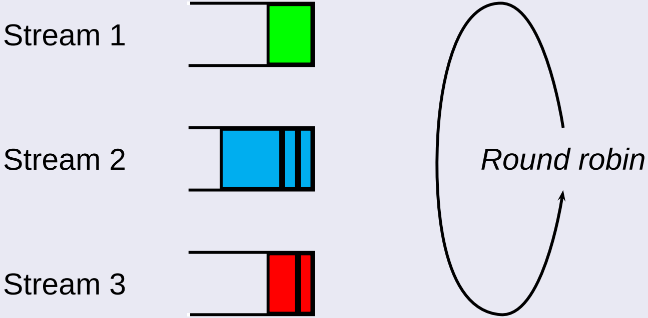
First In First Out (FIFO)

- A **drop tail** procedure may be adhered
- Implies **external** (to the router) congestion control and resource allocation (e.g. at TCP layer)
- **Variations**: to tag IP packets with a priority field (TOS) and to implement multiple queues with different priority
- **Problems**: high priority queues may block lower priorities transmission

Service policies

Fair Queuing (FQ)

- Assuming that congestion control and resource allocation relies on TCP, the problem of having **greedy processes** (e.g. applications sending on UDP) must be overcome
- FQ solves the problem having multiple queues assigned to different streams, being served cyclically (**round robin**)



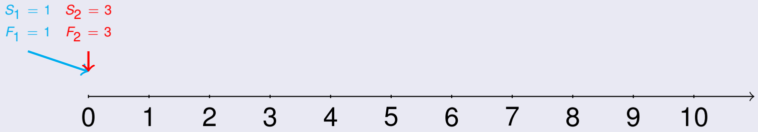
Service policies

Fair Queuing (FQ)

- **Problem**: Different packet sizes. How they are served ?
  1. Packets arrive in sequence. A_i arrival time of the i_th packet. S_i is the packet length
  2. F_i: estimated time to finish. When an arriving packet founds the server free, $F_i = A_i + S_i$. When the server is busy, $F_i = \max\left(F', A_i\right) + S_i$
  3. Being F' the **computed (virtual)** finish time (not the real one) of the packet being served
  4. The packet with the lower F_i is transmitted first

Service policies

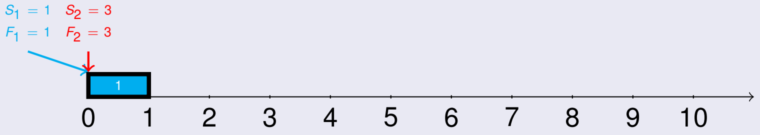
FQ: Two streams example



- $F_1 = 0 + 1 = 1$
- $F_2 = 0 + 3 = 3$

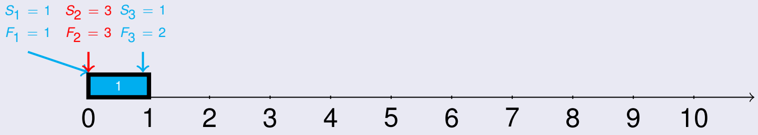
Service policies

FQ: Two streams example



Service policies

FQ: Two streams example



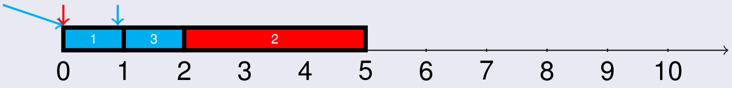
- $F_3 = \max(F_1, 0.9) + 1 = 2$

Service policies

FQ: Two streams example

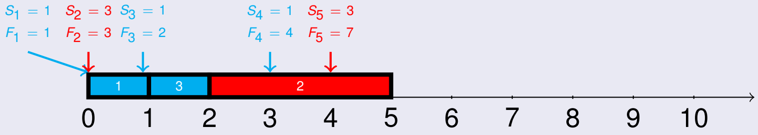
$S_1 = 1 \quad S_2 = 3 \quad S_3 = 1$

$F_1 = 1 \quad F_2 = 3 \quad F_3 = 2$



Service policies

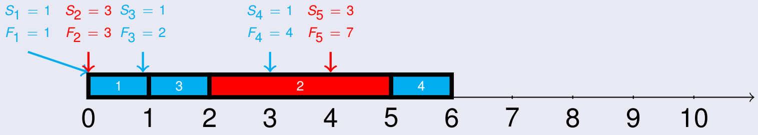
FQ: Two streams example



- $F_4 = \max(F_2, 3) + 1 = 4$
- $F_5 = \max(F_2, 4) + 3 = 7$

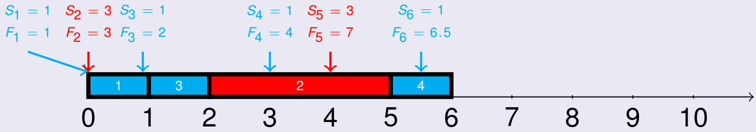
Service policies

FQ: Two streams example



Service policies

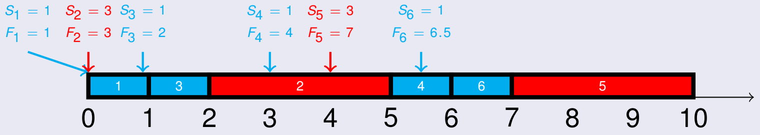
FQ: Two streams example



- $F_6 = \max(F_4, 5.5) + 1 = 6.5$

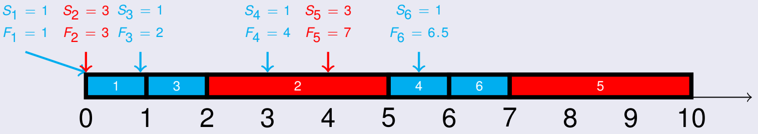
Service policies

FQ: Two streams example



Service policies

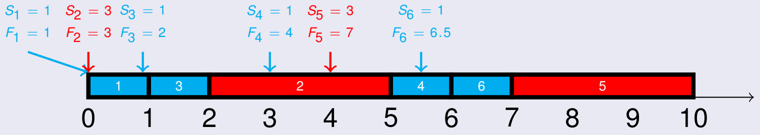
FQ: Two streams example



- The system never remains idle when queues are not empty
- When n active streams, none has more than $\frac{1}{n}$th of the network capacity
- When a stream becomes idle, the released amount of bandwidth is shared among the rest

Service policies

FQ: Two streams example



- One can think on a **weighted** version, **Weighted FQ** (WFQ) doing

$$F_i = \max(F', A_i) + S_i \cdot r_j$$

  being
  - F' the finish time for the current packet
  - $\frac{1}{r_j}$ the bandwidth fraction assigned to queue j. The follow holds: $\sum_j \frac{1}{r_j} \leq 1$

TCP: Congestion control

Introduction

- TCP send segments to the network, without reservation, looking at their behavior and reacting
- Don't depending on routers queue policies
- Mechanisms introduced by Van Jacobson (late 80s), when Internet begins to collapse
- Outline:
    - TCP computes how many segments **fits** inside a non congested network
    - For each received `ACK`, a new segment is sent
    - Adaptive process. Network load changes along time
    - Different strategies. Non exclusive

TCP: Congestion control

Congestion Window

- TCP defines a new variable `CongestionWindow (cwnd)`. Equivalent to `AdvertisedWindow` (flow control), but at transmitter end
- Effective window, `EffectiveWindow` is defined again as follows:

$$\begin{aligned} \texttt{MaxWindow} &= \text{MIN}(\texttt{cwnd}, \texttt{AdvertisedWindow}) \\ \texttt{EffectiveWindow} &= \texttt{MaxWindow} - (\texttt{LastByteSent} - \texttt{LastByteAck}) \end{aligned}$$

- How `CongestionWindow` is computed determines the congestion control mechanism: Original/nil, Additive increase/Multiplicative decrease, Slow Start, ...

TCP: Congestion control

Original (no control)

- `CongestionWindow` maximum value, `CWMAX`
- `cwnd` initial value (usually 1 MSS), `cwini`
- When an `ACK` is received,

  `cwnd = CWMAX`

- When **TimeOut**,

  `cwnd = cwini`

TCP: Congestion control

Additive increase/Multiplicative decrease

- `CWMAX`, `CongestionWindow` absolute maximum value,
- `cwini`, `cwnd` initial value (usually 1 MSS)
- `cwmax`, `cwnd` maximum value (initially set to `CWMAX`)
- When an `ACK` is received,
  - If `cwnd < cwmax`

    `cwnd = cwmax`
  - Otherwise

    `cwnd += MSS/cwnd`

    `cwmax=MIN(CWMAX,cwnd)`
- When *TimeOut*,

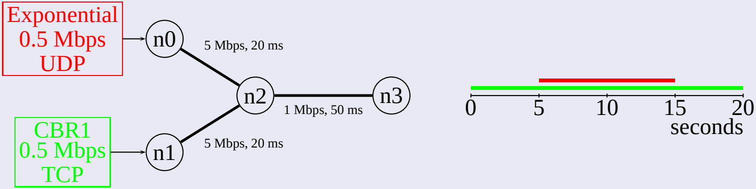
  `cwnd = cwini`

  `cwmax = MAX(cwini,cwmax/2)`
- Idea: `TimeOut` → retransmissions→ worst congestion → fast reaction
- *RTT* must be estimated accurately → adaptive retransmission

TCP: Congestion control

Example (simulation)

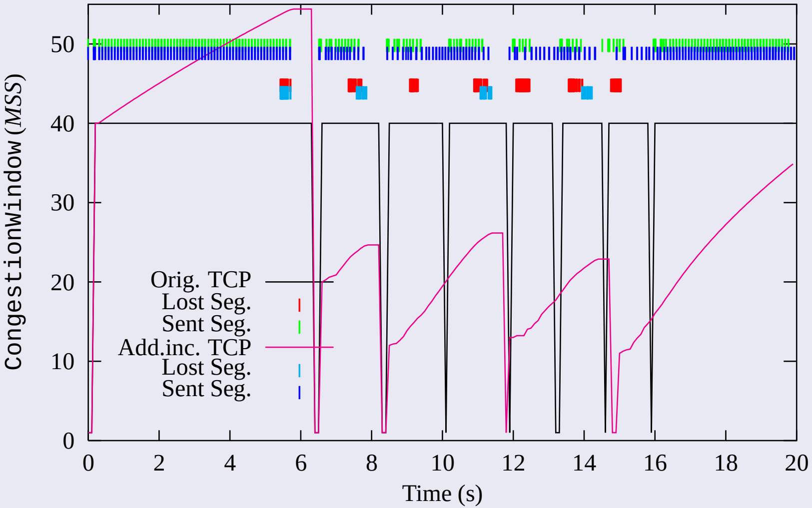
- Topology:



- 2 simulations (cw1.tcl):
  - TCP original congestion control (286 lost segments at router **n2**)
  - TCP Additive increase/Multiplicative decrease (91 loses)

TCP: Congestion control

Simulation example ($\texttt{CWMAX} = 40 \cdot MSS$)



TCP: Congestion control

Slow start

- Algorithm
  - `CWMAX`, `CongestionWindow` absolute maximum value,
  - `cwini`, `cwnd` initial value (usually 1 MSS)
  - `cwmax`, `cwnd` maximum value (initially set to `CWMAX`)
  - When an `ACK` is received,
    - If `cwnd < cwmax` (exponential increase)

      `cwnd += MSS`
    - Otherwise (linear increase)

      `cwnd += MSS/cwnd`

      `cwmax=MIN(CWMAX,cwnd)`
- When TimeOut,

  `cwnd = cwini`

  `cwmax =MAX(cwini, cwmax/2)`

TCP: Congestion control

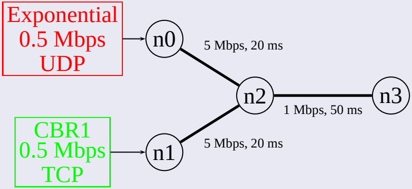
Slow start

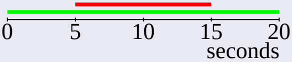
- Less aggressive than Additive increase/Multiplicative decrease
- Slow start performs on 2 different situations:
  - At starting. TCP unknowns the link capacity. A maximum exponential increase may be established
  - After a TimeOut. At this point, TCP knows about the maximum `CongestionWindow` reached. So, exponential increase up to half the maximum reached so far

TCP: Congestion control

Example (simulation)

- Topology:

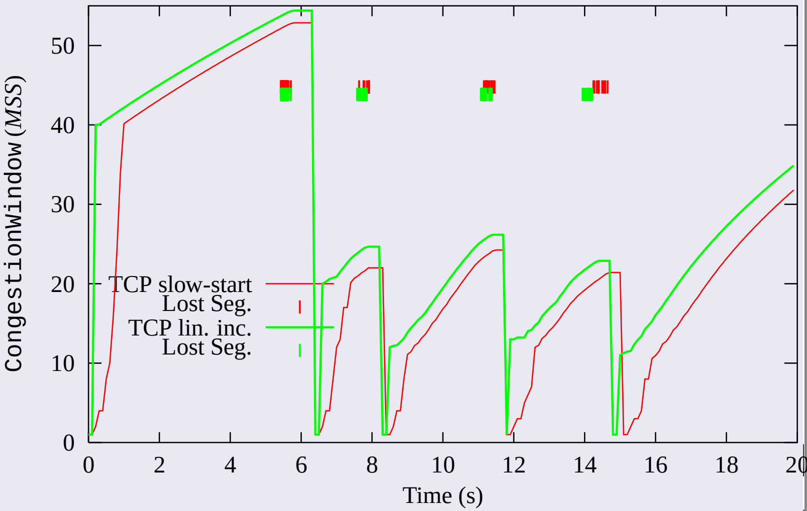




- 3 simulations (cw1.tcl):
  - TCP original congestion control (286 lost segments at router **n2**)
  - TCP Additive increase/Multiplicative decrease (91 loses)
  - TCP Slow Start (63 loses)

TCP: Congestion control

Slow start: simulated example (CWMAX$= 40 \cdot MSS$)



TCP: Congestion control

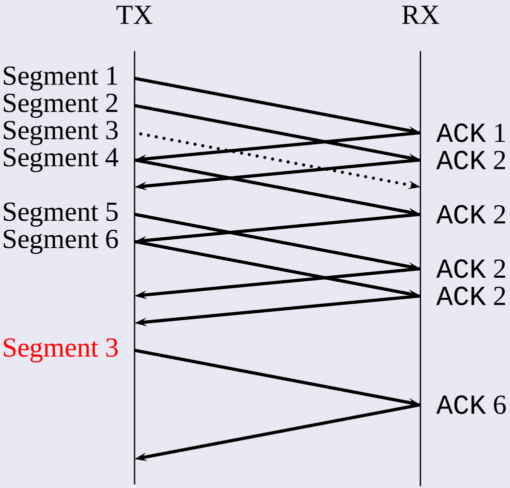
Fast retransmit

Helps TCP with retransmissions before TimeOut

- When receiver gets a segment out of sequence → sends an `ACK` of the last in-sequence segment
- Transmitter will receive a duplicated `ACK`
- When 3 duplicated `ACK`s in a row
  - re-sent the lost segment
  - `cwnd=cwini`, `cwmax=cwmax/2` and slow-start
- Around **half** of the total TimeOut saved → improving throughput by 20%

TCP: Congestion control

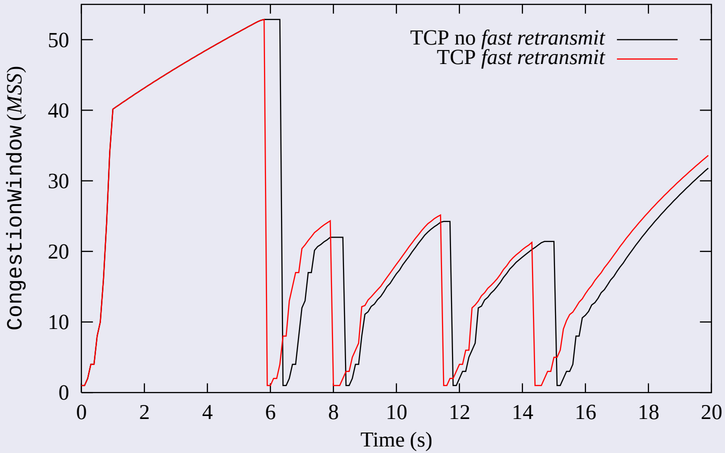
Fast retransmit



TCP: Congestion control

Fast retransmit: simulated example

Same simulation scenario as before



Fast retransmit gets 8% more transmissions

TCP: Congestion control

Fast recovery

Another improvement

- When third duplicate `ACK`s arrive, `CongestionWindow` reduces to half
- At this point, increase linearly, avoiding slow start phase
- Even though, Timeouts occur, do slow start (multiplicative increase)
- Do `cwmax=cwmax/2` when third duplicate and, also, if TimeOut is produced

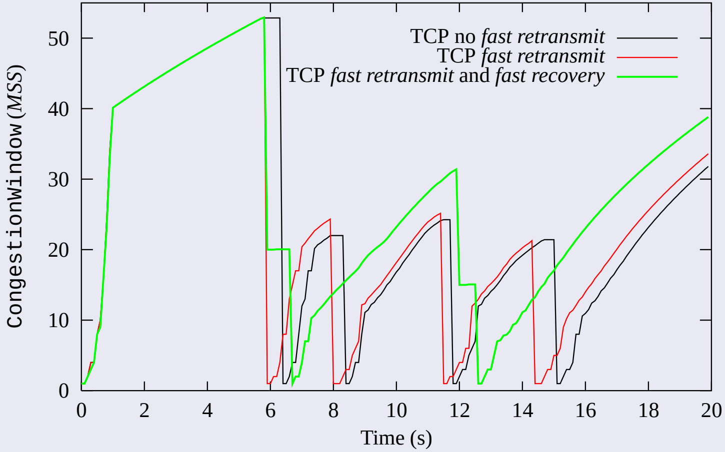
Next plots show performance for the 3 options:

- 8% more transmissions with fast retransmit
- 11% more transmissions with *fast retransmit* and *fast recovery* (together)
- Examples source code

TCP: Congestion control

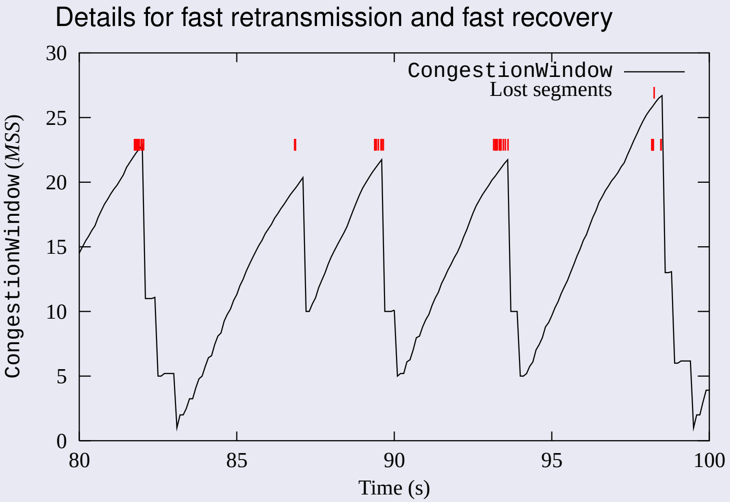
Simulation results

Same simulation scenario than before



TCP: Congestion control

Simulation results



TCP: Congestion control

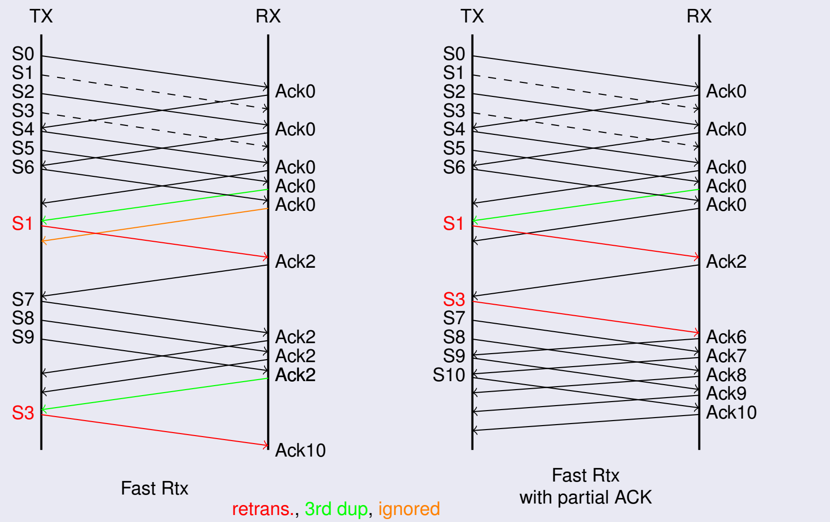
Partial ACKs

Fast retransmission improvement

- Only changes the transmitter side
- Enhaces two or more losses in the same congestion window
- After the first third duplicated ack, sender retansmit on a single duplicated

TCP: Congestion control

Fast Rtx with and without partial acks example



TCP: Congestion control

TCP Reno

- Includes: Jacobson-Karels RTT estimator
- Fast retransmit (without partial ACKs) and fast recovery
- TimeOut below 1” are rounded to 1” (RFC 6298)
- In Fast Recovery phase, `cwnd` substituted by Estimated FlightSize (EFS):
  - Initially, `cwnd` = EFS
  - EFS is decremented for every duplicated ACK
- For retransmission, note point (5.3) of RFC 6298

When an ACK is received that acknowledges new data, restart the retransmission timer so that it will expire after RTO seconds (RFC 6298)

TCP NewReno

- Adds partial ACKs to TCP Reno
- RFC 3782

TCP: Congestion control

TCP Reno example (cwmax=10, $T_{segment}$=33 ms)

	cw	EFS	Buffer	Time	event
Recovery phase	10	10	240:249	43.93984	Sent 249
		9		45.14784	DUP Ack 239
		8		45.51584	DUP Ack 239
	1			45.81984	TO resent 240
		7		45.91744	DUP Ack (3) 239
					Resent 241
	2		241:249	47.52864	Ack 240
					Resent 242
	3		242:249	47.79584	Ack 241
					Resent 243,244
	4		243:249	48.776	Ack 242
					Resent 245,246
	5		244:249	49.073864	Ack 243
					Resent 247,248
	5.2		245:249	49.273864	Ack 244
					Resent 249
	5.4		246:249	50.213064	Ack 245
			246-250		Sent 250
	5.6		250	50.379464	Ack 249
Recovery phase		5	250:254		Sent 251,252,253,254
		4		50.545864	DUP Ack 249
		5	250:255		Sent 255
		4		50.712264	DUP Ack 249
		5	250:256		Sent 256
	5.8		251:256	51.460424	Ack 250
	5.99		252:256	51.626824	Ack 251
	6.16		253:256	51.826824	Ack 252
			254:257		Sent 257, 258

Congestion: Other mechanisms

Some facts

- TCP congestion control mechanisms are **reactive**
- Let's see how **preventive** mechanisms work (**Congestion avoidance**)
  - **Random Early Detection** (RED)
  - **Source** based

Random Early Detection

Outline

- Routers monitor their queues. Just before congestion, segments are bit marked or **selectively** dropped out
- Transmitter being informed:
  - Timeouts
  - Duplicate `ACK`s
- Objective: decrease `CongestionWindow` before congestion by dropping out some segments

Random Early Detection

Queue monitoring

- RED periodically computes the queue average length ($\texttt{AvgLen}$) by filtering

$$\texttt{AvgLen} = (1-\alpha)\cdot\texttt{AvgLen} + \alpha\cdot\texttt{SampleLen}$$

being $\texttt{SampleLen}$ the sampled length and $0 \leq \alpha \leq 1$

- 2 thresholds defined, $\texttt{MinTh}$ i $\texttt{MaxTh}$, and the following policy

If $\texttt{AvgLen} \leq \texttt{MinTh}$

$\rightarrow$ Put the segment in queue

If $\texttt{MinTh} < \texttt{AvgLen} < \texttt{MaxTh}$

$\rightarrow$ Compute probability P

$\rightarrow$ Drop out segment with probability P

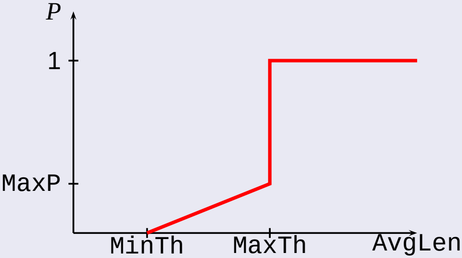
If $\texttt{AvgLen} \geq \texttt{MaxTh}$

$\rightarrow$ Drop out segment

Random Early Detection

Queue monitoring

- P computed as:



Random Early Detection

Problems and improvements

- Segments sometimes dropped out at bursts
- No more than a segment in less than *RTT* should be withdrawn:
  - One segment is enough to stop increasing the congestion window (with fast retransmit)
  - Dropping out more than one segment could lead Transmitter to slow start
- Following improving proposed:

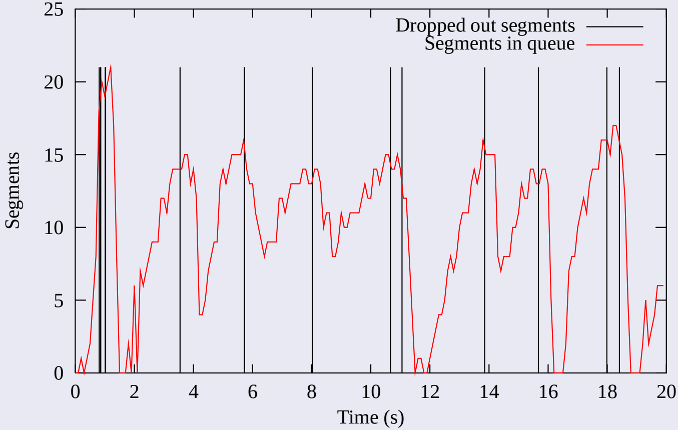
$$
\begin{aligned}
P' &= \mathtt{MaxP}\,\frac{\mathtt{AvgLen} - \mathtt{MinTh}}{\mathtt{MaxTh} - \mathtt{MinTh}} \\
P &= \frac{P'}{1 - \mathtt{counter} \cdot P'}
\end{aligned}
$$

being `counter` the number of segments enqueued from the last drop out

Random Early Detection

Example

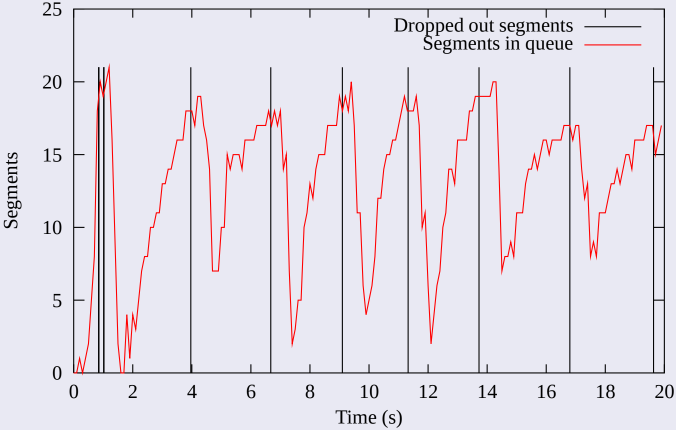
RED **without** `counter`. `MinTh=10`, `MaxTh`=20, `MaxP`=0.05
Number of transmissions = 1506. Dropped out segments = 42



Random Early Detection

Example

RED **with** `counter`. `MinTh=10`, `MaxTh`=20, `MaxP`=0.05
Number of transmissions = 1514. Dropped out segments = 38



red.tcl

Random Early Detection

Some thoughts

- RED requires a larger queue than `MaxTh` in order to absorb variations between following measurements
- When queue is full, segments are dropped (**drop tail**)
- 100 ms could be a good time between samples. It has no sense take more than a sample inside the same *RTT*

Source-Based Congestion Avoidance

Introduction

- Algorithms detecting congestion from the source end
- Different aspects:
  - How they detect when congestion starts: *RTT* increases, throughput changes, ...
  - How they reacts
- TCP implementations:
  - **TCP Tahoe** (*BSD Network Release 1*). It is a TCP with Jacobson/Karels algorithm including the before explained congestion control mechanisms (but fast recovery)
  - **TCP Reno** (*BNR 2*) includes *fast recovery* and more (**delayed** `ACK`)
  - **TCP NewReno** (*RFC 3782*) adds *partial ACKs* and some changes computing `cw`
  - **TCP Vegas**. Adds source-based congestion avoidance

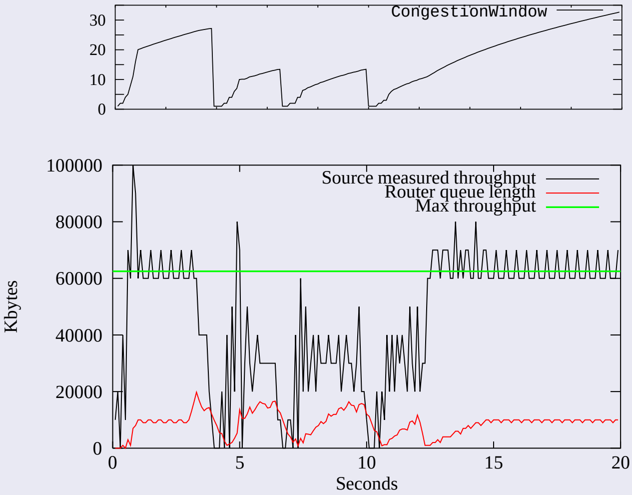
Source-Based Congestion Avoidance

TCP Vegas

- Based on the detection of throughput saturation. Following facts occur:
    - While `CongestionWindow` increases, measured throughput by the transmitter keeps stable
    - Routers start to be saturated

Source-Based Congestion Avoidance

TCP Vegas



Source-Based Congestion Avoidance

TCP Vegas: procedure

- `BaseRTT` defined as *RTT* without congestion. In practice will be the minimum *RTT* measured (generally when connection starts)

$$\texttt{ExpectedThroughput} = \frac{\texttt{CongestionWindow}}{\texttt{BaseRTT}}$$

  Assume that `CongestionWindow` is the number of bits in transit
- Current throughput (`CurrentThroughput`) is measured as follows:
  - Compute the time between a sent segment and its corresponding acknowledgment
  - Compute the amount of data sent in such an elapsed time

  `CurrentThroughput` is sampled at each *RTT*

Source-Based Congestion Avoidance

TCP Vegas: procedure

- We take

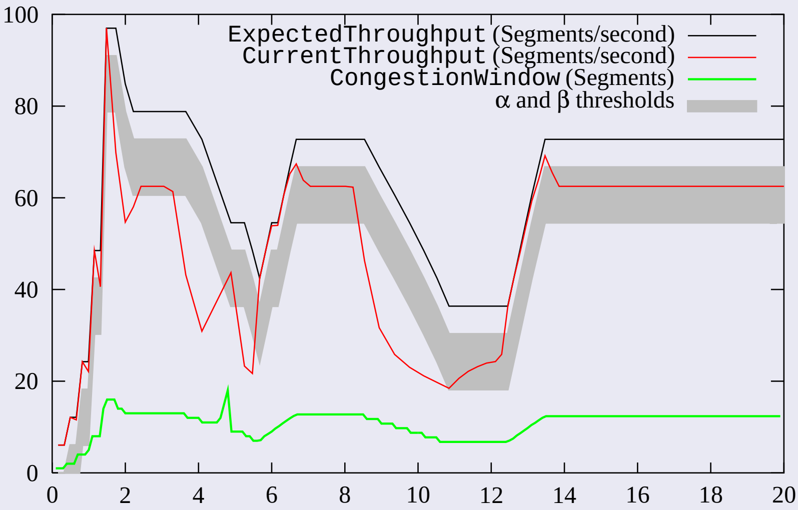
$$\mathtt{Dif} = \mathtt{ExpectedThroughput} - \mathtt{CurrentThroughput}$$

By definition, $\mathtt{Dif} \geq 0$. Otherwise, $\mathtt{BaseRTT}$ must be updated. 2 thresholds are defined, α and β, s.t. $\alpha < \beta$

- If ($\mathtt{Dif} < \alpha$)
  `CongestionWindow` linearly increased in next *RTT*
- If ($\mathtt{Dif} > \beta$)
  `CongestionWindow` linearly decreased
- If ($\alpha < \mathtt{Dif} < \beta$)
  `CongestionWindow` unchanged

Source-Based Congestion Avoidance

TCP Vegas: example ($\alpha = 6$ segments/s, $\beta = 18$ segments/s)



Bibliography

- Internetworking with TCP/IP: Volume I. *Douglas E. Comer*. Prentice Hall, 1991
- TCP/IP Illustrated, Volume I. *William R. Stevens*. Addison-Wesley, 1994
- *TCP Protocol Specification, RFC 793*. 1981
- The network simulator ns-2. ns-2 Wiki