Communication Networks
Spring 2019

Laurent Vanbever
nsg.ee.ethz.ch

ETH Zürich (D-ITET)
April 15 2019

Materials inspired from Scott Shenker & Jennifer Rexford
Last week on
Communication Networks
We started to look at the transport layer.
What Problems Should Be Solved Here?

Data delivering, to the *correct* application

- IP just points towards next protocol
- *Transport needs to demultiplex incoming data (ports)*

Files or bytestreams abstractions for the applications

- Network deals with packets
- *Transport layer needs to translate between them*

Reliable transfer *(if needed)*

Not overloading the receiver

Not overloading the network
What Is Needed to Address These?

Demultiplexing: identifier for application process
  - Going from host-to-host (IP) to process-to-process

Translating between bytestreams and packets:
  - Do segmentation and reassembly

Reliability: ACKs and all that stuff

Corruption: Checksum

Not overloading receiver: “Flow Control”
  - Limit data in receiver’s buffer

Not overloading network: “Congestion Control”
UDP: User Datagram Protocol

Lightweight communication between processes

- Avoid overhead and delays of ordered, reliable delivery
- Send messages to and receive them from a socket

UDP described in RFC 768 – (1980!)

- IP plus port numbers to support (de)multiplexing
- Optional error checking on the packet contents
  - (checksum field = 0 means “don’t verify checksum”)

<table>
<thead>
<tr>
<th>SRC port</th>
<th>DST port</th>
</tr>
</thead>
<tbody>
<tr>
<td>checksum</td>
<td>length</td>
</tr>
<tr>
<td>DATA</td>
<td></td>
</tr>
</tbody>
</table>
Transmission Control Protocol (TCP)

Reliable, in-order delivery
  • Ensures byte stream (eventually) arrives intact
    • In the presence of corruption and loss

Connection oriented
  • Explicit set-up and tear-down of TCP session

Full duplex stream-of-bytes service
  • Sends and receives a stream of bytes, not messages

Flow control
  • Ensures that sender doesn’t overwhelm receiver

Congestion control
  • Dynamic adaptation to network path’s capacity
## TCP Header

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence number</td>
<td></td>
</tr>
<tr>
<td>Acknowledgment</td>
<td></td>
</tr>
<tr>
<td>HdrLen</td>
<td>Flags</td>
</tr>
<tr>
<td>Checksum</td>
<td>Urgent pointer</td>
</tr>
<tr>
<td>Options (variable)</td>
<td></td>
</tr>
</tbody>
</table>

**Data**
This week on
Communication Networks
DNS Congestion Control

google.ch  <->  172.217.16.131
Congestion Control

DNS
Because of traffic burstiness and lack of BW reservation, congestion is inevitable.

If many packets arrive within a short period of time, the node cannot keep up anymore.
Congestion is harmful
average packet arrival rate \( a \) [packet/sec]

transmission rate of outgoing link \( R \) [bit/sec]

fixed packets length \( L \) [bit]

average bits arrival rate \( La \) [bit/sec]

traffic intensity \( La/R \)
When the traffic intensity is >1, the queue will increase without bound, and so does the queuing delay

Golden rule

Design your queuing system, so that it operates far from that point
When the traffic intensity is $\leq 1$, queueing delay depends on the burst size.
Congestion is not a new problem

The Internet almost died of congestion in 1986 throughput collapsed from 32 Kbps to... 40 bps

Van Jacobson saved us with Congestion Control his solution went right into BSD

Recent resurgence of research interest after brief lag new methods (ML), context (Data centers), requirements
The Internet almost died of congestion in 1986. Throughput collapsed from 32 Kbps to... 40 bps.
<table>
<thead>
<tr>
<th>original behavior</th>
<th>On connection, nodes send full window of packets</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Upon timer expiration, retransmit packet immediately</td>
</tr>
<tr>
<td>meaning</td>
<td>sending rate only limited by flow control</td>
</tr>
<tr>
<td>net effect</td>
<td>window-sized burst of packets</td>
</tr>
</tbody>
</table>
Increase in network load results in a decrease of useful work done

Sudden load increased the round-trip time (RTT) faster than the hosts’ measurements of it

As RTT exceeds the maximum retransmission interval, hosts begin to retransmit packets

Hosts are sending each packet several times, eventually some copies arrive at the destination.

This phenomenon is known as congestion collapse
Knee point after which throughput increases slowly and delay increases quickly.

Cliff point after which throughput decreases quickly and delay tends to infinity.

Diagram showing the knee and cliff points with corresponding changes in throughput and delay as a function of load.
Van Jacobson saved us with **Congestion Control**

his solution went right into BSD
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.
TCP solves both using two distinct windows

Flow control: prevents one fast sender from overloading a slow receiver

Congestion control: prevents a set of senders from overloading the network

solved using a receiving window

solved using a “congestion” window
The sender adapts its sending rate based on these two windows:

<table>
<thead>
<tr>
<th>Window Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receiving Window</td>
<td>How many bytes can be sent without overflowing the receiver buffer?</td>
</tr>
<tr>
<td>RWND</td>
<td>based on the receiver input</td>
</tr>
<tr>
<td>Congestion Window</td>
<td>How many bytes can be sent without overflowing the routers?</td>
</tr>
<tr>
<td>CWND</td>
<td>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
There are essentially three ways to detect congestion

**Approach #1**
Network could tell the source but signal itself could be lost

**Approach #2**
Measure packet delay but signal is noisy
delay often varies considerably

**Approach #3**
Measure packet loss fail-safe signal that TCP already has to detect
Packet dropping is the best solution
delay- and signaling-based methods are hard & risky

Approach #3

Measure packet loss
fail-safe signal that TCP already has to detect
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...
Remember that 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
How to adjust the bandwidth of a single flow to the bottleneck bandwidth?

could be 1 Mbps or 1 Gbps…
The goal here is 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
This increase phase, known as slow start, corresponds to an exponential increase of CWND!

Slow start is called like this only because of starting point.
The problem with slow start is that it can result in a full window of packet losses.

<table>
<thead>
<tr>
<th>Example</th>
<th>Assume that CWND is just enough to “fill the pipe”</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>After one RTT, CWND has doubled</td>
</tr>
<tr>
<td></td>
<td>All the excess packets are now dropped</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solution</th>
<th>We need a more gentle adjustment algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>once we have a rough estimate of the bandwidth</td>
</tr>
</tbody>
</table>
How to adjust the bandwidth of a single flow to variation of the bottleneck bandwidth?
The goal here is to track the available bandwidth, and oscillate around its current value

Two possible variations

- **Multiplicative Increase or Decrease**
  \[ \text{cwnd} = a \times \text{cwnd} \]

- **Additive Increase or Decrease**
  \[ \text{cwnd} = b + \text{cwnd} \]

... leading to four alternative design
<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>
To select one scheme, we need to consider the 3rd problem: **fairness**

<table>
<thead>
<tr>
<th>Scheme</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>
#3 fairness How to share bandwidth “fairly" among flows, without overloading the network
TCP notion of fairness: 2 identical flows should end up with the same bandwidth
Consider first a single flow between A and B and AIMD

capacity 50 pkts/RTT

host A queue (20 pkts) host B

without congestion CWND increases by one packet every ACK
upon congestion CWND decreases by a factor 2
Rate (pkts/RTT)

Backlog in router (pkts)
Congested if > 20
We can analyze the system behavior using a system trajectory plot.
The system is efficient if the capacity is fully used, defining an efficiency line where $a + b = 1$.

![Graph showing a straight line with link capacity of 1 Mbps, where efficiency line is $a + b = 1$.]
The goal of congestion control is to bring the system as close as possible to this line, and stay there.
A’s throughput

under-utilization

B’s throughput
The system is fair whenever A and B have equal throughput, defining a fairness line where \( a = b \).
A’s throughput

B’s throughput

1

fairness line

B gets more than A
A’s throughput

B’s throughput

A gets more than B

fairness line
A’s throughput

1

B’s throughput

efficiency line

fairness line

efficient & unfair .3

.7
<table>
<thead>
<tr>
<th>System</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>
AIAD does not converge to fairness, nor efficiency: the system fluctuates between two fairness states.

Adding a constant:
move along 45 deg
AIAD does not converge to fairness, nor efficiency: the system fluctuates between two fairness states.
<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>
MIMD does not converge to fairness, nor efficiency: the system fluctuates along a equi-fairness line.
<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>
MIAD converges to a totally unfair allocation, favoring the flow with a greater rate at the beginning.
If flows start along the fairness line, MIAD fluctuates along it, yet deviating from it at the slightest change.
<table>
<thead>
<tr>
<th></th>
<th>Increase Behavior</th>
<th>Decrease Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AIAD</strong></td>
<td>gentle</td>
<td>gentle</td>
</tr>
<tr>
<td><strong>AIMD</strong></td>
<td>gentle</td>
<td>aggressive</td>
</tr>
<tr>
<td><strong>MIAD</strong></td>
<td>aggressive</td>
<td>gentle</td>
</tr>
<tr>
<td><strong>MIMD</strong></td>
<td>aggressive</td>
<td>aggressive</td>
</tr>
</tbody>
</table>
AIMD converge to fairness and efficiency, it then fluctuates around the optimum (in a stable way)
AIMD converge to fairness and efficiency, it then fluctuates around the optimum (in a stable way)

Intuition

During increase,
both flows gain bandwidth at the same rate

During decrease,
the faster flow releases more
AIMD converge to fairness and efficiency, it then fluctuates around the optimum (in a stable way)
In practice, TCP implements AIMD

<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>
In practice, TCP implements AIMD

**Implementation**

After each ACK,

increment cwnd by $1/cwnd$

linear increase of max. 1 per RTT

**Question**

When does a sender leave slow-start and start AIMD?

Introduce a slow start threshold, adapt it in function of congestion:

on timeout, $sstresh = CNWD/2$
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 = infinite

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

Duplicate ACKs received:
   dup_ack ++;
   if (dup_ack >= 3):
      /* Fast Recovery */
      ssthresh = cwnd/2
      cwnd = ssthresh

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

- **HTTP(S)**
- **TCP/UDP**
- **IP**
- **Ethernet**

- Reliable (or unreliable) transport
- Best-effort global packet delivery
- Best-effort local packet delivery
DNS Congestion Control

google.ch ← 172.217.16.131
Internet has one global system for

- **addressing hosts**  
  by design  

- **naming hosts**  
  by "accident", an afterthought
Internet has one global system for

- **naming** hosts
- DNS

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

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A person has name of entity she wants to access</td>
<td><a href="http://www.ethz.ch">www.ethz.ch</a></td>
</tr>
<tr>
<td>2</td>
<td>She invokes an application to perform the task</td>
<td>Chrome</td>
</tr>
<tr>
<td>3</td>
<td>The application invokes DNS to resolve the name into an IP address</td>
<td>129.132.19.216</td>
</tr>
<tr>
<td>4</td>
<td>The application invokes transport protocol to establish an app-to-app connection</td>
<td></td>
</tr>
</tbody>
</table>
The DNS system is a distributed database which enables to resolve a name into an IP address.

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

<table>
<thead>
<tr>
<th>Name</th>
<th>IP Address</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.ethz.ch">www.ethz.ch</a></td>
<td>129.132.19.216</td>
<td></td>
</tr>
<tr>
<td><a href="http://www.netflix.com">www.netflix.com</a></td>
<td>52.31.246.79</td>
<td>(load-balancing)</td>
</tr>
<tr>
<td></td>
<td>52.49.6.246</td>
<td></td>
</tr>
<tr>
<td></td>
<td>52.50.212.245</td>
<td>+5 more!</td>
</tr>
</tbody>
</table>
In practice, IPs can be mapped by more than one name.
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.

Problem: scalability in terms of query load & speed management, consistency, 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 adopts **three** intertwined hierarchies:

<table>
<thead>
<tr>
<th><strong>naming structure</strong></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 "."
Domains are subtrees
A name, \textit{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 (*)

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

(*) see https://www.switch.ch/about/id/
managed by

ETH Zürich
Informatikdienste ICT-Networks
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
IP traffic
193.0.14.129

Deutsche Telekom

at&t

ETH

skynet

swisscom
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.

Swiss Internet Exchange
ns1.ch-zrh.k.ripe.net

CERN
ns1.ch-gva.k.ripe.net
All locations announce 193.0.14.0/23 in BGP, with 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

```
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

```
com  org  net  edu  gov  mil  be  ch  de  fr
  
epfl  ethz  nzz
  
www  ee  infk
```
To scale, DNS adopt **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.

<table>
<thead>
<tr>
<th>Scalable</th>
<th>#names, #updates, #lookups, #users, but also in terms of administration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available</td>
<td>domains replicate independently of each other</td>
</tr>
<tr>
<td>Extensible</td>
<td>any level (including the TLDs) can be modified independently</td>
</tr>
</tbody>
</table>
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
- local DNS server

The resolver software triggers the resolution process by sending a request to the local DNS server. Typically, the local DNS server is configured either statically in `resolv.conf` or dynamically via DHCP.
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
DNS client
(m.e.e.ee.ethz.ch)

local
DNS server
(dns1.ethz.ch)

www.nyu.edu?

DNS server

root

.edu servers

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

- Root DNS server
- Local DNS server
- DNS client (me.ee.ethz.ch)
- .edu servers
- nyu.edu servers
Where is .edu?

Where is www.nyu.edu?

Where is nyu.edu?
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%
