TCP Performance

COMPUTER NETWORKS A.A. 24/25

Leonardo Maccari, DAIS: Ca’ Foscari University of Venice,
leonardo.maccari@unive.it

Venice, fall 2024

License

TCP Perfomance & Coding

- So far we focused mostly on the mechanisms that allow TCP to work properly
- This lesson briefly introduces the features of TCP that were introduced to improve its performance
- Later on we will analyse C code to create sockets, the POSIX API towards transport layer

Sect. 1 TCP Performance

Window Mechanism

- We know that in go-back-n protocol, there are two window sizes, the sender window size ($swnd$) and the receiver window size ($rwnd$)
- We know that the sender will send data up to the minimum between the two windows $window = min(swnd, cwnd)$, then it will wait for `ACK`s.
- $swnd$ is reduced every time a segment is sent and an `ACK` was not yet received,
- this is because the sender needs to maintain in the sending buffer the data that were not acknowledged.
- $rwnd$ is reduced every time data is correctly received by TCP but not fetched by the receiver application.

Window Mechanism

- So as we already mentioned, the window system is a trade-off between the need to have large buffers and not overload the receiver
- As a consequence, the maximum throughput is given by $\frac{window}{RTT}$
- To explain this number imagine what happens for instance, in a big file-transfer.
- In this case the sender always has something to send, and the receiver does not have to elaborate data, just save them on disk.

Sending Window

- Due to Nagle's algorithm the sender will send as many segments of size MSS as $swnd$ allows (practically all together), this happens at time t_0
- Let's assume that the link capacity of both ends is very high, and that segments are all received
- Segments are received at time $t_0 + RTT/2$
- Then the receiver will send a cumulative ACK
- At time $t_0 + RTT$ the sender receives the ACK, the sending buffer is cleared and then the sender will start again.

Sending Window

- So for every RTT, at most $window$ data is sent and the maximum throughput is $\frac{window}{RTT}$
- This throughput is reached from the very beginning of the TCP connection

Receiving Window

- If the receiver instead is not fast enough, it will send an `ACK`, but in the `ACK` message, there will be a new (smaller) size for the receiver window, we call it $rwnd' < rwnd$
- In that case when the `ACK` is received, at time $t_0 + RTT$ the sender can only send as much data as $window' = min(rwnd', swnd)$
- The throughput is $\frac{window'}{RTT}$, so in general, the formula is still valid, but the window is reduced
- However, what happens if there is network congestion?

Network Congestion

- This is a situation in which the routers on the path from the client to the server drop packets
- This is due to the fact they need to route too many packets and their queues get filled up, so they start dropping packets
- However, it has nothing to deal with the receiver congestion: there could be network congestion but the application at the receiver is able to read all the data as soon as they come.

Network Congestion

- Let's assume there is no problem at the receiver application, but we have network congestion
- This is detected by the sender as a loss of packets
- For instance, at time $t_0 + RTT$ the cumulative `ACK` message does not allow to empty the whole sending buffer, but only, for example, half of it.

Network Congestion

- However, the receiver window will not be decreased
- So if the sender always has data to send (as in the file-transfer) the effect is:
  - the sending buffer is filled with new data
  - the sender may just send the whole buffer again, including old data to be re-sent and new data[1]

[1] We know that the fast retransmit strategy waits for three ACKs before retransmitting, but let's forget about this for this example.

Network Congestion

- As a consequence, the average amount of data sent is still $\frac{window}{RTT}$, even if the new data delivered is at most $\frac{window}{2 \times RTT}$ (because half of the buffer contains old data to be re-sent)
- That is, the sender will not slow down sending data, and the router will be still congested and will still drop packets
- If we want to solve the network congestion problem, the only possibility is to reduce $window$, so the sender sends less data, and the router becomes less congested, and stops to drop packets.
- So we need another way to detect and mitigate congestion, and reduce the sender window

Implicit Congestion Detection

- The problem of congestion starts even before . . . congestion itself.
- The sender, at the beginning of the connection should not just flood the receiver with a throughput that is too high, or this may actually create the congestion.
- The initial window must be small and grow with time.
- So congestion control is mostly about answering these two questions:
    - *What is the function that is used to increase the window from small initial value?*
    - *How is the window reduced when loss occurs?*

Congestion Window

- Let us introduce another window, the so-called congestion window ($cwnd$)
- $cwnd$ is initialized to some fixed value, initially it was $cwnd_0 = MSS$, now it is common to be $cwnd_0 = 10 \times MSS$.
- As MSS is normally 1460B (1500 Ethernet MTU - 40B of IP and TCP headers) $cwnd_0 = 14600B$

Congestion Window

- At every instant we have $window = min(cwnd, swnd, rwnd)$
- We also introduce another parameter: the slow-start threshold $sstrash$, that is also initialized at some small value (such as MSS)

Slow Start

- The Slow Start algorithm was proposed by Van Jacobson, and was the first congestion control algorithm adopted. It is now in RFC 5681. After that, a lot more methods followed.
- At every RTT, if all segments are `ACKed`, $cwnd$ is doubled, so there is an exponential growth of $cwnd$.
- This is not so slow, but better than starting with the maximum window...
- However, after a congestion event, if $cwnd > sstrash$ then $cwnd$ is not doubled at every MSS, but is increased of only one MSS per RTT.

Congestion Events:

There are two events that are interpreted as a congestion event

mild congestion : three `ACK`s are received with the same sequence number, a retransmission is attempted and it is successful (the following `ACK` confirms the reception)

severe congestion : the retransmission timeout fires

Mild Congestion Event

- When a mild congestion event happens then:

$$cwnd' = cwnd$$

$$cwnd = sstrash$$

$$sstrash = \frac{cwnd'}{2}$$

- that is: $cwnd$ is reset to the value of $sstrash$ and $sstrash$ is reset to half the value of $cwnd$ before the congestion event

Sever Congestion Event

- When a severe congestion event happens:

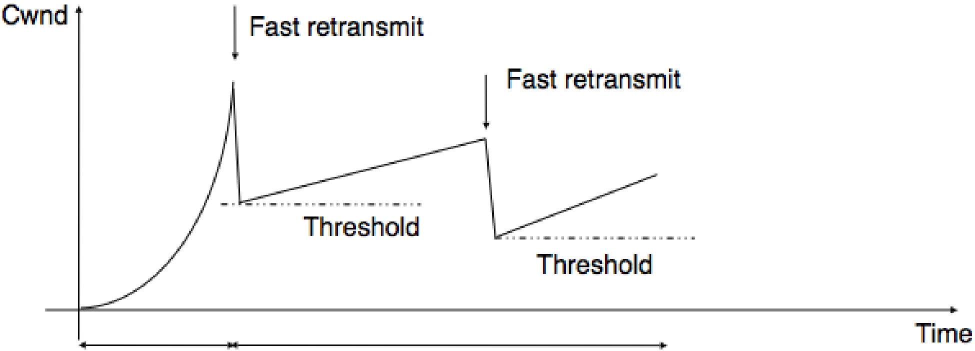
$$cwnd' = cwnd$$

$$cwnd = cwnd_0$$

$$sstrash = \frac{cwnd'}{2}$$

- that is: do slow-start from the beginning, and (as in mild congestion) reset $sstrash$ to half the value of $cwnd$ before the congestion event

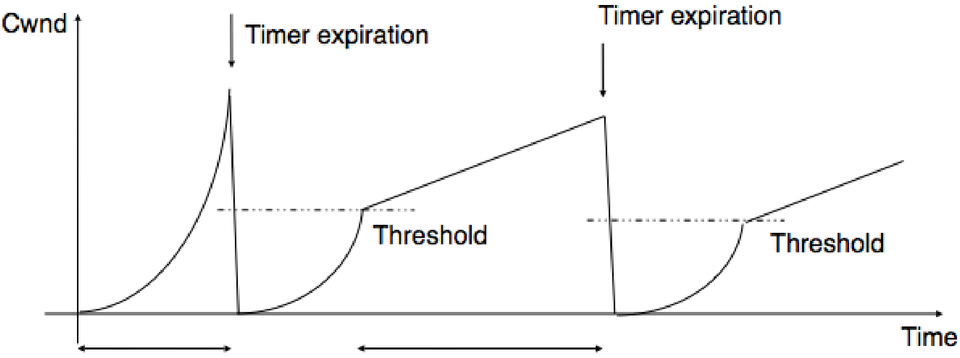
Mild Congestion



Slow-start
exponential increase of cwnd

Congestion avoidance
linear increase of cwnd

Severe Congestion



Slow-start
exponential increase of cwnd

Congestion avoidance
linear increase of cwnd

Explicit Congestion Control (ECN)

- While the general concept of the original algorithm is still valid (using the congestion window), the way congestion is detected can be also explicit
- This means the congested routers change some header bit to signal that packets were dropped
- We don't have time to go through this

Performance Estimation

- It can be shown that the maximum achievable throughput between two hosts can be estimated as:

$$throughput = min\left(\frac{MSS}{RTT}, \sqrt{\frac{3}{2}}\frac{MSS}{RTT\sqrt{p}}\right)$$

- Where p is the probability of a mild congestion event, assuming that severe congestion is very unlikely.
- Since MSS is generally set to 1460 the throughput is capped by the RTT and p

Congestion Control Zoo

- During the years, many variants of the original congestion control algorithm were proposed, including TCP Reno (the original one by VJ), TCP NewReno, TCP Vegas, and literally tens more.
- Among them we mention CUBIC, tha does not increase linearly, but with a cubic function and it is the default in Linux and BBR by Google (and Van Jacobson) that is now very popular.

Concluding Remarks

- If you want to find out the state of your TCP connections, on Linux you can do it with `ss`
  This will list all the connections, pick one and launch:
  `ss -i dst 1.2.3.4:3000`
- This will show all the state information of a connection to IP 1.2.3.4 and port 3000

Exercises

- You are now acknowledged enough to make the exercises TCP block from the Ingenious platform.

Sect. 2 Berkeley Sockets

Network Sockets

- A socket is a standard interface, defined by POSIX between applications and layer 4. The terms *network socket*, *Berkeley socket* or *BSD socket* are used interchangeably
- That is, when an application needs to send or receive data from the network, it uses an API that creates, manages and uses sockets.

UDP and TCP sockets in C Language

- Professor Campbell from Dartmouth has an excellent page describing how to program network sockets in C:
  `https://www.cs.dartmouth.edu/~campbell/cs60/socketprogramming.html`
- We will look at that instead than copying it here. . .

improving the code

- Andy Campbell is an excellent professor and a very nice guy. . .
- . . . however :-)
- Do the following:
  1. Run the server and then connect with the client
  2. Kill the server (CTRL+C) and re-launch it immediately
  3. Kill the client and re-launch it
- can the client re-connect? if not, try to find the problem, and fix the code.
- And then, find an overall motivation, solution, next week

Hints

- `netstat -tpna` tells the state of all ongoing connections in your POSIX machine
- essentially, it dumps the list of the TCB
- Run it in another terminal right after killing the server. Is the server connection dead?

Berkeley Sockets

↳ 2.1 TLS Socket

Code for a TLS/SSL socket

- In a similar way we can set-up a TCP socket that supports TLS
- We are going to look at the code from the OpenSSL examples:
  `https://wiki.openssl.org/index.php/Simple_TLS_Server`
- Some example functions that deserve a little explaining

  `TLS_server_method` This returns a method that can be used in a TLS connection. It is used when all TLS versions need to be supported, while instead one can restrict a specific version using `TLSv1_2_server_method`. If none is specified, all are allowed and the higher layer protocols will decide on the best one

  SSL_CTX_new creates a new `SSL_CTX` object, which holds various configuration and data relevant to SSL/TLS for session establishment

Code for a TLS/SSL socket

- In a similar way we can set-up a TCP socket that supports TLS
- We are going to look at the code from the OpenSSL examples: `https://wiki.openssl.org/index.php/Simple_TLS_Server`
- Some example functions that deserve a little explaining

  `SSL_CTX_use_certificate_file` Load a file containing a certificate

  `SSL_CTX_use_PrivateKey_file` Load a file containing a key