Internet Architecture

A review

Lecture 3

Srinivas Narayana

http://www.cs.rutgers.edu/~sn624/553-S23

RUTGERS
UNIVERSITY | NEW BRUNSWICK

Software/hardware organization at hosts

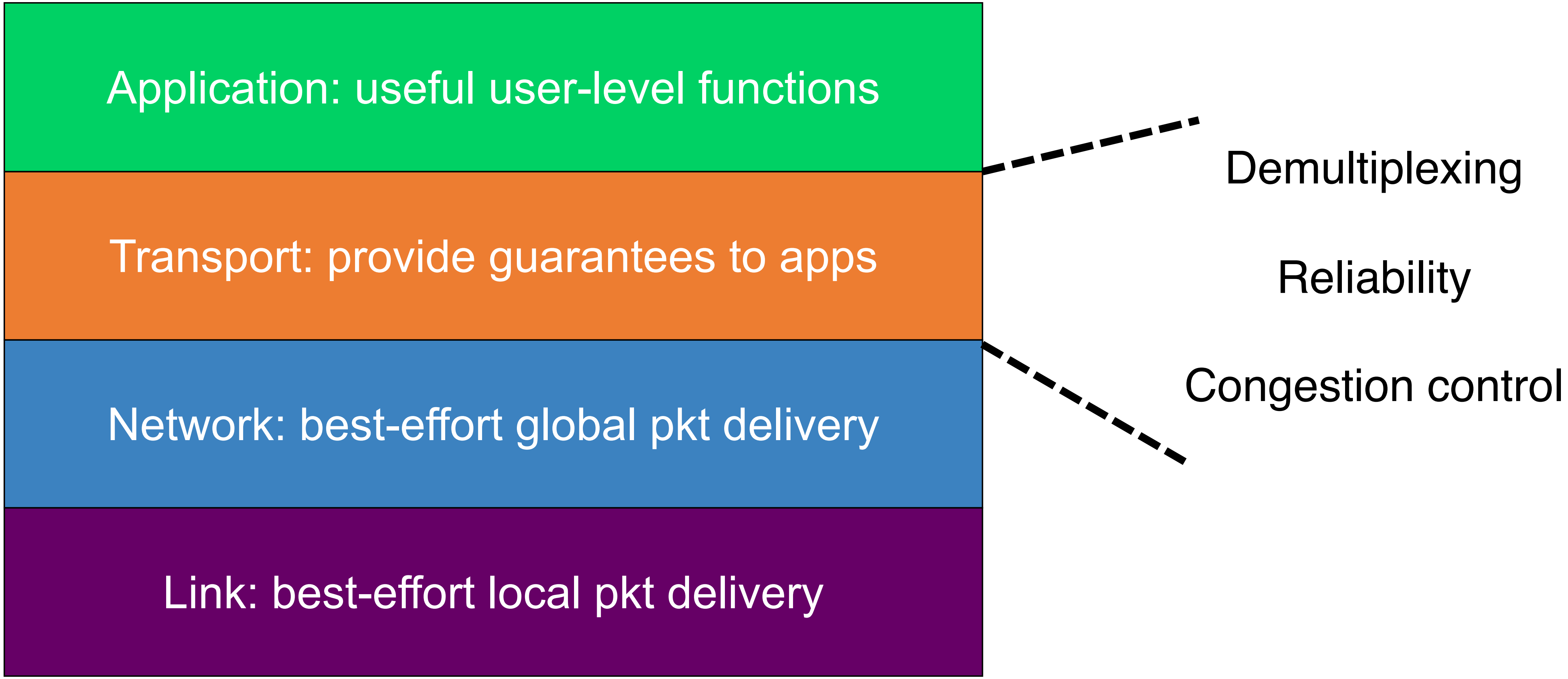
Communication functions broken up and “stacked”

Each layer depends on the one below it.

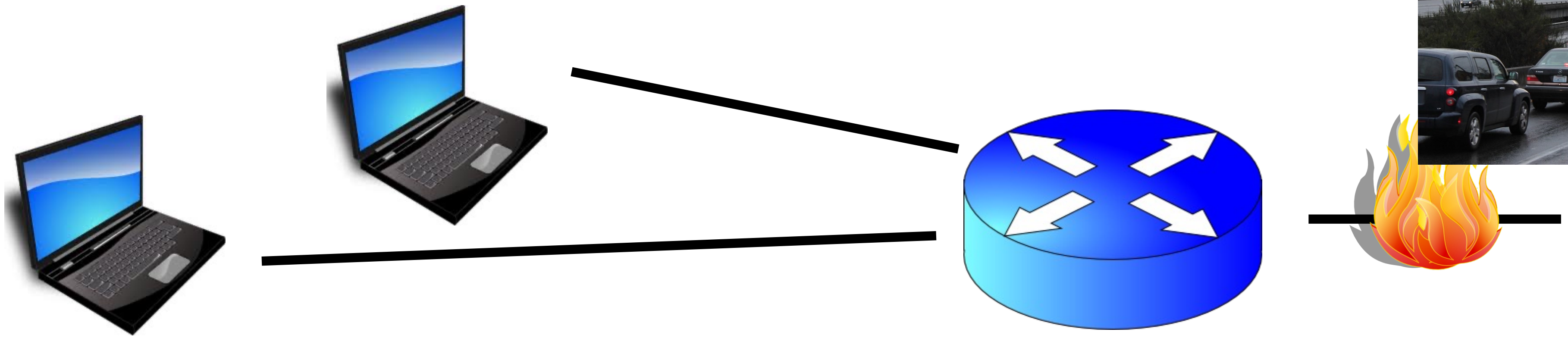
Each layer supports the one above it.

The interfaces between layers are well-defined and standardized.

Software/hardware organization at hosts



(3) How much data to keep in flight?



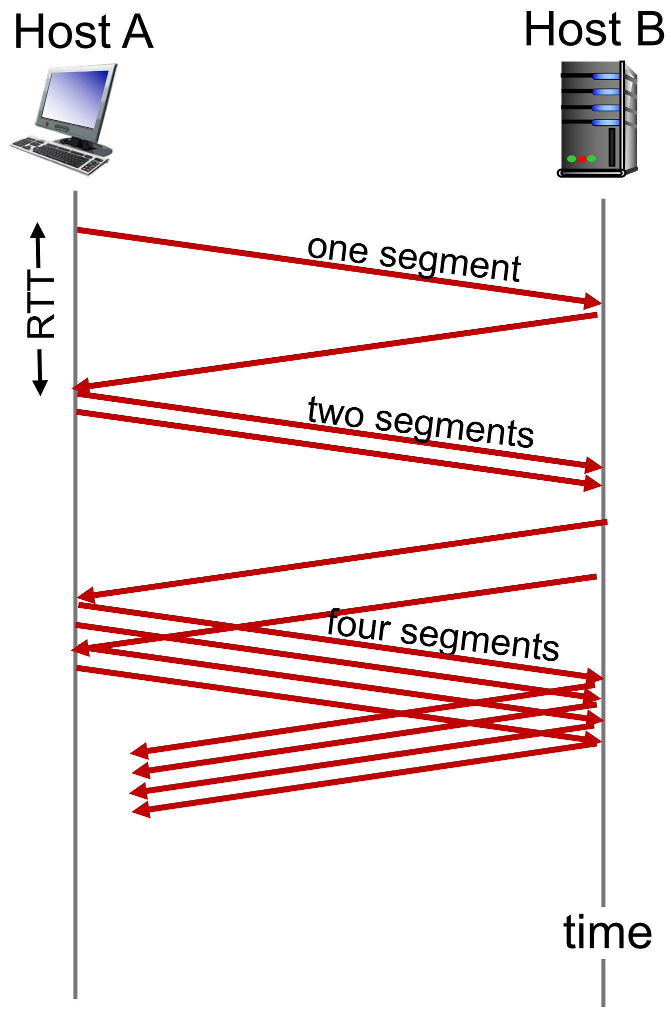
- Avoid overwhelming network resources: Congestion control
- Internet: every endpoint makes its own decisions!
  - Distributed algorithm: no central authority
  - Goal 1: efficiency (use available capacity)
  - Goal 2: fairness (distribute capacity equitably)

Feedback Control

Finding the right congestion window

- There is an unknown bottleneck link rate that the sender must match
- If sender sends more than the bottleneck link rate:
  - packet loss, delays, etc.
- If sender sends less than the bottleneck link rate:
  - all packets get through; successful ACKs
- Congestion window (`cwnd`): amount of data in flight

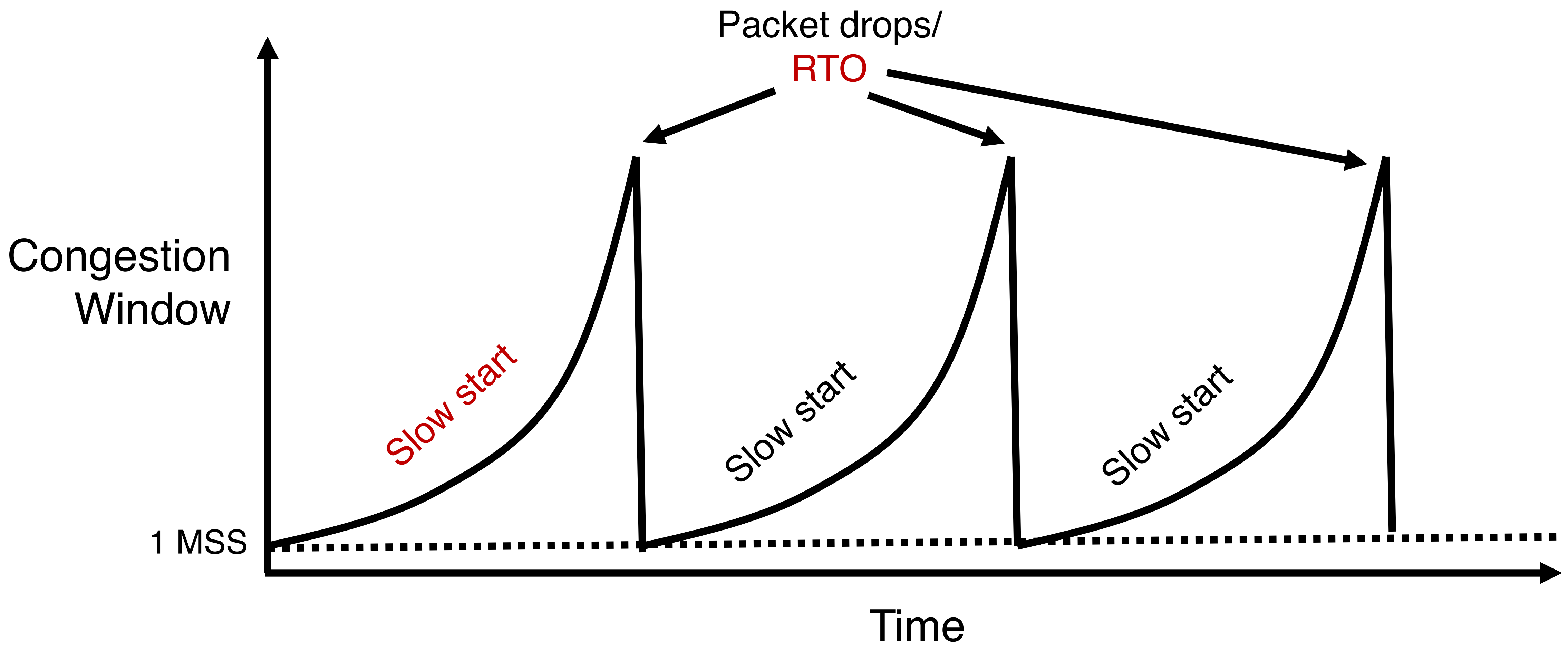
Quickly finding a rate: TCP slow start



- Initially `cwnd` = 1 MSS
  - MSS is "maximum segment size"
- Upon receiving an ACK of each MSS, increase the `cwnd` by 1 MSS
- Effectively, double `cwnd` every RTT

- Initial rate is slow but ramps up exponentially fast
- On loss (RTO), restart from `cwnd := 1 MSS`

Behavior of slow start

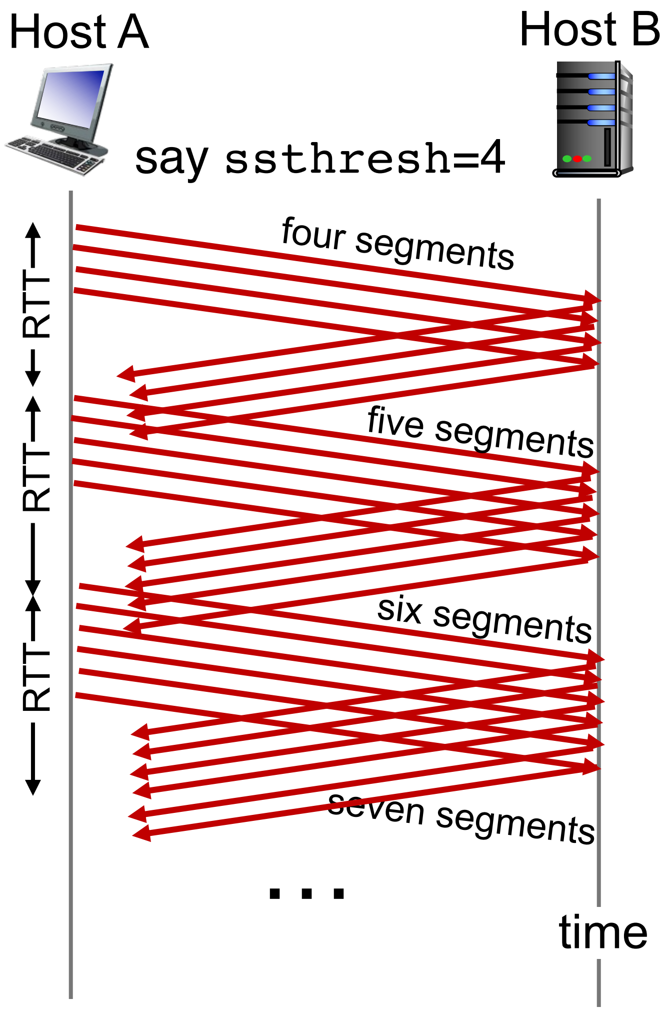


Slow start has problems

- Congestion window increases too rapidly
  - Example: suppose the “right” window size `cwnd` is 17
  - `cwnd` would go from 16 to 32 and then dropping down to 1
  - Result: massive packet drops
- Congestion window decreases too rapidly
  - Suppose the right `cwnd` is 31, and there is a loss when `cwnd` is 32
  - Slow start will resume all the way back from `cwnd` 1
  - Result: unnecessarily low speed of sending data
- Instead, perform finer adjustments of `cwnd`: congestion avoidance

TCP New Reno: Additive Increase

- Remember the recent past to find a good estimate of link rate
- The last good `cwnd` without packet drop is a good indicator
  - TCP New Reno calls this the slow start threshold (`ssthresh`)
- Increase `cwnd` by 1 MSS every RTT after `cwnd` hits `ssthresh`
  - Effect: increase window additively per RTT



TCP New Reno: Additive increase

- Start with `ssthresh = 64K bytes` (TCP default)
- Do slow start until `ssthresh`
- Once the threshold is passed, do additive increase
  - Add one MSS to `cwnd` for each `cwnd` worth data ACK'ed
  - For each MSS ACK'ed, `cwnd = cwnd + (MSS * MSS) / cwnd`
- Upon a TCP timeout (RTO),
  - Set `cwnd = 1 MSS`
  - Set `ssthresh = max(2 * MSS, 0.5 * cwnd)`
  - i.e., the next linear increase will start at half the current `cwnd`

Behavior of Additive Increase

Say `MSS` = 1 KByte

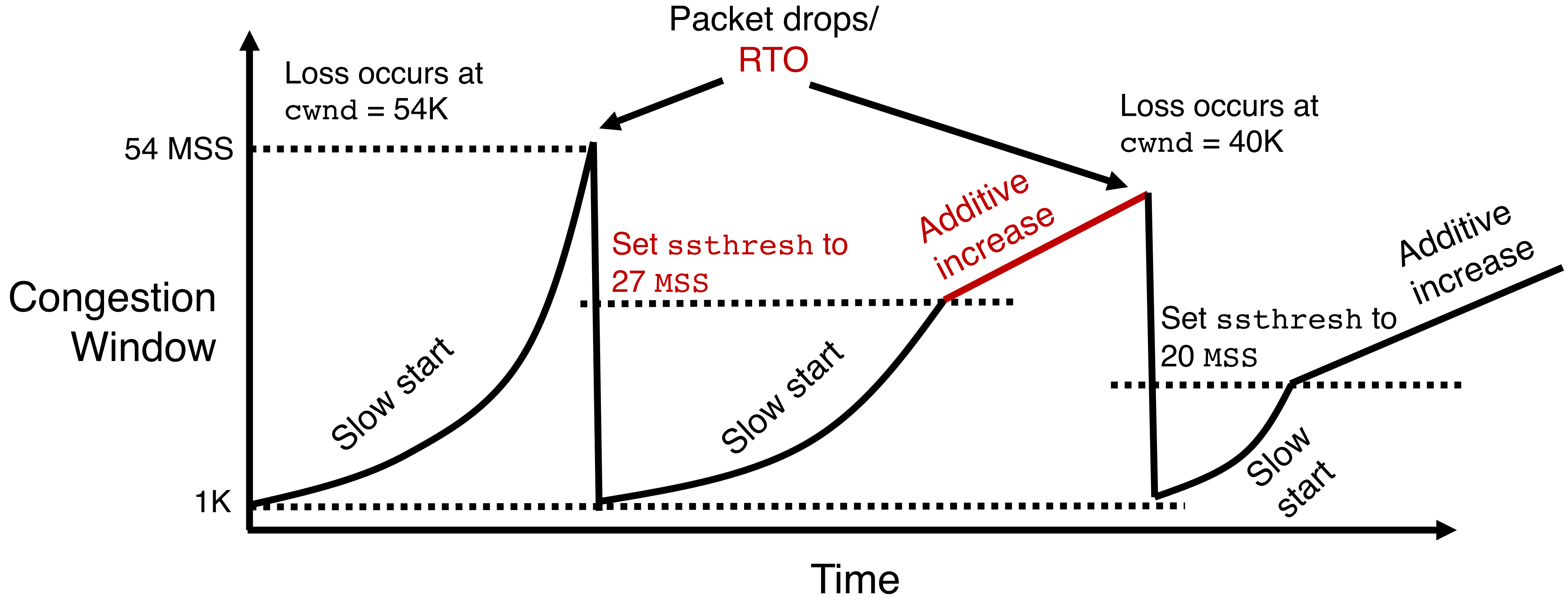
Default `ssthresh` = 64KB = 64 `MSS`

AI is slow.

Persistent connections

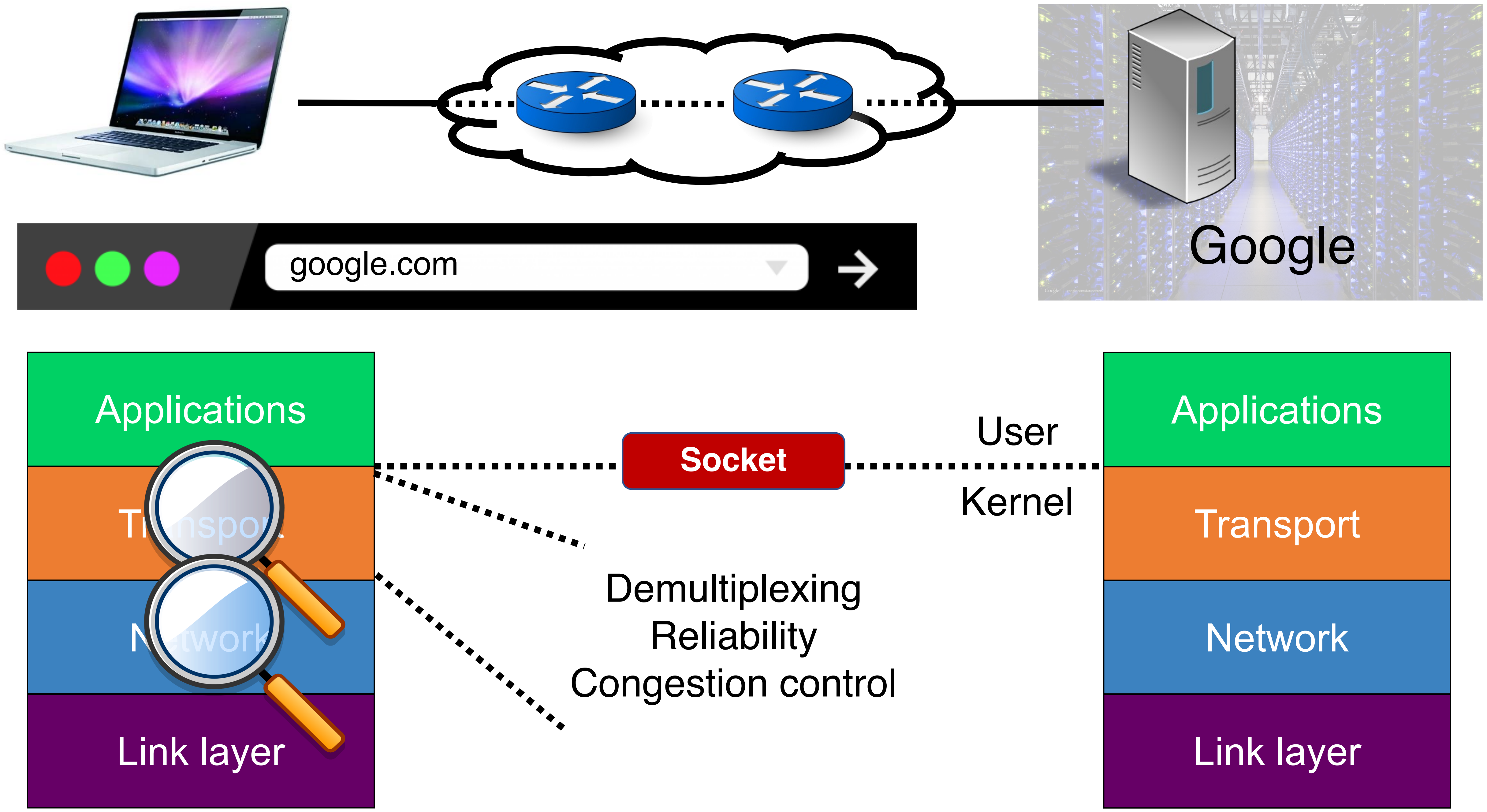
Large window sizes

Different laws to evolve congestion window



Routing


Norwegian Sea
Iceland
Sweden
Finland
Norway
Russia
Denmark
Ireland
Poland
Belarus
Ukraine
Austria
Romania
Kazakhstan
Sea of Okhotsk
Spain
Portugal
Turkey
Turkmenistan
Sea of Japan
China
Syria
Iraq
Iran
Afghanistan
Pakistan
Tunisia
Morocco
Algeria
Libya
Egypt
Saudi Arabia
Nepal
India
Western Sahara
Oman
Myanmar (Burma)
Mauritania
Mali
Niger
Chad
Sudan
Yemen
Gulf of Aden
Arabian Sea
Bay of Bengal
Philippines
Burkina Faso
Guinea
Nigeria
Ethiopia
South Sudan
Ghana
Gulf of Guinea
Laccadive Sea
Somalia
Kenya
Gabon
DR Congo
Indonesia
Tanzania
Banda Sea
Papua New Guinea
Arafura Sea


google.com
Google
Applications
Transport
Network
Link layer
Socket
User
Kernel
Demultiplexing
Reliability
Congestion control
Applications
Transport
Network
Link layer

Two key network-layer functions

- Forwarding: move packets from router's input to appropriate router output

- Routing: determine route taken by packets from source to destination
  - routing algorithms

- The network layer solves the routing problem.

Analogy: taking a road trip

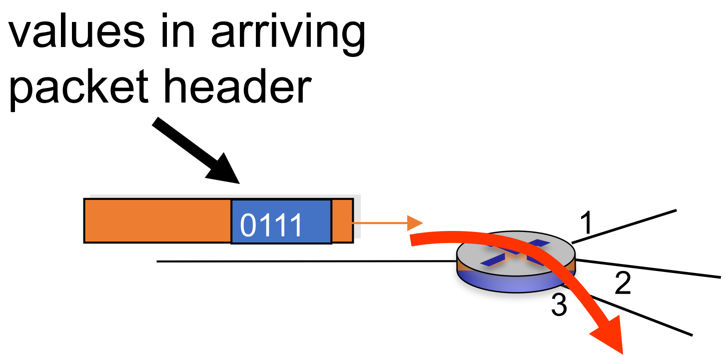
- Forwarding: process of getting through single exit

- Routing: process of planning trip from source to destination

Control/Data Planes

Data plane = Forwarding

- local, per-router function
- determines how datagram arriving on router input port is forwarded to router output port



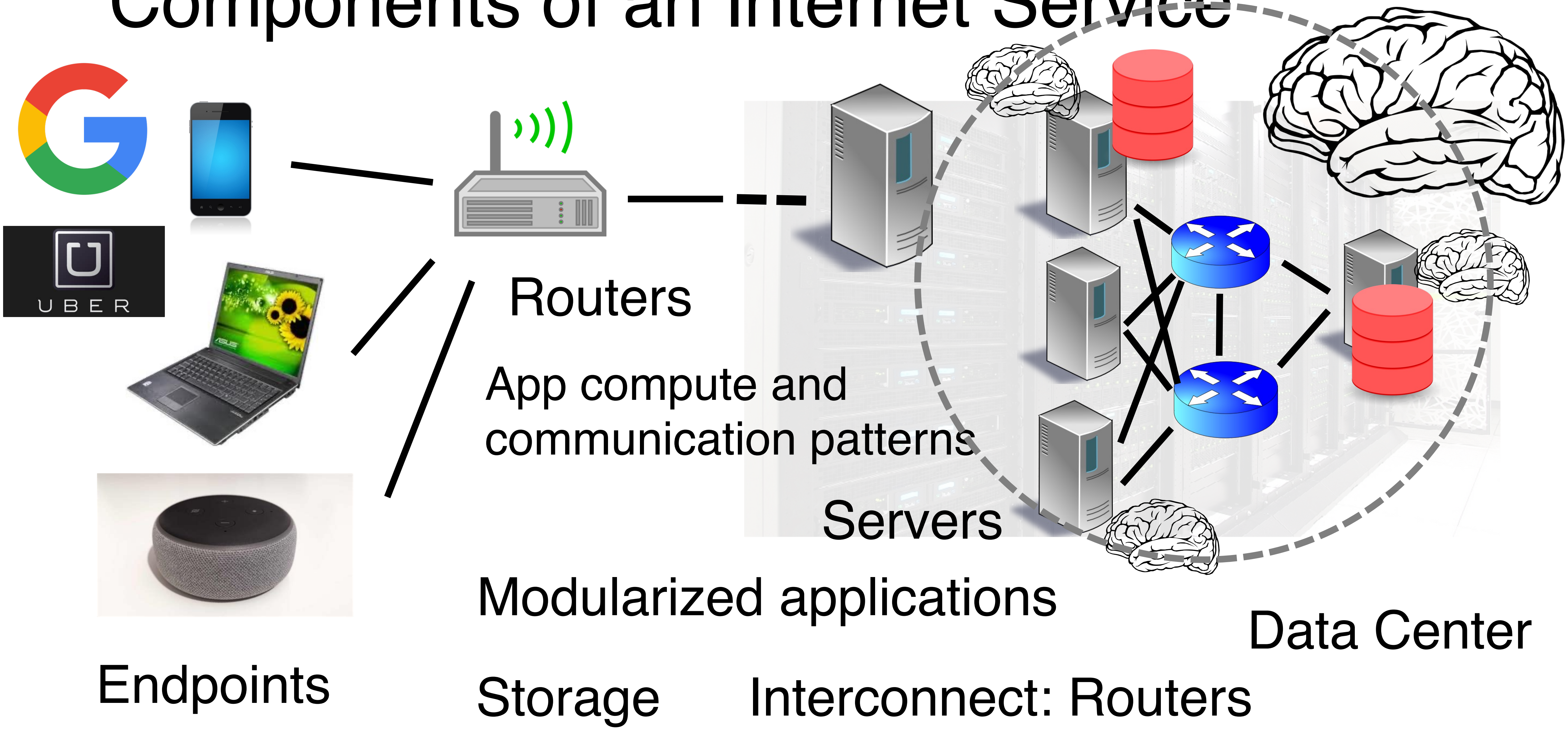
Control plane = Routing

- network-wide logic
- determines how datagram is routed along end-to-end path from source to destination endpoint
- two control-plane approaches:
  - Distributed routing algorithm running on each router
  - Centralized routing algorithm running on a (logically) centralized machine

Application architecture

Web servers

Components of an Internet Service



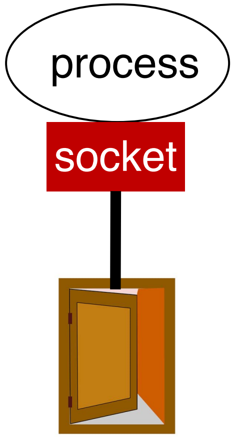
Web server

Often the first app point where a user request lands

Parse HTTP request

```
GET / HTTP/1.1
Host: example.com
```

(many other headers!)



IP_B + $port_B$

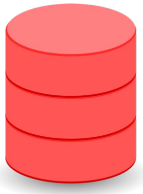
bind($IPaddr_B$, $port_B$)

listen()

accept()

recv()/send()/..

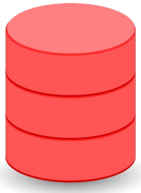
Find a file, run a script, …

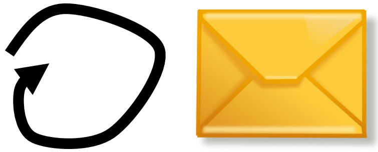


```
HTTP/1.1 200 OK
Content-Type: text/html
```

Send response header

Read file, `send()` data

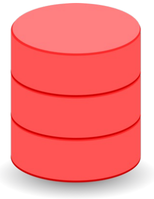




Overloaded with functionality

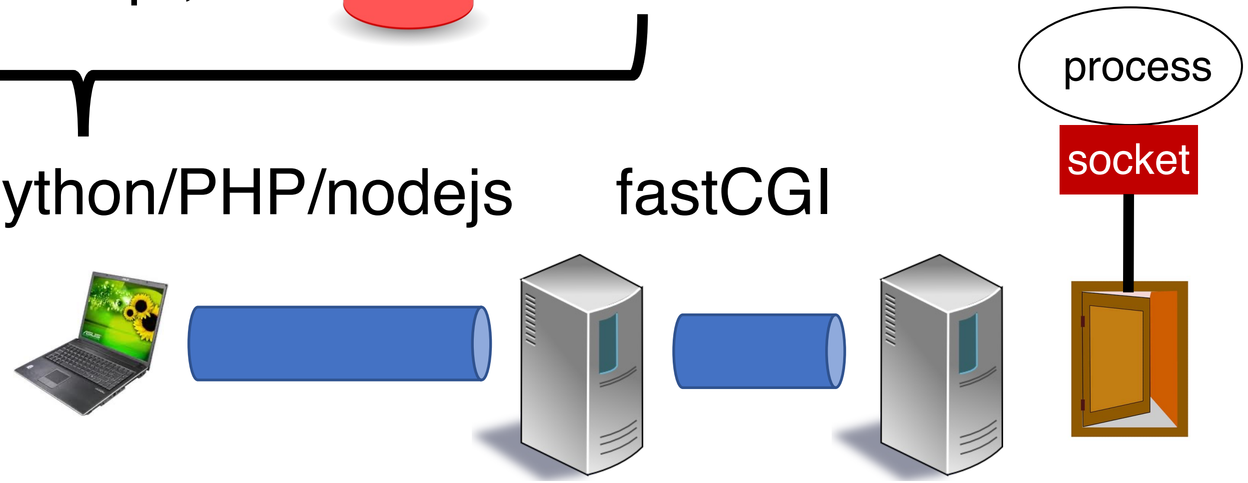
Often the first app point where a user request lands

Find a file, run a script, …



Scripting: Python/PHP/nodejs fastCGI

Reverse proxy



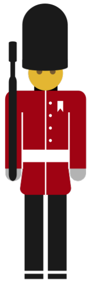
IP_B + $port_B$

```
bind(IPaddr_B, port_B)
listen()
accept()
recv()/send()/..
```

Caching

Compression

Access control



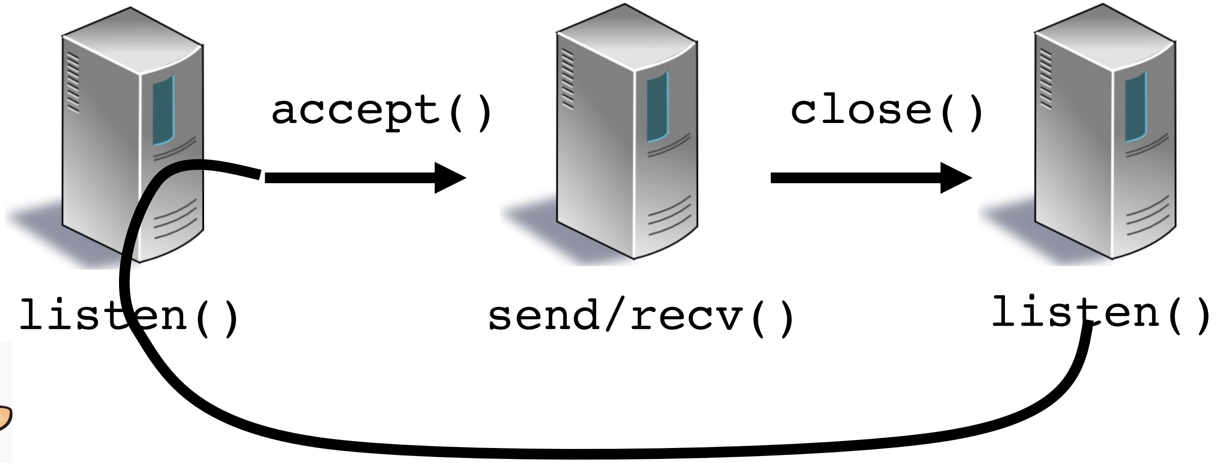
TLS

Media streaming

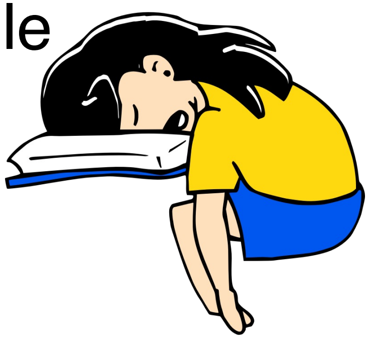
Image filtering

How does one design a web server?

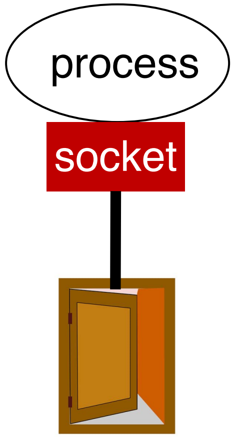
- Process connections one at a time?



Many other requests waiting in the meanwhile



Powerful server doing nothing most of the time



$IP_B + port_B$

`bind(IPaddr`$_B$`, port`$_B$`)`

`listen()`

`accept()`

`recv()/send()/..`

How does one design a web server?

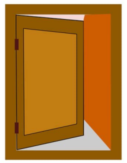
- Process other requests while waiting for one to finish

process

socket

IP_B + $port_B$

`bind(IPaddr_B, port_B)`



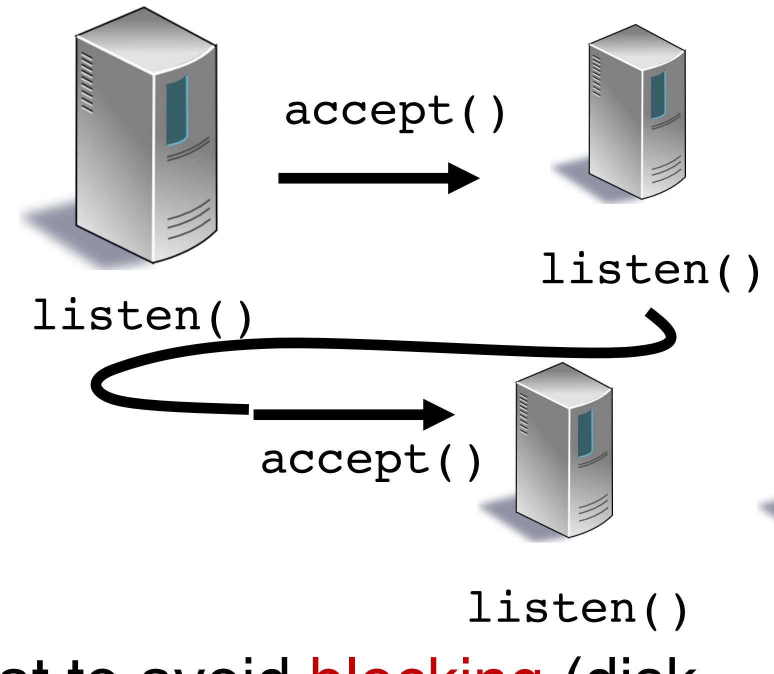
`listen()`

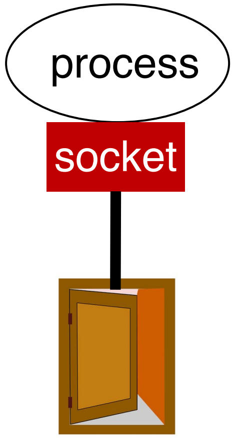
`accept()`

`recv()/send()/..`

Parallelism

- Process other requests while waiting for one to finish
- A first design: multiprocessing/threading





IP$_B$ + port$_B$

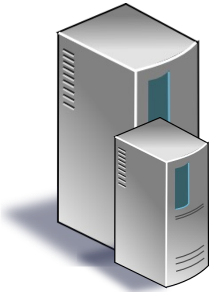
```
bind(IPaddr_B, port_B)

listen()

accept()
fork()
recv()/send()/..
```

Great to avoid **blocking** (disk I/O, fastCGI, …)

Overhead grows with # connections — more / longer lived

Concurrency

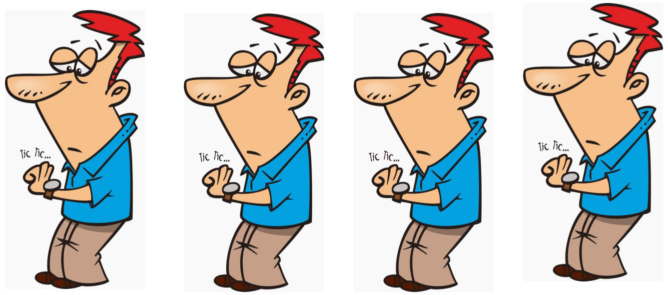
- Process other requests while waiting for one to finish
- A better design: event driven



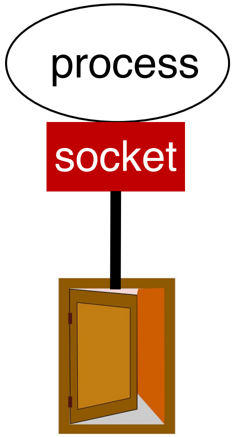
`accept()`

`recv()`

A queue of events



`epoll, select, kqueue,` etc.



IP_B + $port_B$

`bind(IPaddr_B, port_B)`

`listen()`

`accept()`

`recv()/send()/..`

Lightweight

Can block if any of the requests block

State of the art designs combine parallelism (multiprocess/thread) with concurrency (event-driven)