Unterrichtsbeurteilung
aka course evaluation

Please fill in the survey!
You should have received the link by email
Two weeks ago on
Communication Networks
TCP Congestion Control
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

Flow control prevents one fast sender from overloading a slow receiver

Congestion control prevents a set of senders from overloading the network
The sender adapts its sending rate based on these two windows.

- **Receiving Window (RWND):** How many bytes can be sent without overflowing the receiver buffer? based on the receiver input.
- **Congestion Window (CWND):** How many bytes can be sent without overflowing the routers? based on network conditions.
- **Sender Window:** minimum(CWND, RWND)
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

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
#1 bandwidth estimation

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**
\[
\text{cwnd} = 1 \quad \text{initially}
\]
\[
\text{cwnd} += 1 \quad \text{upon receipt of an ACK}
\]
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)
TCP congestion control in less than 10 lines of code

Initially:
   cwnd = 1
   ssthresh = infinite

New ACK received:
   if (cwnd < ssthresh):
     /* Slow Start*/
     cwnd = cwnd + 1
   else:
     /* Congestion Avoidance */
     cwnd = cwnd + 1/cwnd

Timeout:
   /* Multiplicative decrease */
   ssthresh = cwnd/2
   cwnd = 1
The congestion window of a TCP session typically undergoes multiple cycles of slow-start/AIMD.
Going back all the way back to 0 upon timeout completely destroys throughput

solution

Avoid timeout expiration…

which are usually >500ms
Detecting losses can be done using **ACKs** or timeouts, the two signal differ in their degree of severity

<table>
<thead>
<tr>
<th>duplicated ACKs</th>
<th>mild congestion signal</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>packets are still making it</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>timeout</th>
<th>severe congestion signal</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>multiple consequent losses</td>
</tr>
</tbody>
</table>
TCP automatically resends a segment after receiving 3 duplicates ACKs for it. This is known as a “fast retransmit”
After a fast retransmit, TCP switches back to AIMD, without going all the way back to 0. This is known as "fast recovery".
TCP congestion control (almost complete)

Initially:
  \(cwnd = 1\)
  \(ssthresh = \text{infinite}\)

New ACK received:
  if \((cwnd < ssthresh)\):
    /* Slow Start*/
    \(cwnd = cwnd + 1\)
  else:
    /* Congestion Avoidance */
    \(cwnd = cwnd + 1/cwnd\)
    \(dup\_ack = 0\)

Timeout:
  /* Multiplicative decrease */
  \(ssthresh = cwnd/2\)
  \(cwnd = 1\)

Duplicate ACKs received:
  \(dup\_ack ++;\)
  if \((dup\_ack >= 3)\):
    /* Fast Recovery */
    \(ssthresh = cwnd/2\)
    \(cwnd = ssthresh\)
Initially:

cwnd = 1
ssthresh = infinite

New ACK received:

if (cwnd < ssthresh):
  /* Slow Start*/
  cwnd = cwnd + 1
else:
  /* Congestion Avoidance */
  cwnd = cwnd + 1/cwnd
dup_ack = 0

Timeout:

/* Multiplicative decrease */
ssthresh = cwnd/2
cwnd = 1

Duplicate ACKs received:

dup_ack ++;
if (dup_ack >= 3):
  /* Fast Recovery */
  ssthresh = cwnd/2
cwnd = ssthresh
Congestion control makes TCP throughput look like a “sawtooth”
We now have completed the transport layer (!)

- **Application**: HTTP(S)
  - Reliable (or unreliable) transport
- **Transport**: TCP/UDP
  - Best-effort global packet delivery
- **Network**: IP
  - Best-effort local packet delivery
- **Link**: Ethernet
This week on

Communication Networks
google.ch 172.217.16.131  http://www.google.ch
google.ch  ➔  172.217.16.131
Internet has one **global system** for

- **addressing hosts** (IP) by design
- **naming hosts** (DNS) by "accident", an afterthought
Internet has one global system for

- naming hosts  
  by "accident", an afterthought  
  DNS
Using Internet services can be divided into four logical steps

- **Step 1**: A person has name of entity she wants to access (www.ethz.ch)

- **Step 2**: She invokes an application to perform the task (Chrome)

- **Step 3**: The application invokes DNS to resolve the name into an IP address (129.132.19.216)

- **Step 4**: The application invokes transport protocol to establish an app-to-app connection
The DNS system is a distributed database which enables to resolve a name into an IP address

```
name               DNS               IP address

www.ethz.ch        129.132.19.216
```
In practice, names can be mapped to more than one IP address.

- `www.ethz.ch` maps to `129.132.19.216`.
In practice,
IPs can be mapped by more than one name

<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><a href="http://www.vanbever.eu">www.vanbever.eu</a></td>
<td>188.165.240.60</td>
</tr>
<tr>
<td><a href="http://www.routeur.be">www.routeur.be</a></td>
<td>188.165.240.60</td>
</tr>
</tbody>
</table>
How does one resolve a name into an IP?

Initially, all host to address mappings were in a file called hosts.txt in /etc/hosts.

However, there were problems with scalability in terms of query load & speed management, consistency, and availability.
When you need... more flexibility,
you add... a layer of indirection

When you need... more scalability,
you add... a hierarchical structure
To scale, DNS adopt three intertwined hierarchies

<table>
<thead>
<tr>
<th>naming structure</th>
<th>hierarchy of addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><a href="https://www.ee.ethz.ch/de/departement/">https://www.ee.ethz.ch/de/departement/</a></td>
</tr>
</tbody>
</table>

| management        | hierarchy of authority over names |

| infrastructure    | hierarchy of DNS servers |
naming structure
addresses are hierarchical
https://www.ee.ethz.ch/de/departement/
Top Level Domain (TLDs) sit at the top

```
root “.”
```

- `com`
- `org`
- `net`
- `edu`
- `gov`
- `mil`
- `be`
- `ch`
- `de`
- `fr`

+ many more
Domains are subtrees

com   org   net   edu   gov   mil   be  ch  de  fr

root ".

epfl  ethz  nzz

+ many more
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.
managed by IANA (*)

com    org    net    edu    gov    mil    be    ch    de    fr

root

epfl    ethz    nzz

www    ee    infk

(*) see http://www.iana.org/domains/root/db
managed by The Swiss Education & Research Network (*)

(*) see https://www.switch.ch/about/id/
Hierarchical administration means that name collision is trivially avoided.
infrastructure hierarchy of DNS servers
The DNS infrastructure is hierarchically organized
13 root servers (managed professionally) serve as root (*)

(*) see http://www.root-servers.org/
a. root-servers.net  VeriSign, Inc.
b. root-servers.net  University of Southern California
c. root-servers.net  Cogent Communications
d. root-servers.net  University of Maryland
e. root-servers.net  NASA
f. root-servers.net  Internet Systems Consortium
g. root-servers.net  US Department of Defense
h. root-servers.net  US Army
i. root-servers.net  Netnod
j. root-servers.net  VeriSign, Inc.
k. root-servers.net  RIPE NCC
l. root-servers.net  ICANN
m. root-servers.net  WIDE Project
To scale root servers, operators rely on **BGP anycast**

**Intuition**

Routing finds shortest-paths

If several locations announce the same prefix, then routing will deliver the packets to the “closest” location

This enables seamless replications of resources
Do you see any problems in performing load-balancing this way?
Instances of the k-root server (*) are hosted in more than 40 locations worldwide.

(*) see k.root-servers.org
Two of these locations are in Switzerland: in Zürich and in Geneva.
All locations announce \textit{193.0.14.0/23} in BGP, with \textit{193.0.14.129} being the IP of the server.
Two of these locations are in Switzerland: in Zürich and in Geneva

Do you mind guessing which one we use, here… in Zürich?
Each instance receives up to 70k queries per second summing up to more than 4 billions queries per day.
TLDs server are also managed professionally by private or non-profit organization
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 root server knows the address of all TLD servers

lvanbever:~$ dig @a.root-servers.net ch.

ch.  172800 IN  NS  a.nic.ch.
ch.  172800 IN  NS  b.nic.ch.
ch.  172800 IN  NS  c.nic.ch.
ch.  172800 IN  NS  d.nic.ch.
ch.  172800 IN  NS  e.nic.ch.
ch.  172800 IN  NS  f.nic.ch.
ch.  172800 IN  NS  h.nic.ch.
From there on, each server knows the address of all children.
Any .ch DNS server knows the addresses of all sub-domains
To scale, DNS adopts three intertwined hierarchies.

- **naming structure**: addresses are hierarchical
  - https://www.ee.ethz.ch/de/departement/
- **management**: hierarchy of authority over names
- **infrastructure**: hierarchy of DNS servers
To ensure availability, each domain must have at least a primary and secondary DNS server.

Ensure name service availability as long as one of the servers is up.

DNS queries can be load-balanced across the replicas.

On timeout, client use alternate servers exponential backoff when trying the same server.
Overall, the DNS system is highly scalable, available, and extensible

**scalable**
- #names, #updates, #lookups, #users,
- but also in terms of administration

**available**
- domains replicate independently of each other

**extensible**
- any level (including the TLDs)
- can be modified independently
You’ve founded next-startup.ch and want to host it yourself, how do you insert it into the DNS?

You register next-startup.ch at a registrar X

*e.g.* Swisscom or GoDaddy

Provide X with the name and IP of your DNS servers

*e.g.*, [ns1.next-startup.ch,129.132.19.253]

You set-up a DNS server @129.132.19.253

define A records for www, MX records for next-startup.ch...
Using DNS relies on two components

- **resolver software**
  - trigger resolution process
  - send request to local DNS server

- **local DNS server**
  - usually, near the endhosts
  - configured statically *(resolv.conf)*
  - or dynamically *(DHCP)*

The diagram shows the process with `gethostbyname()` function.
DNS query and reply uses UDP (port 53), reliability is implemented by repeating requests (*)

(*) see Book (Section 5)
A DNS server stores Resource Records composed of a (name, value, type, TTL)
<table>
<thead>
<tr>
<th>Records</th>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>hostname</td>
<td>IP address</td>
</tr>
<tr>
<td>NS</td>
<td>domain</td>
<td>DNS server name</td>
</tr>
<tr>
<td>MX</td>
<td>domain</td>
<td>Mail server name</td>
</tr>
<tr>
<td>CNAME</td>
<td>alias</td>
<td>canonical name</td>
</tr>
<tr>
<td>PTR</td>
<td>IP address</td>
<td>corresponding hostname</td>
</tr>
</tbody>
</table>
DNS resolution can either be recursive or iterative
When performing a recursive query, the client offload the task of resolving to the server
local DNS server
(dns1.ethz.ch)

DNS client
(me.ee.ethz.ch)

root servers

.edu servers

.edu servers

Dns1.ethz.ch

www.nyu.edu?

Nyu.edu servers
DNS client (me.ee.ethz.ch) asks for www.nyu.edu? which is then passed to the local DNS server (dns1.ethz.ch). The local DNS server asks the root DNS server for the location of the .edu servers. The root DNS server then directs to the .edu servers, which respond with the information for nyu.edu servers.
www.nyu.edu?

DNS client (me.ee.ethz.ch)

local DNS server (dns1.ethz.ch)

root DNS server

.edu servers

nyu.edu servers
When performing a iterative query, the server only returns the address of the next server to query.
Where is .edu?

Where is nyu.edu?

Where is www.nyu.edu?

DNS client
(me.ee.ethz.ch)

.edu servers

nyu.edu servers

root

DNS server

local

DNS server

(me.ee.ethz.ch)
To reduce resolution times, DNS relies on caching.

DNS servers cache responses to former queries and your client and the applications (!)

Authoritative servers associate a lifetime to each record Time-To-Live (TTL)

DNS records can only be cached for TTL seconds after which they must be cleared
As top-level servers rarely change & popular website visited often, caching is very effective (*)

Top 10% of names account for 70% of lookups

9% of lookups are unique
Limit cache hit rate to 91%

Practical cache hit rates ~75%

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**
  - Clients/Browser
  - Servers
  - Proxies

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

- **Implementation**
  - 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
**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
HTTP is a rather simple synchronous request/reply protocol

HTTP is layered over a bidirectional byte stream
almost always TCP

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://

Performance

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>
</tbody>
</table>

body
<table>
<thead>
<tr>
<th>method</th>
<th>URL</th>
<th>version</th>
</tr>
</thead>
<tbody>
<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>header field name:</td>
<td>value</td>
<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

HTTP request

| method <sp> URL <sp> version         <cr><lf> |
|-------------------------------------|------|
| header field name: value            <cr><lf> |
| ...                                 |      |
| header field name: value            <cr><lf> |
| <cr><lf>                            |      |

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

Uses

Authorization info

Acceptable document types/encoding

From (user email)

If-Modified-Since

Referrer (cause of the request)

User Agent (client software)
# HTTP servers answers to clients’ requests

<table>
<thead>
<tr>
<th>HTTP response</th>
</tr>
</thead>
</table>
| version  status  phrase  
| header field name:  value  
| ...  
| header field name:  value  
| body  

- The **version** field indicates the version of the HTTP protocol used.
- The **status** field is a three-digit code indicating the result of the request.
- The **phrase** field provides a text description of the status code.
- The **header field name: value** pairs contain additional information about the request or response.

Each header field is followed by a newline character (<cr><lf>) to separate different fields.
<table>
<thead>
<tr>
<th>version</th>
<th>status</th>
<th>phrase</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

header field name: value

...  

header field name: value

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></td>
<td></td>
<td>303 Moved Temporarily</td>
</tr>
<tr>
<td></td>
<td></td>
<td>304 Not Modified</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>
</tbody>
</table>

header field name: value

... 

header field name: value

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

**Advantages**

- Server-side scalability
- Failure handling is trivial

**Disadvantages**

- Some applications need state! (shopping cart, user profiles, tracking)

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=g6lgURTq_BG4hSTFhEy1gTVFmSncQVsyTJI260B3xyiXqy2wxD2YeHq1bBlwFyLoJhSc7jmcA6TIFIBY7-dW5IhjiRiQmY1JxT8hGCOtnLjfCL0mYcBBkpk8X4NwAO28; 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
User

wish
  fast downloads
  high availability

solution
  Improve HTTP to compensate for TCP weakspots
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 \(~2n\) RTTs

TCP establishment
HTTP request/response
One solution to that problem is to use multiple TCP connections in parallel.
Another solution is to use persistent connections across multiple requests, 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

<table>
<thead>
<tr>
<th># RTTS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>one-at-a-time</td>
<td>$\sim 2n$</td>
</tr>
<tr>
<td>M concurrent</td>
<td>$\sim 2n/M$</td>
</tr>
<tr>
<td>persistent</td>
<td>$\sim n+1$</td>
</tr>
<tr>
<td>pipelined</td>
<td>2</td>
</tr>
</tbody>
</table>
Considering the time to retrieve \( n \) big objects, there is no clear winners as bandwidth matters more

\[
\text{\# RTTS}
\]

\[
\sim n \times \text{avg. file size} \over \text{bandwidth}
\]
Today, the average webpage size 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 Facebook logo per day

vs

how often it actually changes

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

<table>
<thead>
<tr>
<th>Examples</th>
<th>dynamic data</th>
<th>stock prices, scores, ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>scripts</td>
<td>results based on parameters</td>
<td></td>
</tr>
<tr>
<td>cookies</td>
<td>results may be based on passed data</td>
<td></td>
</tr>
<tr>
<td>SSL</td>
<td>cannot cache encrypted data</td>
<td></td>
</tr>
<tr>
<td>advertising</td>
<td>wants to measure # of hits ($$$)</td>
<td></td>
</tr>
</tbody>
</table>
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

This is typically done by content provider
Forward proxies cache documents close to clients, decreasing network traffic, server load and latencies.

This is typically done by ISPs or enterprises.
Network operators wish for no overload while content providers strive for happy users. A cost-effective infrastructure solution is provided by Caching and Replication.
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.
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