Streaming Multimediale

The Micro Transport Protocol

Simone Basso

Nexa Center for Internet & Society
Dept. Of computer and control Engineering
Politecnico di Torino

Roadmap:

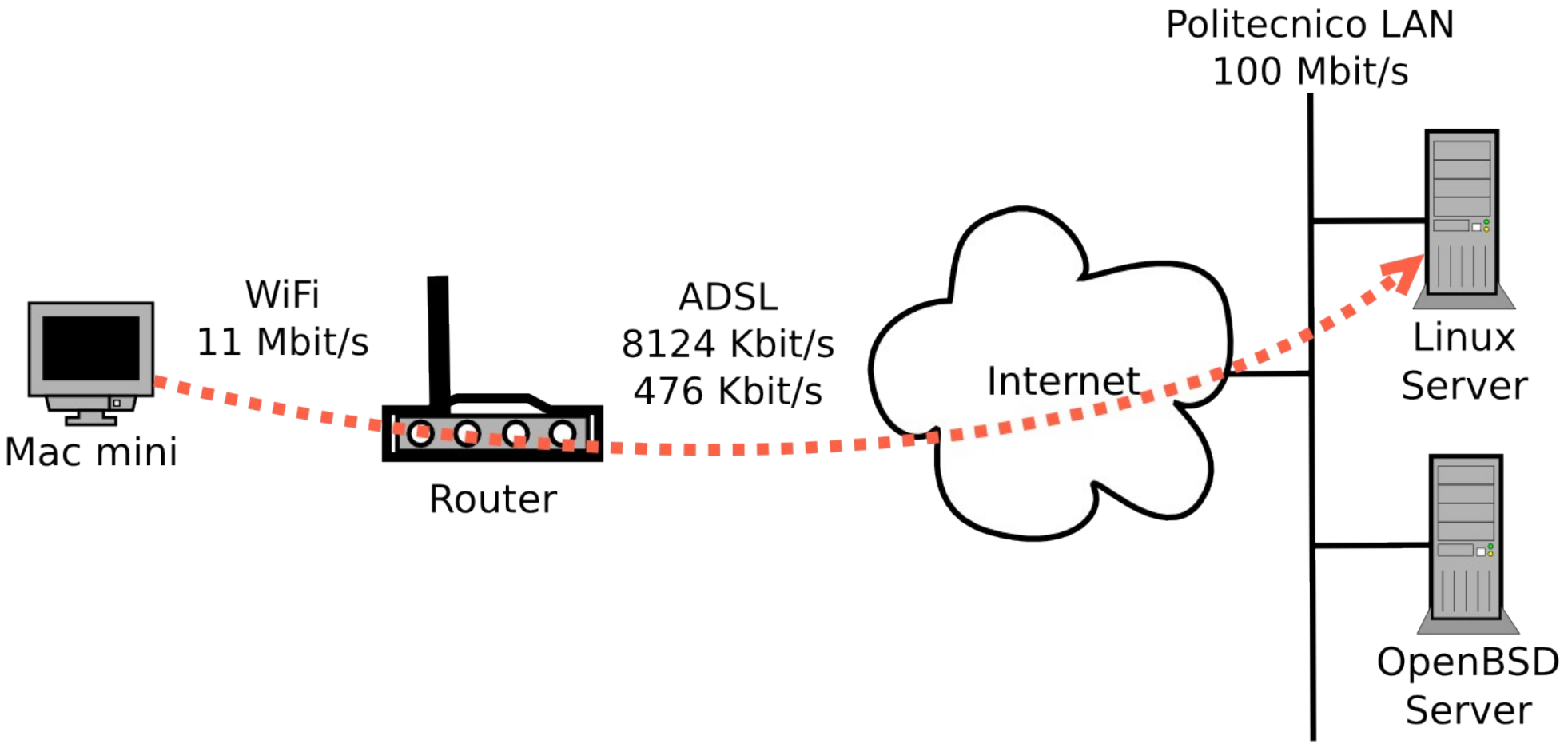
1. understand the bufferbloat;

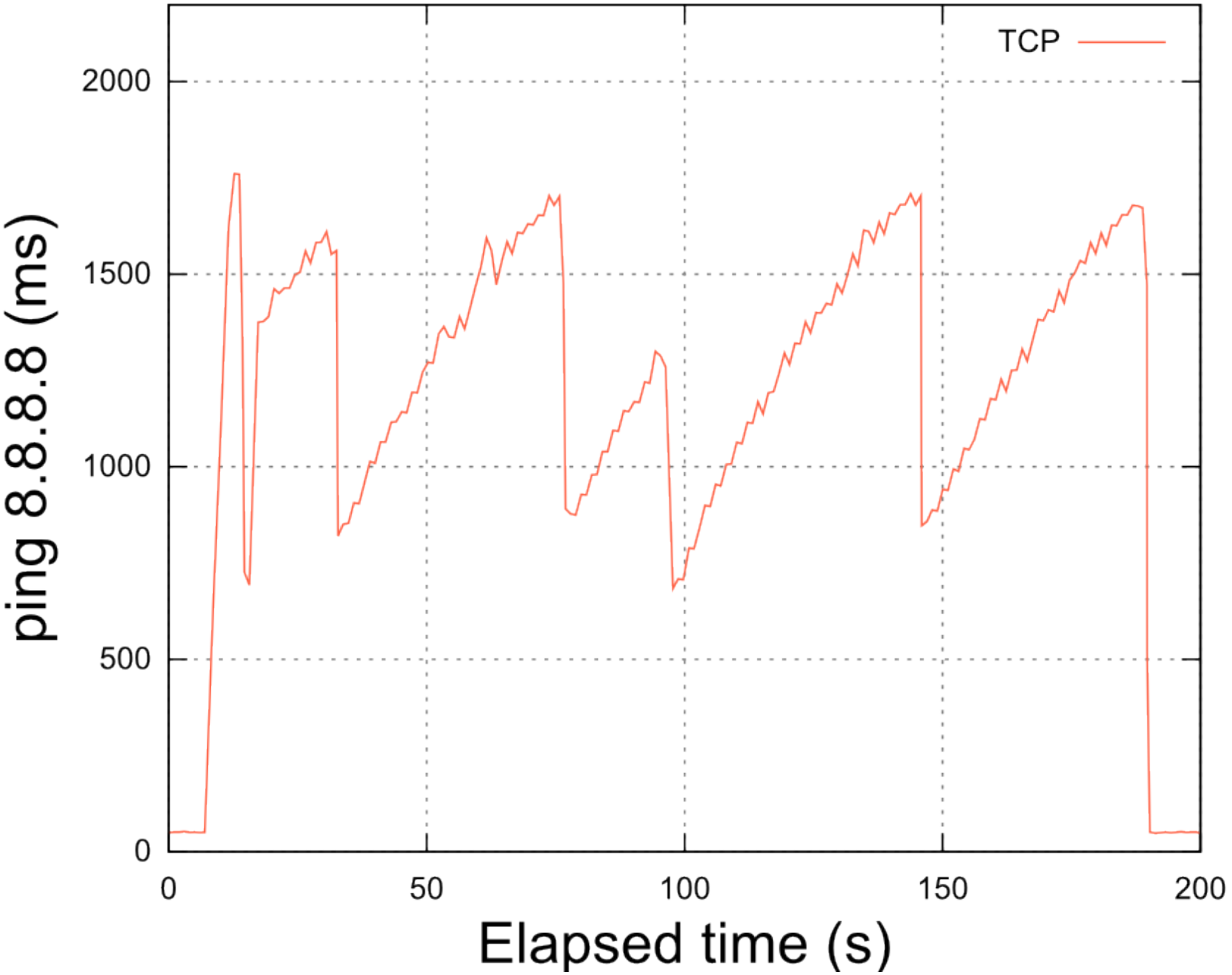
2. describe the LEDBAT algorithm;

3. describe the uTP as a particular implementation of LEDBAT.

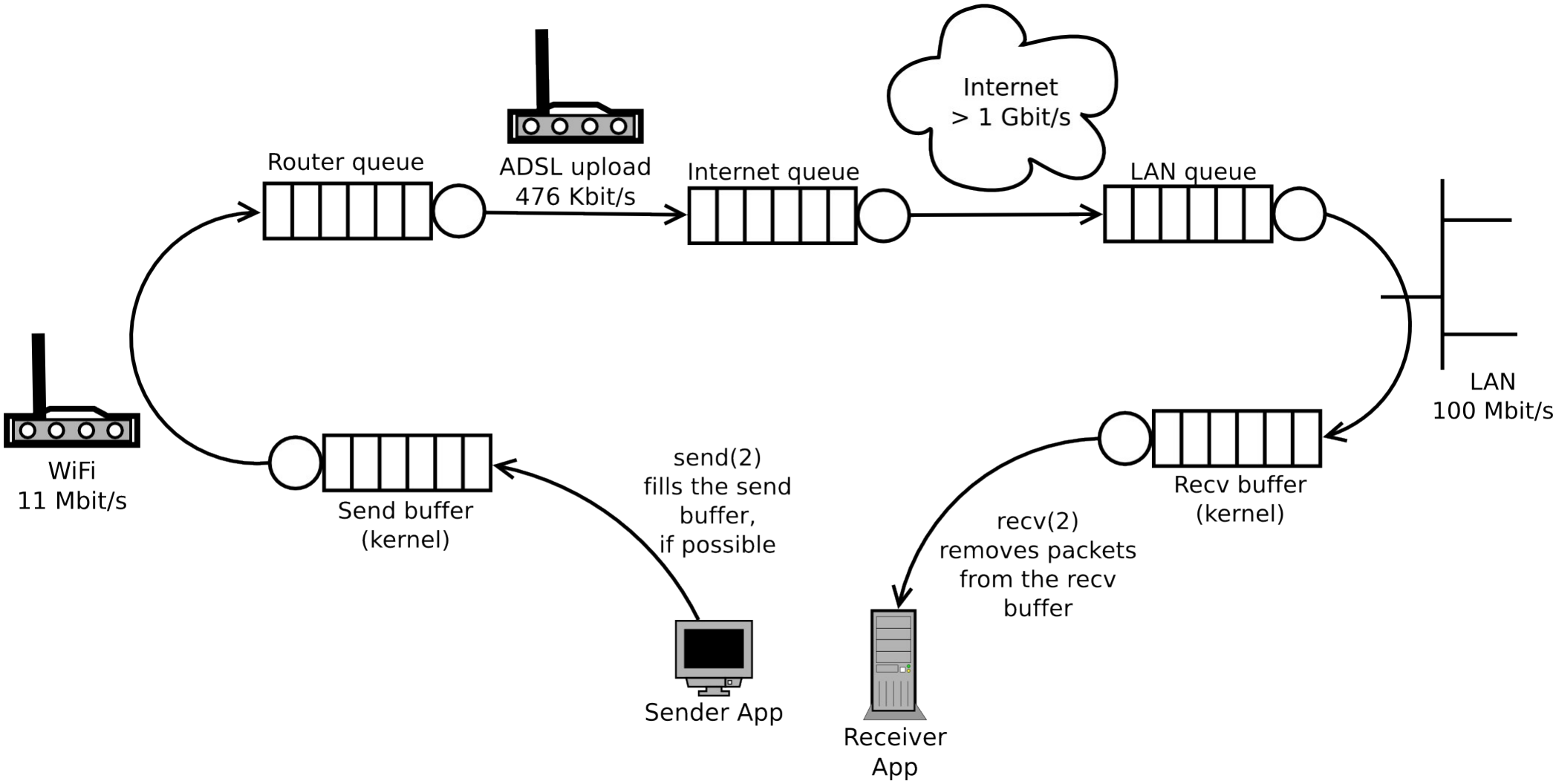
<BufferBloat>

D.I.Y. Upload Experiment w/ TCP




TCP
2000
1500
1000
500
0
ping 8.8.8.8 (ms)
0
50
100
150
200
Elapsed time (s)

The router queue is the cause



Milestone:

- we have experimentally measured the bufferbloat of my home router, and we have seen that it is quite large (never less than 0.5 s of extra RTT delay);

- class discussion: let's reason on why such extra delay significantly reduces the QoE (Quality of Experience) of DNS, web browsing, video streaming, VoIP.

Design problems:

- we have a design problem in the router: the buffer is too large with respect to the uplink speed;

- we have a design problem in the application: BitTorrent traffic should behave as background traffic.

Innovation without permission:

- Active Queue Management (AQM) has been possible for a couple of decades now;

- however the manufacturer of the router was not in a good position to optimize the user's traffic (using the AQM);

- instead, BitTorrent, which is closer to the edge (which is the user) had better understanding and much more incentives.

Take aways:

- bufferbloat: large router buffers, slow uplinks, and TCP, cause long queues and high extra delays;

- these extra delays reduce the QoE of interactive traffic: DNS, web browsing, video streaming, VoIP;

- BitTorrent designed LEDBAT to replace TCP, to control the extra delays, and to be gentler than TCP in dealing with interactive traffic;

- The router manufacturers could have solved the problem, but BitTorrent did, because it was at the edges of the network and had more incentives.

</BufferBloat>

<LEDBAT>

Like TCP, LEDBAT (RFC6817) is a window-based congestion control algorithm:

- LEDBAT uses a Congestion Window (CWND) to control the amount of traffic injected into the network; also, it uses sequence numbers and ACK numbers.

- But, TCP seeks to keep the optimal number of packets in flight, while LEDBAT aims to control the bufferbloat in the router.

The design goals of LEDBAT are spelled out by the acronym:

1) Low Extra Delay: control the extra delay and avoid the bufferbloat;

2) BAckground: when there is contention for the bottleneck, yield to TCP, otherwise use the bandwidth to the extent permitted by 1);

3) Transport: be reliable, just like TCP.

<DelayBasedCongestionControl>

Delay-Based Congestion Control:

Each ACK carries a measurement of the current one-way delay of the path. Also, LEDBAT tracks the minimum current delay (the base delay).

The difference between the current and the base delays is the extra delay, which is an estimate of the self-inflicted bufferbloat in the router.

LEDBAT target delay is to have 100 millisecond of extra delay (this is an engineering choice).

When the extra delay is smaller than 100 ms, LEDBAT inflates the CWND; when the extra delay is higher than 100 ms, LEDBAT deflates the CWND.

/* We have four diverse delays, so here's a recap, just in case you are lost. */

current delay = the delay measurement included into the current ACK

base delay = min(base delay, current delay)

extra delay = current delay - base delay

target delay = 100 ms ("magic number")

Q: Why does LEDBAT use the one-way delay instead of the RTT?

Well, the objective is to estimate the bufferbloat; therefore, it is more precise to consider the one-way delay only. An increase in the return path delay, in fact, should not impact onto the bufferbloat estimate.

Q: Why is the target 100 ms?

The 100 ms "magic number" is the result of a compromise between the performance of the LEDBAT on the one hand and the QoE of other protocols on the other hand.

100 ms, in fact, is 50 ms below the ITU recommendation concerning the maximum one-way delay acceptable for voice.

Therefore, by setting the max. acceptable extra delay (i.e., the target) to 100 ms, one does not make VoIP impossible.

Very simplified sender algorithm:

When LEDBAT receives an in-sequence ACK for a single max-size packet, it updates its CWND using the following formulae:

$$off\_target = 1 - extra\_delay / target$$

$$CWND \mathrel{+}= off\_target / CWND$$

Note: the CWND unit is a max-size packet (the typical max-size for Ethernet is 1,500 bytes minus the higher layer headers)

Example #1: no extra delay

```
CWND = 24     /* unit: max. size packets */
base_delay = 67    /* millisecond (one way!) */

<<arrives ACK>>

cur_delay = 67    /* transported by the ACK */

off_target = 1 - (67 - 67) / 100 = 1

CWND += 1 * 1/24     /* linear, like TCP */
```

Note: extra_delay *cannot* become negative, because if the current delay is smaller than the base delay, the base delay is updated and becomes equal to the current delay.

As said, the base delay is, in fact, the min. of the current delays measured over a reasonable time window.

Example #2: moderate, below-target extra delay

```
CWND = 24     /* unit: max. size packets */
base_delay = 67    /* millisecond (one way!) */

<<arrives ACK>>

cur_delay = 133    /* transported by the ACK */

off_target = 1 - (133 - 67) / 100 = 0.34

CWND += 0.34 * 1/24    /* slower than TCP */
```

Example #3: above-target extra delay

```
CWND = 24     /* unit: max. size packets */
base_delay = 67    /* millisecond (one way!) */

<<arrives ACK>>

cur_delay = 512    /* transported by the ACK */

off_target = 1 - (512 - 67) / 100 = -3.45

CWND += -3.45 * 1/24     /* shrink!!! */
```

Note: here, instead, TCP would grow the CWND.

/* Perhaps obvious, but:

If the CWND was 24 (max. size packets), and we deflate it to 23.85, it means that:

a) before, OK to have 24 packets 'in flight';

b) now, we MUST reduce the number of packets 'in flight' to 23.

That is, we are slowing down (which means less queue in the home router). */

QoE discussion:

LEDBAT is never more aggressive than TCP is, plus its extra delay is around 100 ms (i.e., about 1/10 of the TCP extra delay we observed).

So, a video stream should have a better QoE when competing with LEDBAT than when competing with TCP. Also, DNS, Skype, and web browsing should have a better Quality of Experience than they had with TCP.

But, the 100 ms target is a compromise and may not fit well any possible scenario.

</DelayBasedCongestionControl>

<LossAndRecovery>

When there are packet losses, LEDBAT behaves, basically, like TCP, i.e.:

1. Fast Retransmit and Fast Recovery on the receipt of the triple-duplicate ACK;

2. Also, Fast Retransmit and Fast Recovery when Selective ACKs indicate that, after a lost packet, three or more packets 'left the network' (more on this soon);

3. Timeout as extrema ratio.

Aside: Selective ACKs (SACKs)

SACKs allow the sender to recovery from a loss very quickly;

Basically, the receiver includes into the duplicate ACK packet a vector that says which of the packets 'in flight' have been received and which have been lost.

By knowing exactly which packets were lost and which packets are in flight, the sender can respond to congestion more quickly and more precisely (e.g., no unneeded retransmissions)

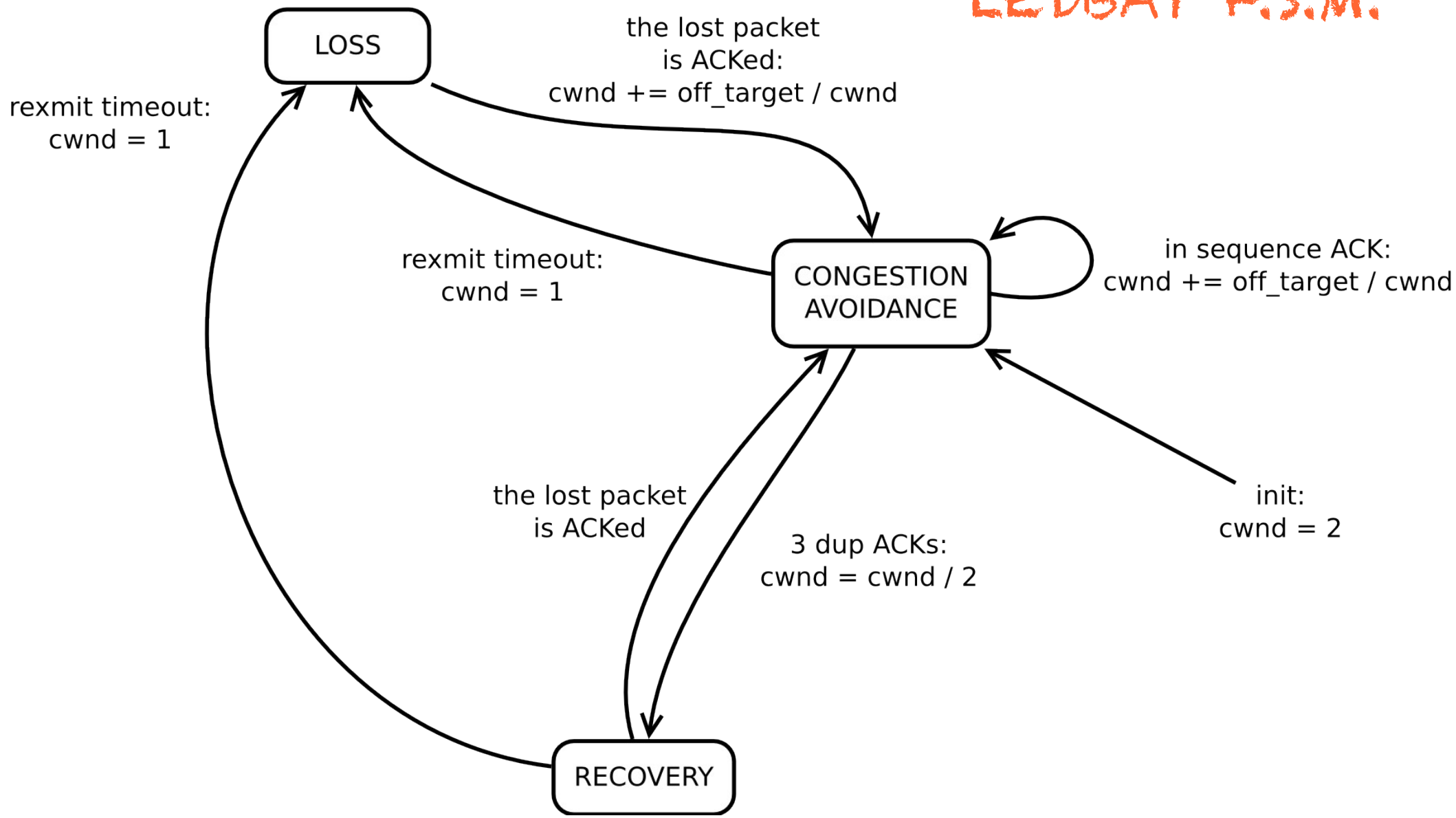
Aside: Retransmit Timeout (RTO)

Here's the formula that LEDBAT uses to compute the RTO from RTT measurements:

```
delta = rtt - packet_rtt
rtt_var += (abs(delta) - rtt_var) / 4;
rtt += (packet_rtt - rtt) / 8;
rto = max(rtt + rtt_var * 4, 500);
```

Basically, it's same-as the one of TCP, except that LEDBAT forces the RTT to be greater or equal than 500 ms.

LEDBAT F.S.M.


LOSS
the lost packet
is ACKed:
cwnd += off_target / cwnd
rexmit timeout:
cwnd = 1
rexmit timeout:
cwnd = 1
CONGESTION
AVOIDANCE
in sequence ACK:
cwnd += off_target / cwnd
the lost packet
is ACKed
3 dup ACKs:
cwnd = cwnd / 2
init:
cwnd = 2
RECOVERY

</LossAndRecovery>

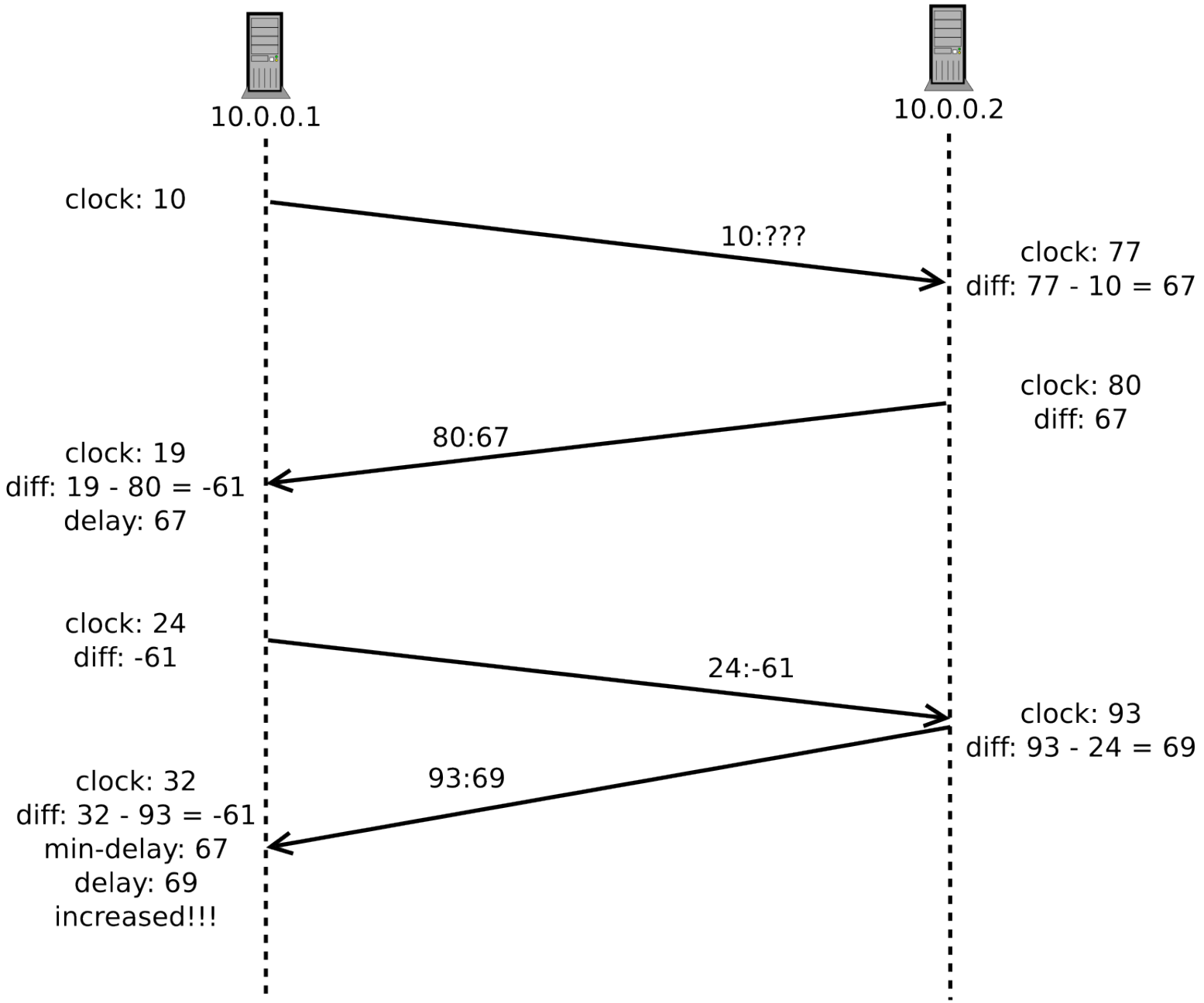
<OneWayDelayMeasurements>

Each LEDBAT packet must carry a timestamp from the sender.

When the receiver processes the packet, it computes the difference between its own timer and the sender's timestamp (called the timestamp difference).

Note: It's unlikely that the sender's and the receiver's clock are in sync; However, it is still useful to compare the difference to previous values.

Therefore, when the receiver has something to send back to the sender (e.g., an ACK), it will include the timestamp difference (which is, in fact, the cur_delay) into the packet.


10.0.0.1
10.0.0.2
clock: 10
10:???
clock: 77
diff: 77 - 10 = 67
clock: 80
diff: 67
80:67
clock: 19
diff: 19 - 80 = -61
delay: 67
clock: 24
diff: -61
24:-61
clock: 93
diff: 93 - 24 = 69
93:69
clock: 32
diff: 32 - 93 = -61
min-delay: 67
delay: 69
increased!!!

Complications:

a) the base one-way delay is not constant, therefore it is reasonable to update its value over time (e.g., use the minimum sample measured in the last two minutes);

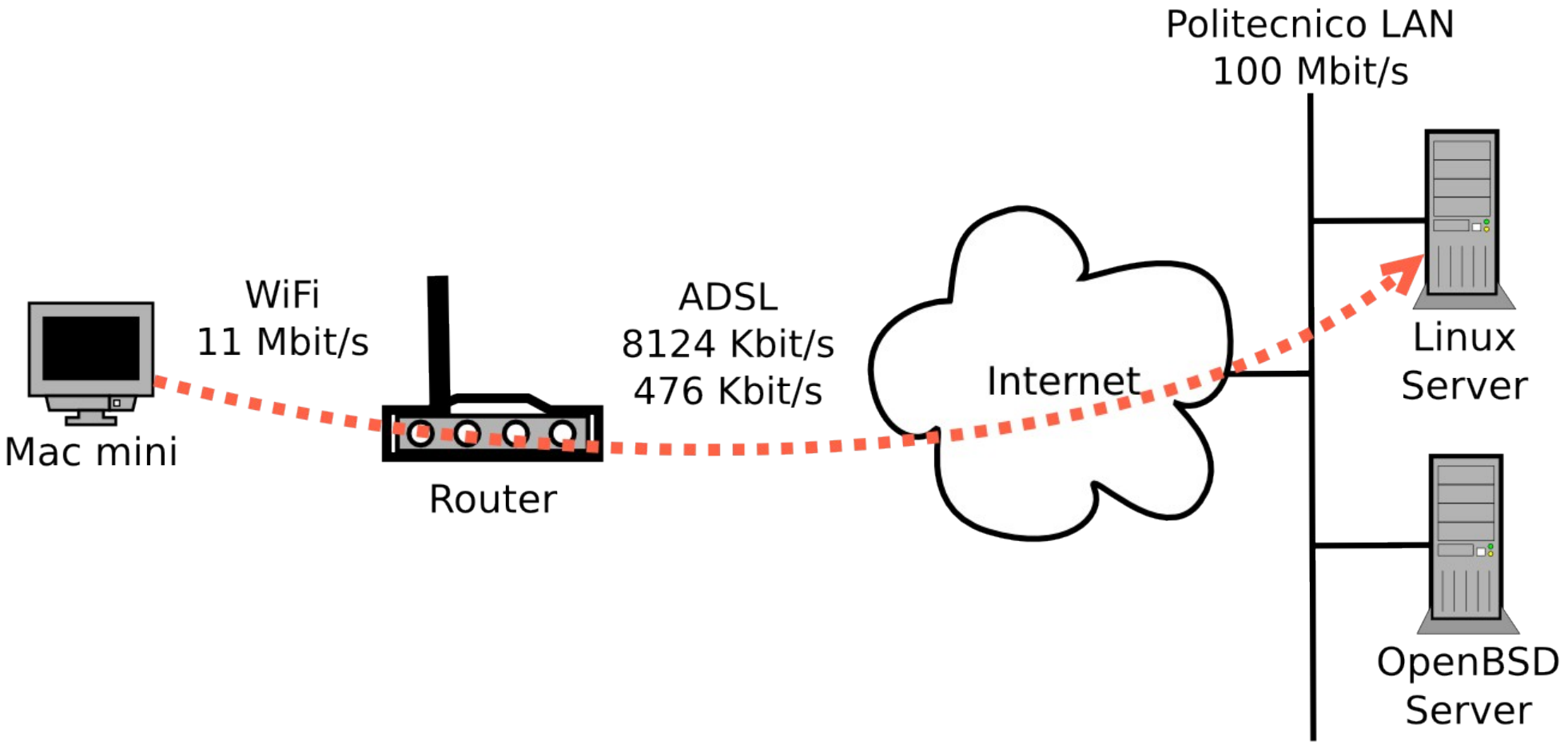
b) the timestamp difference is noisy, and there may be good reasons to filter it (e.g., using EWMA - Exponentially Weighted Moving Average);

c) the problem of the clock skew (i.e., that one peer's clock moves faster than the other's).

For more details on the above, see: RFC6817.

</OneWayDelayMeasurements>

D.I.Y. Upload Experiment w/ LEDBAT and uTP



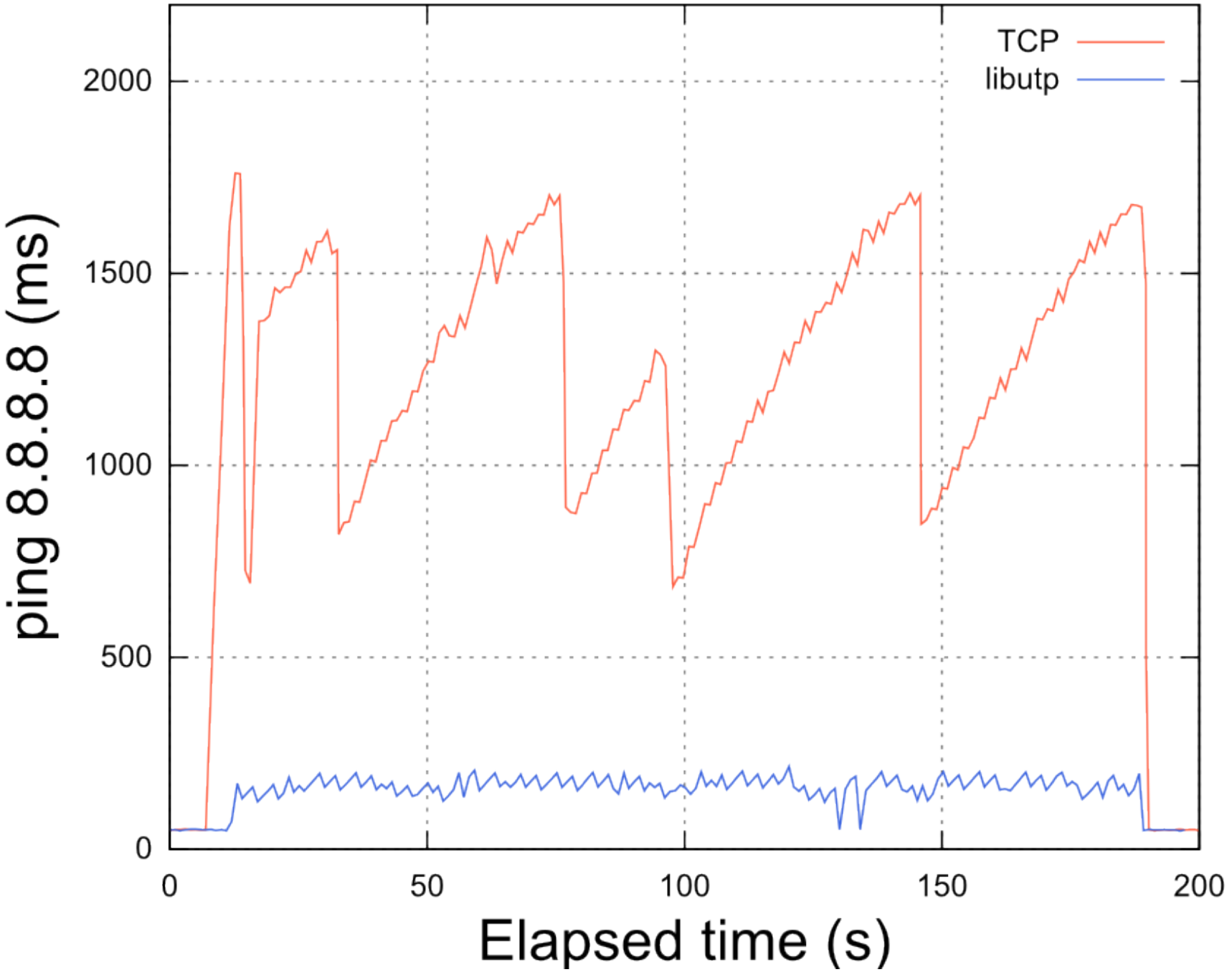
In this second experiment I use:

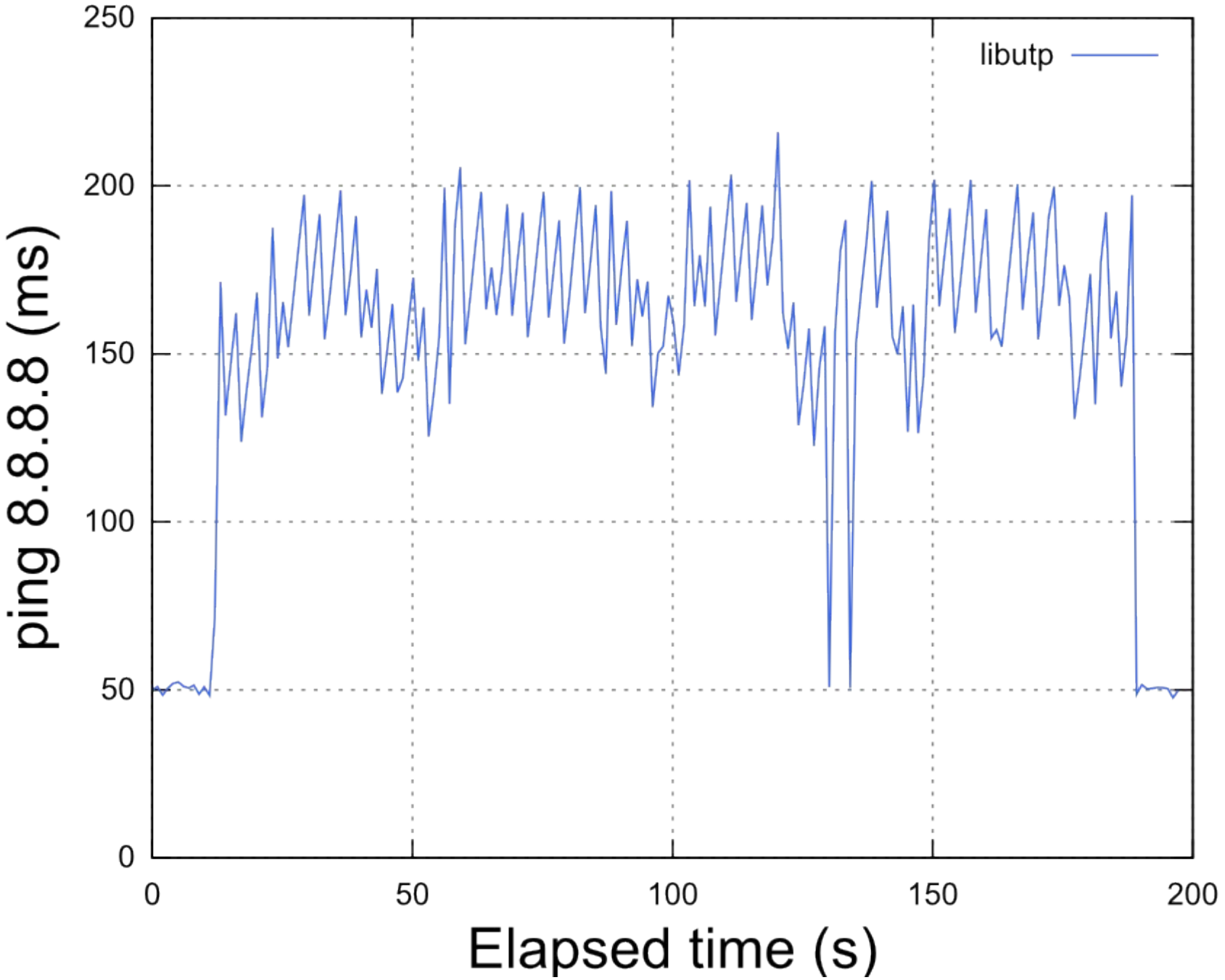
1) BitTorrent's libutp, which is an user-space C/C++ implementation of the LEDBAT algorithm on top of the UDP-based uTP protocol (we will describe the uTP packet format soon);

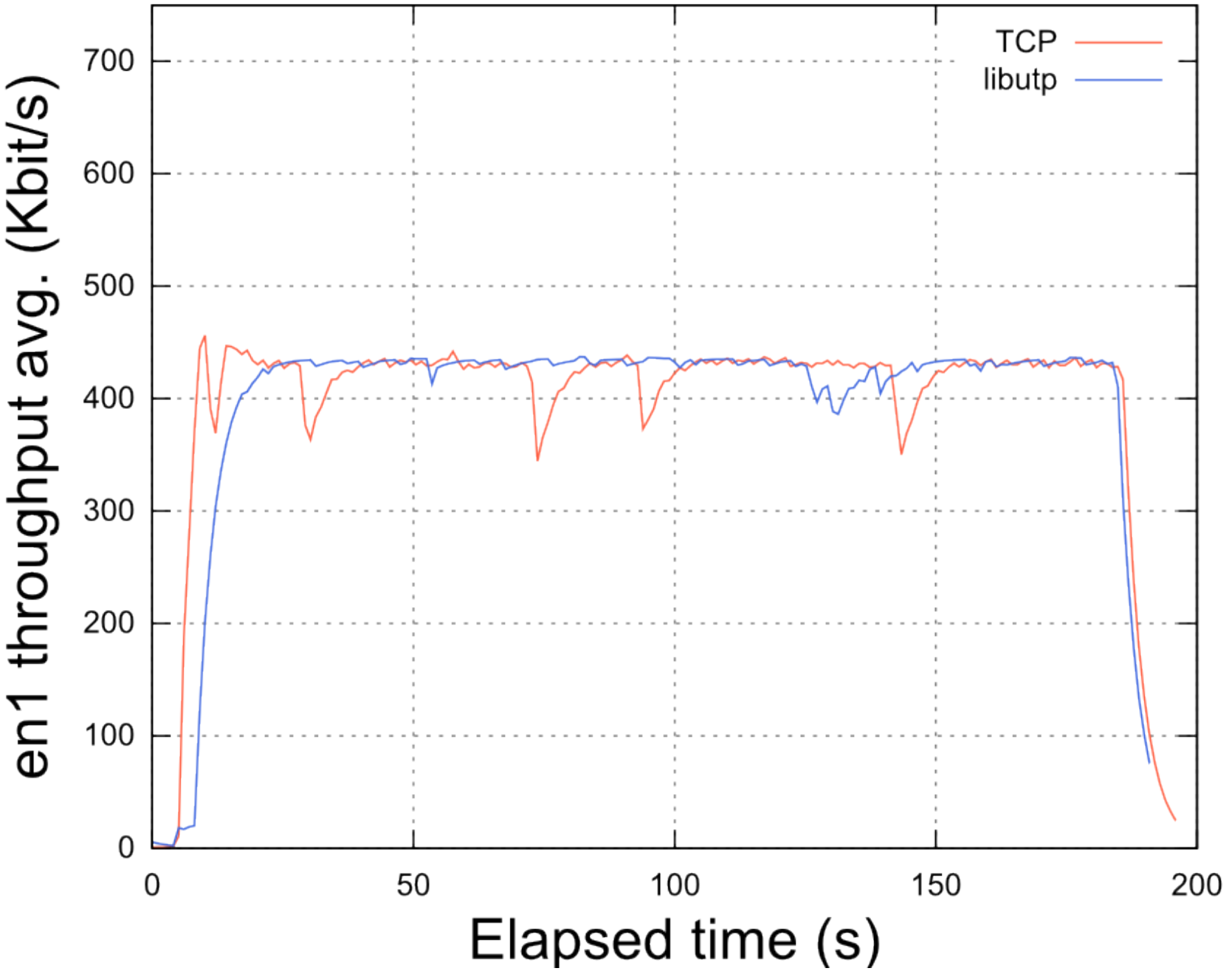
2) my own implementation, which emulates a subset of libutp (dubbed emul_utp).

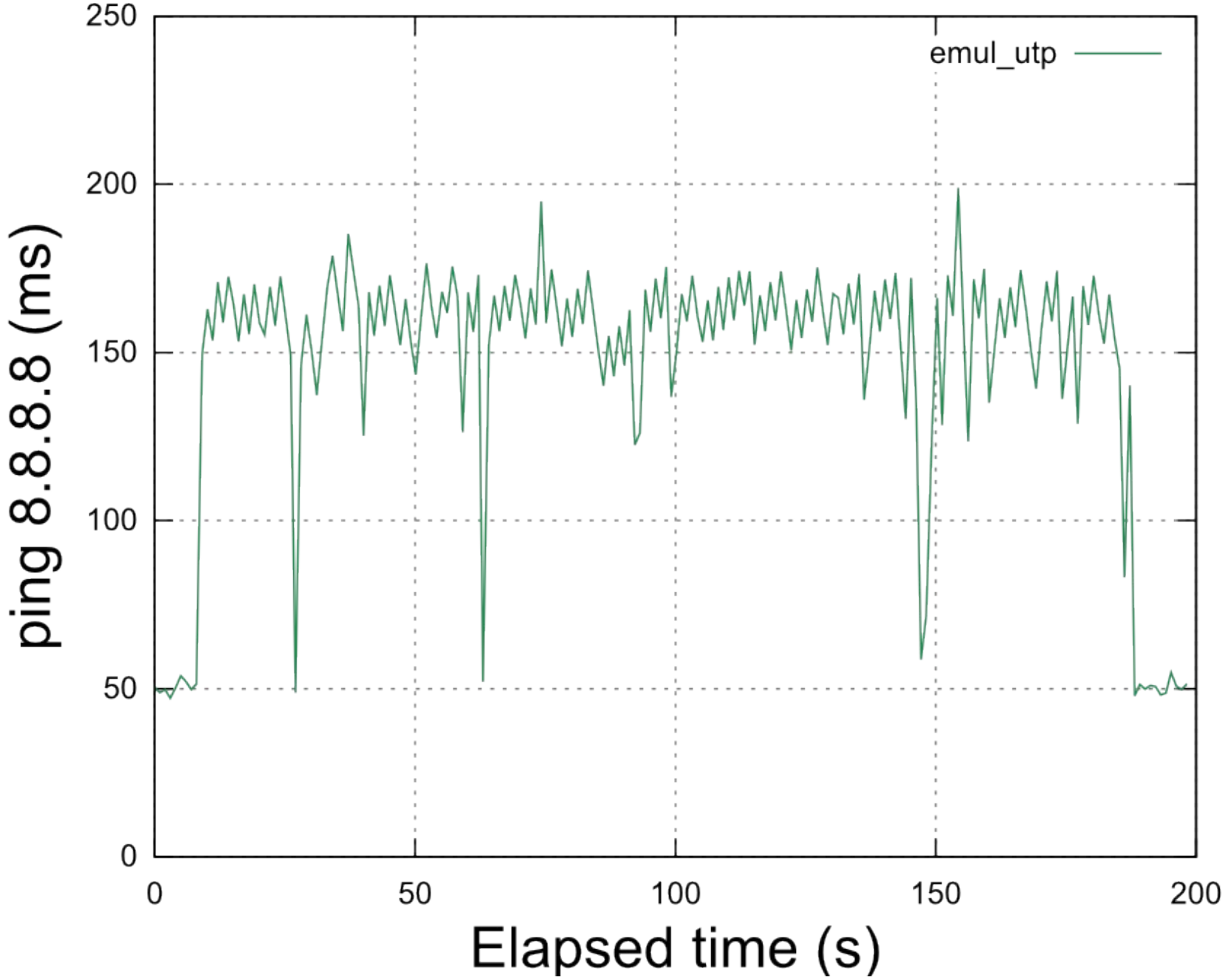
BTW, everything related with this lecture (including the source code and the slides) is available under permissive licenses at:

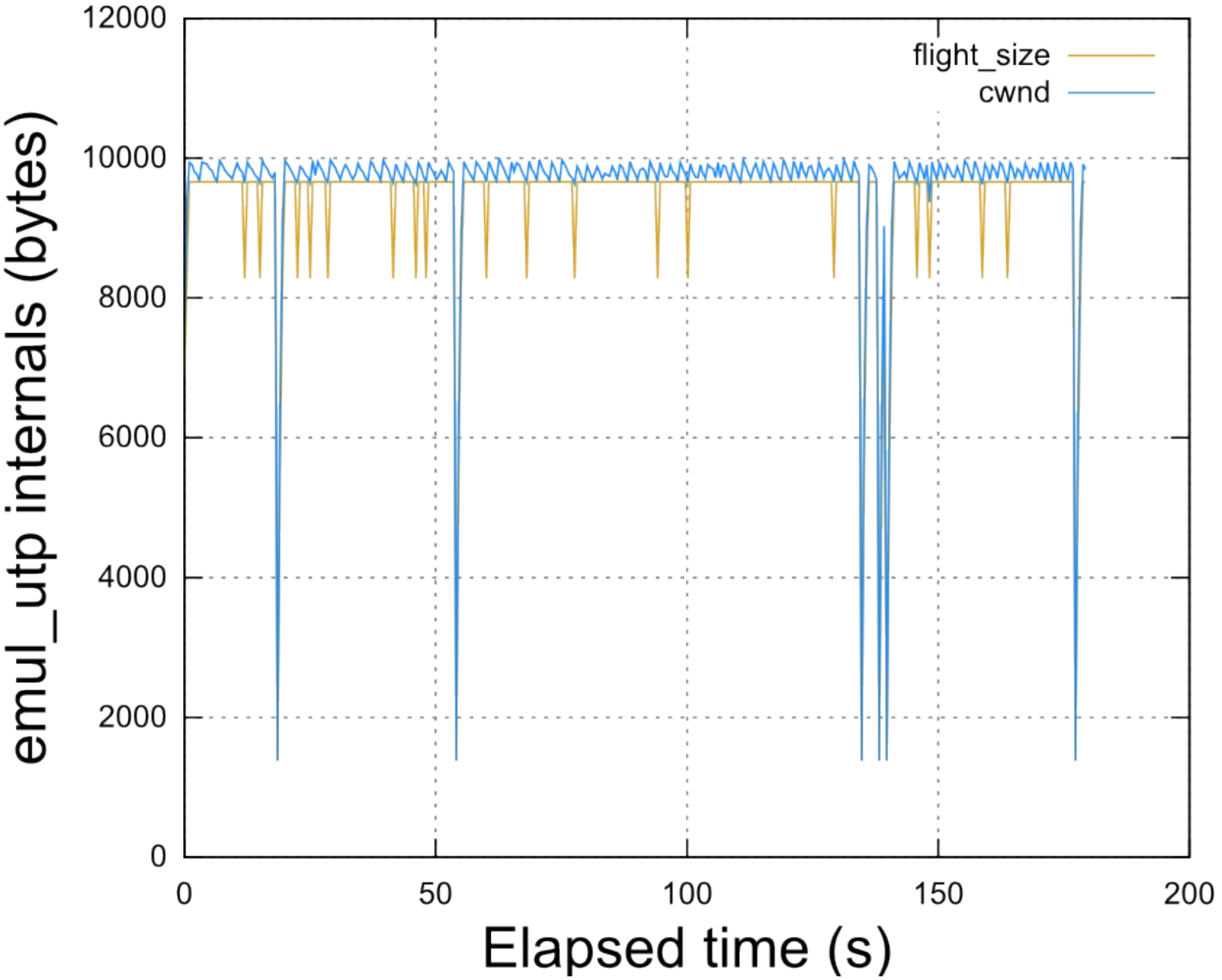
http://github.com/bassosimone/utpintro

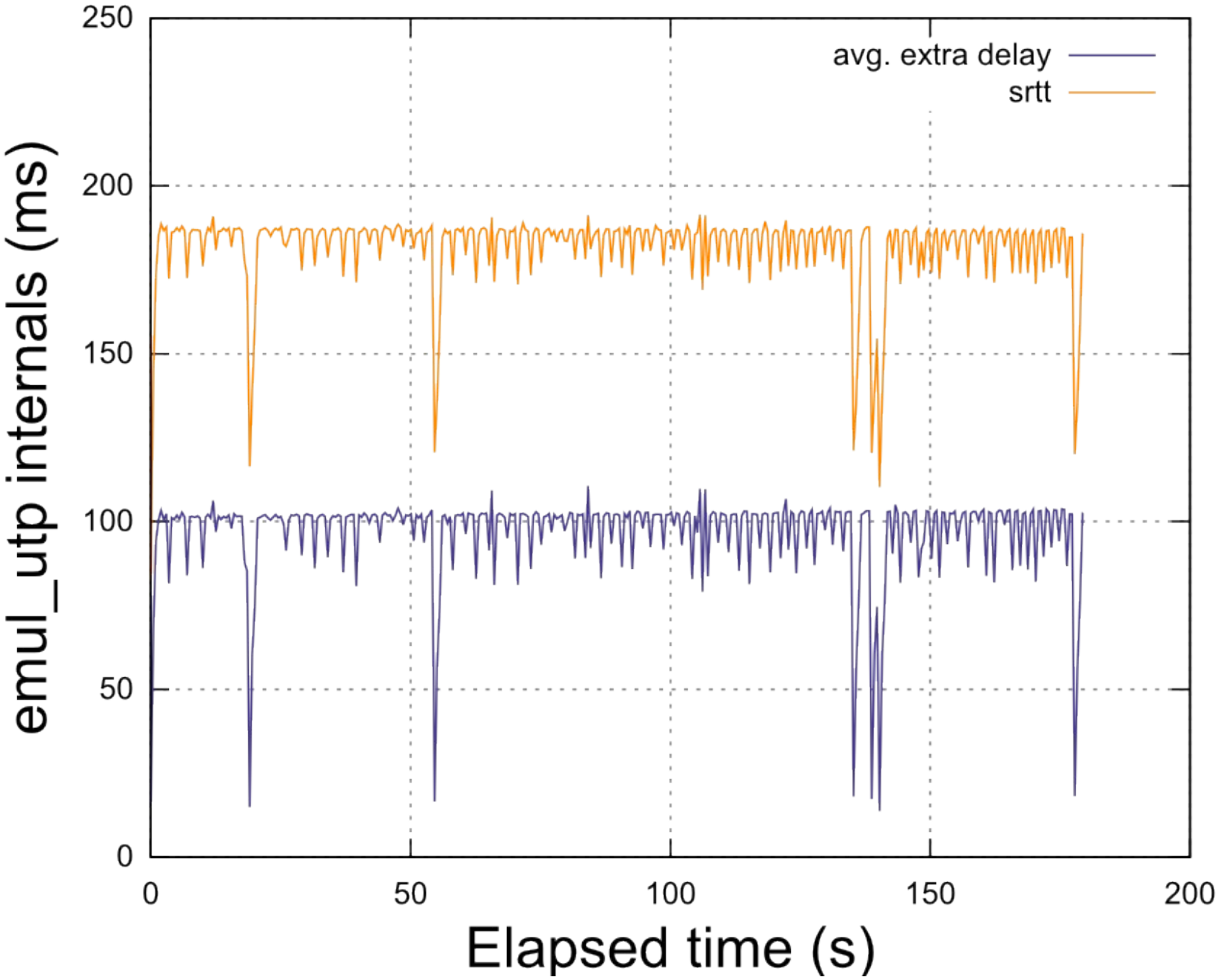

TCP
libutp
2000
1500
1000
500
0
ping 8.8.8.8 (ms)
0
50
100
150
200
Elapsed time (s)


libutp
ping 8.8.8.8 (ms)
Elapsed time (s)
250
200
150
100
50
0
0
50
100
150
200


TCP
libutp
en1 throughput avg. (Kbit/s)
0
100
200
300
400
500
600
700
0
50
100
150
200
Elapsed time (s)


emul_utp
ping 8.8.8.8 (ms)
Elapsed time (s)
0
50
100
150
200
250
0
50
100
150
200


12000
10000
8000
6000
4000
2000
0
0
50
100
150
200
emul_utp internals (bytes)
Elapsed time (s)
flight_size
cwnd


avg. extra delay
srtt
emul_utp internals (ms)
Elapsed time (s)
250
200
150
100
50
0
0
50
100
150
200

Take aways:

LEDBAT is like a TCP in which the Congestion Avoidance state is different, because the CWND increases or decreases proportionally to the extra one-way delay (and never increases more aggressively than what TCP does).

This algorithm allows the sender to control the extra one-way delay, which the sender strives to keep around 100 ms (engineering choice).

Since the bufferbloat is under control and since LEDBAT is typically nicer than TCP, many interactive applications have an improved QoE.

Research topics

1. the 100 ms threshold is a 'magic number';

2. the latecomer-unfairness problem (but a more-fair LEDBAT was proposed by Carofiglio, et al.);

3. LEDBAT does not interact well with Active Queue Management and becomes more aggressive;

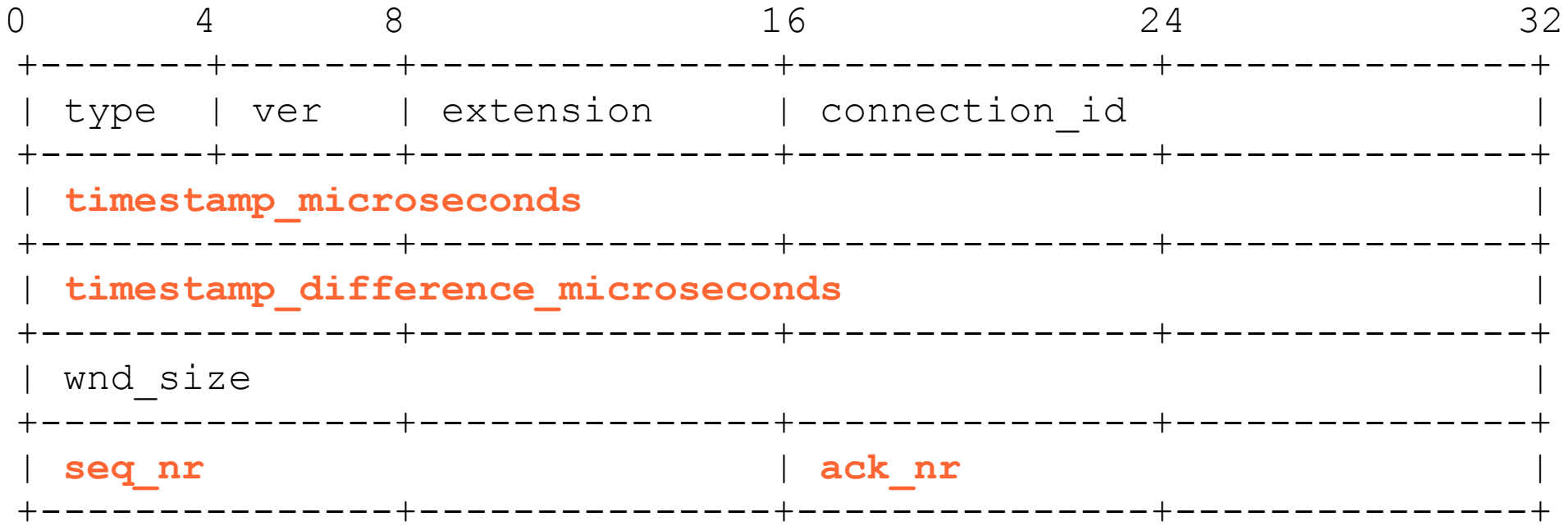
4. characterizing the bufferbloat with distributed passive and active measurement techniques.

(More details in the git repository.)

</LEDBAT>


<uTP>

The Packet Format (from BEP 29):



/* I have highlighted the fields that we have already implicitly discussed. */

Timestamps:

- we already discussed the timestamps when we presented the delay measurements;

- the timestamp is the system timestamp modulo 32 bit (i.e., 4294.967296 s):

```
struct timeval tv;
uint64_t sys_timestamp;
uint32_t timestamp;

(void)gettimeofday(&tv, NULL);
sys_timestamp = tv.tv_sec * 1000000 + tv.tv_usec;
timestamp = (uint32) sys_timestamp;
```

Sequence and ACK numbers:

- as we already said, like TCP, LEDBAT and uTP use sequence numbers and ACK numbers;

- differently from TCP, however, in uTP the sequence and ACK numbers indicate packets, not bytes.

The window size (wnd_size):

- like TCP, LEDBAT and uTP do flow control in addition to congestion control; and the window size is, in fact, the space (in bytes) available in the receiver's queue;

- when uTP wants to send a packet (because the CWND allows it to do so), it checks whether the receiver has enough space in queue;

- if there's not enough space in queue, the sender *does not* send the packet.

The extension field:

The extension field indicates the type of the first extension header, if any.

BEP 29 defines only one kind of extension, which is the SACK vector (type = 1).

If the extension field is zero, there are no extension headers.

/* Here I don't discuss the extension headers and the SACK vector; if you are interested, please refer to BEP 29. */

The connection_id:

The connection_id is, basically, an application-level port number.

That is, BitTorrent bind()s one single UDP socket (e.g., *:6881/udp), and routes the incoming traffic to the right UTPSocket object (which is 'connected' to a peer), by using the connection_id value.

The version field:

This field is always 1, because we are discussing, in fact, the version 1 header :-).

The type of the packet indicates the packet semantic:

ST_DATA: a packet that contains a payload and an ACK number;

ST_FIN: like the TCP FIN;

ST_STATE: a pure ACK, i.e., there is no payload;

ST_RESET: like the TCP RST;

ST_SYN: like the TCP SYN.

Connection Setup:

- Like in TCP, the connector sends to the acceptor a SYN packet, and

- (differently from TCP) the acceptor responds by sending back a STATE packet (TCP, instead, responds with a SYN|ACK segment);

- also, the connection is set up just after these two packets (TCP does a 3-way handhsake);

Connection id agreement:

Connector	Acceptor
recv_id = 7; /* random! */	
send_id = 8;	
-- ST_SYN, 7 -->	
	send_id = 7;
	recv_id = 8;
<-- ST_STATE, 7 --	
...	
-- ST_DATA, 8 -->	
<-- ST_STATE, 7 --	
...	

Key idea: in the SYN, the connector sends to the acceptor its own receiver id; after that, the two peers always send packets including the peer's receiver id (a.k.a. their sender id).

Packet sizes:

- when a link is congested, uTP (which typically uses 1300+ bytes packets), reduces the size of the packets to avoid the serialization delay on the router;

- the minimum packet size is 150 bytes (in which, of course, the headers overhead is more significant).

</uTP>

References

BEP 29: http://www.bittorrent.org/beps/bep_0029.html

RFC 6817: http://tools.ietf.org/html/rfc6817

This lecture's git repository:
https://github.com/bassosimone/utpintro

Me: http://nexa.polito.it/people/sbasso
https://twitter.com/bassosimone