Communication Networks
Spring 2020

Laurent Vanbever
nsg.ee.ethz.ch

ETH Zürich (D-ITET)
May 4 2020

Materials inspired from Scott Shenker, Jennifer Rexford, and Ankit Singla
Last week on

Communication Networks
DNS Congestion Control

google.ch ←→ 172.217.16.131 (the beginning)
Congestion control aims at solving three problems:

1. **bandwidth estimation**
   - How to adjust the bandwidth of a single flow to the bottleneck bandwidth?
   - Could be 1 Mbps or 1 Gbps...

2. **bandwidth adaptation**
   - How to adjust the bandwidth of a single flow to variation of the bottleneck bandwidth?

3. **fairness**
   - How to share bandwidth “fairly” among flows, without overloading the network
Congestion control differs from flow control both are provided by TCP though

<table>
<thead>
<tr>
<th>Control Type</th>
<th>Prevents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow control</td>
<td>prevents one fast sender from overloading a slow receiver</td>
</tr>
<tr>
<td>Congestion control</td>
<td>prevents a set of senders from overloading the network</td>
</tr>
</tbody>
</table>
The sender adapts its sending rate based on these two windows

<table>
<thead>
<tr>
<th>Window</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receiving Window</td>
<td>How many bytes can be sent without overflowing the receiver buffer? based on the receiver input</td>
</tr>
<tr>
<td>Congestion Window</td>
<td>How many bytes can be sent without overflowing the routers? based on network conditions</td>
</tr>
<tr>
<td>Sender Window</td>
<td>minimum(CWND, RWND)</td>
</tr>
</tbody>
</table>
The 2 key mechanisms of Congestion Control

detecting congestion

reacting to congestion
The 2 key mechanisms of Congestion Control

detecting congestion

reacting to congestion
Detecting losses can be done using ACKs or timeouts, the two signal differ in their degree of severity.

- Duplicated ACKs: **mild** congestion signal
  - Packets are still making it

- Timeout: **severe** congestion signal
  - Multiple consequent losses
The 2 key mechanisms of Congestion Control

detecting congestion

reacting to congestion
TCP approach is to **gently increase** when not congested and to **rapidly decrease** when congested

**question**

What **increase/decrease function** should we use?

*It depends on the problem we are solving...*
Congestion control aims at solving three problems

<table>
<thead>
<tr>
<th>#1</th>
<th>bandwidth estimation</th>
<th>How to adjust the bandwidth of a single flow to the bottleneck bandwidth?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>could be 1 Mbps or 1 Gbps...</td>
</tr>
<tr>
<td>#2</td>
<td>bandwidth adaptation</td>
<td>How to adjust the bandwidth of a single flow to variation of the bottleneck bandwidth?</td>
</tr>
<tr>
<td>#3</td>
<td>fairness</td>
<td>How to share bandwidth “fairly&quot; among flows, without overloading the network</td>
</tr>
</tbody>
</table>
How to adjust the bandwidth of a single flow to the bottleneck bandwidth?

could be 1 Mbps or 1 Gbps...
Initially, you want to quickly get a first-order estimate of the available bandwidth

**Intuition**
- Start slow but rapidly increase until a packet drop occurs

**Increase policy**
- \( cwnd = 1 \) initially
- \( cwnd + = 1 \) upon receipt of an ACK
How to adjust the bandwidth of a single flow to variation of the bottleneck bandwidth?
<table>
<thead>
<tr>
<th></th>
<th>Increase Behavior</th>
<th>Decrease Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIAD</td>
<td>gentle</td>
<td>gentle</td>
</tr>
<tr>
<td>AIMD</td>
<td>gentle</td>
<td>aggressive</td>
</tr>
<tr>
<td>MIAD</td>
<td>aggressive</td>
<td>gentle</td>
</tr>
<tr>
<td>MIMD</td>
<td>aggressive</td>
<td>aggressive</td>
</tr>
</tbody>
</table>
How to share bandwidth “fairly” among flows, without overloading the network
AIMD converge to fairness and efficiency, it then fluctuates around the optimum (in a stable way)
Congestion control makes TCP throughput look like a “sawtooth”
The DNS system is a distributed database which enables to resolve a name into an IP address.

www.ethz.ch: 129.132.19.216
To scale, DNS adopt three intertwined hierarchies:

- **naming structure**: addresses are hierarchical. For example, `www.ee.ethz.ch`.
- **management**: hierarchy of authority over names.
- **infrastructure**: hierarchy of DNS servers.
naming structure addresses are hierarchical

www.ee.ethz.ch
A name, e.g. ee.ethz.ch, represents a leaf-to-root path in the hierarchy.
management  hierarchy of authority  over names
The DNS system is hierarchically administered.
infrastructure  hierarchy of DNS servers
13 root servers (managed professionally) serve as root (*)

(*) see http://www.root-servers.org/
The bottom (and bulk) of the hierarchy is managed by Internet Service Provider or locally.

Diagram:
- root
- com
- org
- net
- edu
- gov
- mil
- be
- ch
- de
- fr
- epfl
- ethz
- nzz
- www
- ee
- infk
Every server knows the address of the root servers (*) required for bootstrapping the systems

(*) see https://www.internic.net/domain/named.root
Each server knows the address of all children
This week on
Communication Networks
DNS

google.ch  172.217.16.131

Web

http://www.google.ch

(the end)
google.ch  <->  172.217.16.131
starting from slide 47/90
DNS

Web

http://www.google.ch
The Web as we know it was founded in ~1990, by Tim Berners-Lee, physicist at CERN

His goal: provide distributed access to data

The World Wide Web (WWW): a distributed database of “pages” linked together via the Hypertext Transport Protocol (HTTP)
The Web was and still is so successful as it enables everyone to self-publish content

Self-publishing on the Web is easy, independent & free and accessible, to everyone

People weren’t looking for technical perfection little interest in collaborative or idealistic endeavor

People essentially want to make their mark and find something neat...
The WWW is made of three key components

- **Infrastructure**
- **Content**
- **Implementation**

**Clients/Browser**
- Servers
- Proxies

**Objects**
- files, pictures, videos, ...

**organized in**
- Web sites
  - a collection of objects

**URL**: name content
**HTTP**: transport content
We’ll focus on its implementation

Infrastructure

Clients/Browser
Servers
Proxies

Content

Objects
files, pictures, videos, …

organized in
Web sites
a collection of objects

Implementation

URL: name content
HTTP: transport content
Infrastructure

Clients/Browser
Servers
Proxies

Content

Objects
files, pictures, videos, ...

organized in
Web sites
a collection of objects

Implementation

URL: name content
HTTP: transport content
A Uniform Resource Locator (URL) refers to an Internet resource

protocol://hostname[:port]/directory_path/resource
protocol://hostname[:port]/directory_path/resource

HTTP(S)
FTP
SMTP…
protocol:////hostname[:port]/directory_path/resource
default to protocol’s standard
HTTP:80, HTTPs:443

protocol://hostname[:port]/directory_path/resource
protocol://hostname[:port]/directory_path/resource

identify the resource on the destination
Clients/Browser
Servers
Proxies

Infrastructure

Objects
files, pictures, videos, …
organized in
Web sites
a collection of objects

Content

Implementation

URL: name content
HTTP: transport content
HTTP is a rather simple synchronous request/reply protocol

HTTP is layered over a bidirectional byte stream typically TCP, but QUIC is ramping up

HTTP is text-based (ASCII) human readable, easy to reason about

HTTP is stateless it maintains *no info* about past client requests
Performance

Protocol

http://

SPEED LIMIT
80
Protocol

http://

Performance

SPEED LIMIT 80
HTTP clients make request to the server

<table>
<thead>
<tr>
<th>HTTP request</th>
</tr>
</thead>
<tbody>
<tr>
<td>method &lt;sp&gt; URL &lt;sp&gt; version &lt;cr&gt;&lt;lf&gt;</td>
</tr>
<tr>
<td>header field name: value &lt;cr&gt;&lt;lf&gt;</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>header field name: value &lt;cr&gt;&lt;lf&gt;</td>
</tr>
<tr>
<td>&lt;cr&gt;&lt;lf&gt;</td>
</tr>
<tr>
<td>body</td>
</tr>
<tr>
<td>method</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>header field name: value</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>header field name: value</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

**body**
<table>
<thead>
<tr>
<th>method</th>
<th>GET</th>
<th>return resource</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>HEAD</td>
<td>return headers only</td>
</tr>
<tr>
<td></td>
<td>POST</td>
<td>send data to server (forms)</td>
</tr>
<tr>
<td>URL</td>
<td>relative to server (e.g., /index.html)</td>
<td></td>
</tr>
<tr>
<td>version</td>
<td>1.0, 1.1, 2.0</td>
<td></td>
</tr>
</tbody>
</table>
HTTP clients make request to the server

<table>
<thead>
<tr>
<th>HTTP request</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>method &lt;sp&gt; URL &lt;sp&gt; version</td>
<td>&lt;cr&gt;&lt;lf&gt;</td>
</tr>
<tr>
<td>header field name: value</td>
<td>&lt;cr&gt;&lt;lf&gt;</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>header field name: value</td>
<td>&lt;cr&gt;&lt;lf&gt;</td>
</tr>
<tr>
<td>&lt;cr&gt;&lt;lf&gt;</td>
<td></td>
</tr>
</tbody>
</table>

body
Request headers are of variable lengths, but still, human readable

<table>
<thead>
<tr>
<th>Uses</th>
<th>Authorization info</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acceptable document types/encoding</td>
<td></td>
</tr>
<tr>
<td>From</td>
<td>(user email)</td>
</tr>
<tr>
<td>Host</td>
<td>(identify the server to which the request is sent)</td>
</tr>
<tr>
<td>If-Modified-Since</td>
<td></td>
</tr>
<tr>
<td>Referrer</td>
<td>(cause of the request)</td>
</tr>
<tr>
<td>User Agent</td>
<td>(client software)</td>
</tr>
</tbody>
</table>
Uses

Authorization info

Acceptable document types/encoding

From (user email)

Host (identify the server to which the request is sent)

If-Modified-Since

Referrer (cause of the request)

User Agent (client software)
Recall that multiple DNS names can map to the same IP address.

<table>
<thead>
<tr>
<th>Name</th>
<th>IP Address</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.ethz.ch">www.ethz.ch</a></td>
<td>129.132.19.216</td>
</tr>
<tr>
<td>vanbever.eu</td>
<td>82.130.102.71</td>
</tr>
<tr>
<td>route-aggregation.net</td>
<td>82.130.102.71</td>
</tr>
<tr>
<td>comm-net.ethz.ch</td>
<td>82.130.102.71</td>
</tr>
</tbody>
</table>
The "Host" header indicates the server *(82.130.102.71)* the desired domain name *(this is known as virtual hosting)*

<table>
<thead>
<tr>
<th>DNS name</th>
<th>IP address</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.ethz.ch">www.ethz.ch</a></td>
<td>129.132.19.216</td>
</tr>
<tr>
<td>vanbever.eu</td>
<td>82.130.102.71</td>
</tr>
<tr>
<td>route-aggregation.net</td>
<td>82.130.102.71</td>
</tr>
<tr>
<td>comm-net.ethz.ch</td>
<td>82.130.102.71</td>
</tr>
</tbody>
</table>
Virtual hosting enables *one* IP address to host *multiple* websites
connect  openssl s_client -crlf -quiet -connect comm-net.ethz.ch:443

request  GET / HTTP/1.1
        Host: comm-net.ethz.ch

answer  HTTP/1.1 200 OK
        Date: Fri, 01 May 2020 08:36:56 GMT
        Server: Apache/2.4.18 (Ubuntu)

        <head>
        ...
        <title>Communication Networks 2020</title>
        ....
connect: `openssl s_client -crlf -quiet -connect comm-net.ethz.ch:443`

request: GET / HTTP/1.1  
Host: vanbever.eu

answer: HTTP/1.1 200 OK  
Date: Fri, 01 May 2020 08:44:26 GMT  
Server: Apache/2.4.18 (Ubuntu)  

<head>  
...  
	<title>Laurent Vanbever</title>  
...
HTTP servers answers to clients’ requests

<table>
<thead>
<tr>
<th>HTTP response</th>
<th>version</th>
<th>status</th>
<th>phrase</th>
<th>&lt;cr&gt;&lt;lf&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>header field name:</td>
<td>value</td>
<td></td>
<td></td>
<td>&lt;cr&gt;&lt;lf&gt;</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td>&lt;cr&gt;&lt;lf&gt;</td>
</tr>
<tr>
<td>header field name:</td>
<td>value</td>
<td></td>
<td></td>
<td>&lt;cr&gt;&lt;lf&gt;</td>
</tr>
<tr>
<td>&lt;cr&gt;&lt;lf&gt;</td>
<td></td>
<td></td>
<td></td>
<td>&lt;cr&gt;&lt;lf&gt;</td>
</tr>
</tbody>
</table>

body
<table>
<thead>
<tr>
<th>version  &lt;sp&gt; status &lt;sp&gt; phrase</th>
</tr>
</thead>
<tbody>
<tr>
<td>header field name: value</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>header field name: value</td>
</tr>
<tr>
<td>&lt;cr&gt;&lt;lf&gt;</td>
</tr>
</tbody>
</table>

body
<table>
<thead>
<tr>
<th>Status</th>
<th>3 digit response code</th>
<th>reason phrase</th>
</tr>
</thead>
<tbody>
<tr>
<td>1XX</td>
<td>informational</td>
<td></td>
</tr>
<tr>
<td>2XX</td>
<td>success</td>
<td>200 OK</td>
</tr>
<tr>
<td>3XX</td>
<td>redirection</td>
<td>301 Moved Permanently</td>
</tr>
<tr>
<td>4XX</td>
<td>client error</td>
<td>404 Not Found</td>
</tr>
<tr>
<td>5XX</td>
<td>server error</td>
<td>505 Not Supported</td>
</tr>
<tr>
<td>version</td>
<td>status</td>
<td>phrase</td>
</tr>
<tr>
<td>---------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>header field name:</td>
<td>value</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>body</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Like request headers, response headers are of variable lengths and human-readable.

<table>
<thead>
<tr>
<th>Uses</th>
<th>Location (for redirection)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Allow (list of methods supported)</td>
</tr>
<tr>
<td></td>
<td>Content encoding (e.g., gzip)</td>
</tr>
<tr>
<td></td>
<td>Content-Length</td>
</tr>
<tr>
<td></td>
<td>Content-Type</td>
</tr>
<tr>
<td></td>
<td>Expires (caching)</td>
</tr>
<tr>
<td></td>
<td>Last-Modified (caching)</td>
</tr>
</tbody>
</table>
HTTP is a stateless protocol, meaning each request is treated independently.

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server-side scalability</td>
<td>Some applications need state!</td>
</tr>
<tr>
<td>Failure handling is trivial</td>
<td>(shopping cart, user profiles, tracking)</td>
</tr>
</tbody>
</table>

How can you maintain state in a stateless protocol?
HTTP makes the client maintain the state. This is what the so-called **cookies** are for!

- Client stores small state on behalf of the server $X$
- Client sends state in all future requests to $X$
- Can provide authentication
telnet google.ch 80

request
GET / HTTP/1.1
Host: www.google.ch

answer
HTTP/1.1 200 OK
Date: Sun, 01 May 2016 14:10:30 GMT
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
Server: gws

Set-Cookie:
NID=79=g6IgURTq_BG4hSTFhEy1gTVFmSncQVsyTJI260B3xyiXqy2wxD2YeHq1bBlwFyLoJhSc7jmCA6TIFiBY7-dW5lhjiRiQmY1JxT8hGCOnLjfCL0mYcBBkpk8X4NwAO28; expires=Mon, 31-Oct-2016 14:10:30 GMT; path=/; domain=.google.ch; HttpOnly
Performance goals vary depending on who you ask

User
- wish:
  - fast downloads
  - high availability

Network operators
- no overload

Content provider
- happy users
- cost-effective infrastructure

solution
- Improve HTTP to compensate for TCP weakspots

Caching and Replication
<table>
<thead>
<tr>
<th>wish</th>
<th>fast downloads</th>
</tr>
</thead>
<tbody>
<tr>
<td>high availability</td>
<td></td>
</tr>
<tr>
<td>solution</td>
<td>Improve HTTP to compensate for TCP weakspots</td>
</tr>
</tbody>
</table>
Relying on TCP forces a HTTP client to open a connection before exchanging anything.
Most Web pages have multiple objects, naive HTTP opens one TCP connection for each...

Fetching \( n \) objects requires \( \sim 2n \) RTTs

TCP establishment
HTTP request/response
One solution to that problem is to use multiple TCP connections in parallel.

User: Happy!
Content provider: Happy!
Network operator: Not Happy!

Why?
Another solution is to use persistent connections across multiple requests (the default in HTTP/1.1).

Avoid overhead of connection set-up and teardown. Clients or servers can tear down the connection.

Allow TCP to learn more accurate RTT estimate and with it, more precise timeout value.

Allow TCP congestion window to increase and therefore to leverage higher bandwidth.
Yet another solution is to pipeline requests & replies asynchronously, on one connection.

- Batch requests and responses to reduce the number of packets.
- Multiple requests can be packed into one TCP segment.
Considering the time to retrieve $n$ small objects, pipelining wins

\[
\text{\# RTTS} \\
\text{one-at-a-time} & \sim 2n \\
\text{M concurrent} & \sim 2n/M \\
\text{persistent} & \sim n+1 \\
\text{pipelined} & 2
\]
Considering the time to retrieve $n$ big objects, there is no clear winner as bandwidth matters more

\[ \# \text{RTTS} \]

\[ \sim n \times \text{avg. file size} \]

\[ \frac{\text{bandwidth}}{\text{RTTS}} \]
The average webpage size nowadays is 2.3 MB as much as the original DOOM game…

(*) see https://mobiforge.com/research-analysis/the-web-is-doom
Top web sites have decreased in size though because they care about TCP performance

(*) see https://mobiforge.com/research-analysis/the-web-is-doom
User

Network operators

Content provider

wish

no overload

happy users
cost-effective infrastructure

solution

Caching and Replication
Caching leverages the fact that highly popular content largely overlaps.

Just think of how many times you request the Instagram logo per day

versus

how often it actually changes.

Caching it saves time for your browser and decreases network and server load.
Yet, a significant portion of the HTTP objects are “uncachable"

| Examples     | dynamic data | stock prices, scores, ...
|              | scripts      | results based on parameters
|              | cookies      | results may be based on passed data
|              | SSL          | cannot cache encrypted data
|              | advertising  | wants to measure # of hits ($$$)
To limit staleness of cached objects, HTTP enables a client to validate cached objects

Server hints when an object expires (kind of TTL)
as well as the last modified date of an object

Client conditionally requests a resources
using the “if-modified-since” header in the HTTP request

Server compares this against “last modified” time
of the resource and returns:

- Not Modified if the resource has not changed
- OK with the latest version
Caching can and is performed at different locations

- **client**
- **close to the client**
- **close to the destination**

- browser cache
- forward proxy
- Content Distribution Network (CDN)
- reverse proxy
Many clients request the same information
This increases servers and network’s load, while clients experience unnecessary delays.
Reverse proxies cache documents close to servers, decreasing their load.
Forward proxies cache documents close to clients, decreasing network traffic, server load and latencies.

This is typically done by ISPs or enterprises.
Content provider

Network operators

happy users
cost-effective infrastructure

no overload

Caching and Replication

wish

solution
The idea behind replication is to duplicate popular content all around the globe.

- Spreads load on server
  - e.g., across multiple data-centers

- Places content closer to clients
  - only way to beat the “speed-of-light”

- Helps speeding up uncachable content
  - still have to pull it, but from closer
The problem of CDNs is to direct and serve your requests from a close, non-overloaded replica.

- **DNS-based**
  - returns ≠ IP addresses based on:
    - client geo-localization
    - server load

- **BGP Anycast**
  - advertise the same IP prefix from different locations

 avoided in practice, any idea why?
Akamai is one of the largest CDNs in the world, boasting servers in more than 20,000 locations.

http://wwwnui.akamai.com/gnet/globe/index.html
Akamai uses a combination of

- **pull caching**
  direct result of clients requests

- **push replication**
  when expecting high access rate

  together with some dynamic processing

dynamic Web pages, transcoding,…
“Akamaizing” content is easily done by modifying content to reference the Akamai’s domains

Akamai creates domain names for each client
a128.g.akamai.net for cnn.com

Client modifies its URL to refer to Akamai’s domain
http://www.cnn.com/image-of-the-day.gif becomes
http://a128.g.akamai.net/image-of-the-day.gif

Requests are now sent to the CDN infrastructure