Service Delivery Architecture

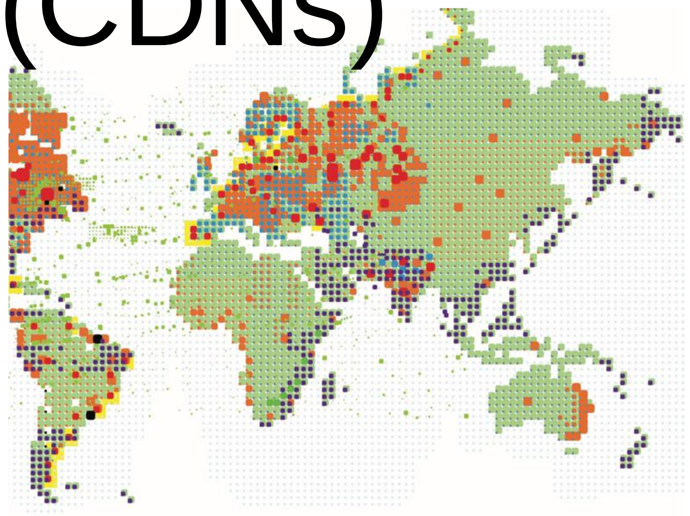
Lecture 3

Srinivas Narayana

http://www.cs.rutgers.edu/~sn624/553-S25

RUTGERS
UNIVERSITY | NEW BRUNSWICK

Content Distribution Networks (CDNs)



A global network of web caches

- Provisioned by ISPs and network operators
- Or content providers, like Netflix, Google, etc.

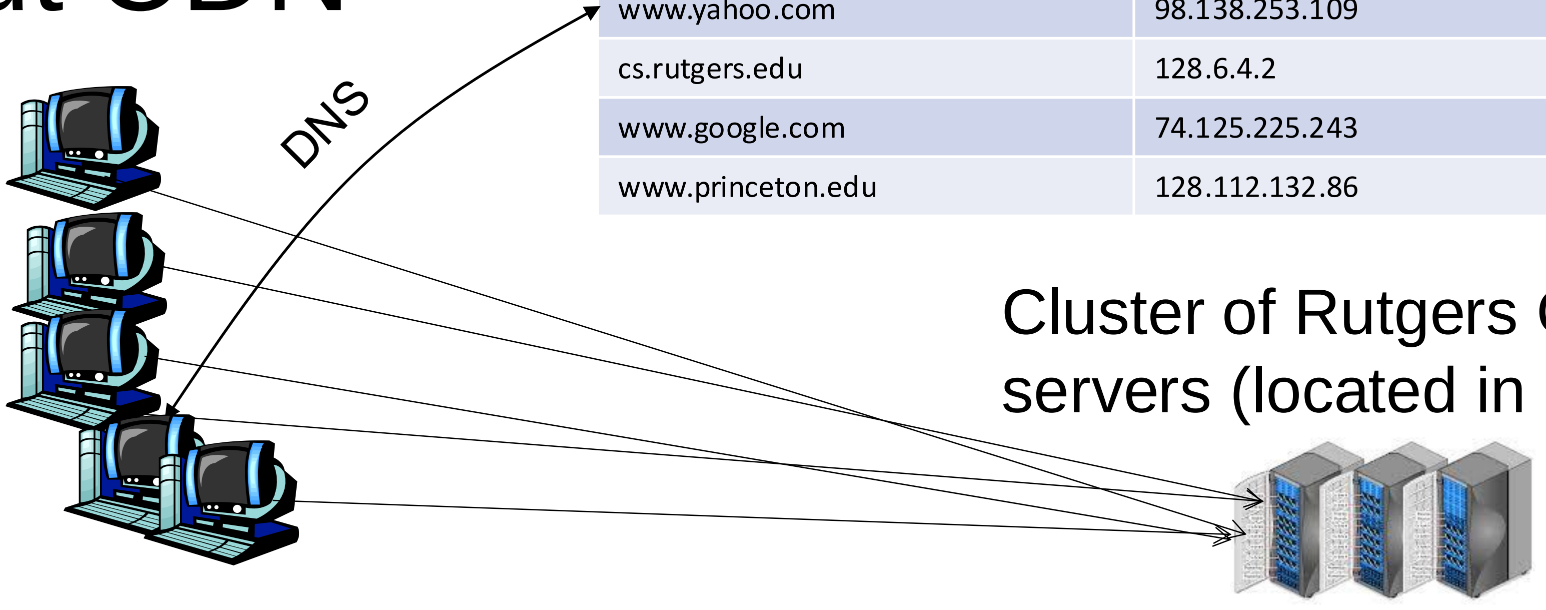
Uses

- Reduce traffic on a network's Internet connection, e.g., Rutgers
- Improve response time for users: CDN nodes are closer to users than origin servers (servers holding original content)
- Reduce bandwidth requirements on content provider
- Reduce $$ to maintain origin servers

Without CDN

DOMAIN NAME	IP ADDRESS
www.yahoo.com	98.138.253.109
cs.rutgers.edu	128.6.4.2
www.google.com	74.125.225.243
www.princeton.edu	128.112.132.86

Clients distributed all over the world



Cluster of Rutgers CS origin servers (located in NJ, USA)

128.6.4.2

- Problems:
- Huge bandwidth requirements for Rutgers
- Large propagation delays to reach users

Where the CDN comes in

- Distribute content of the origin server over geographically distributed CDN servers

- But how will users get to these CDN servers?

- Use DNS!
  - DNS provides an additional layer of indirection
  - Instead of returning IP address, return another DNS server (NS record)
  - The second DNS server (run by the CDN) returns IP address to client

- The CDN runs its own DNS servers (CDN name servers)
  - Custom logic to send users to the “closest” CDN web server

With CDN

DOMAIN NAME	IP ADDRESS
www.yahoo.com	98.138.253.109
cs.rutgers.edu	124.8.9.8 (NS record pointing to CDN name server)
www.google.com	74.125.225.243

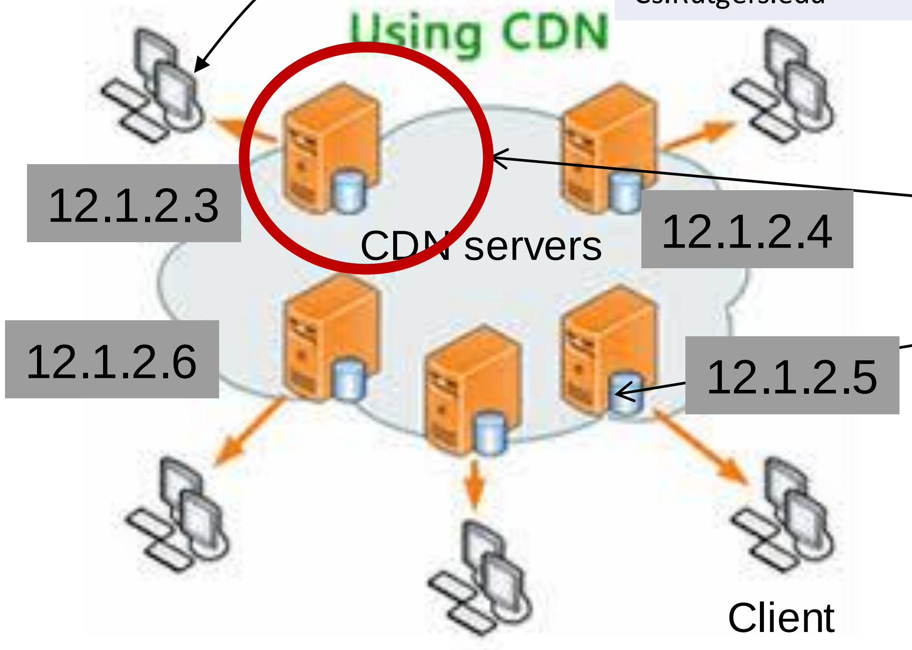
DNS reply

CDN Name Server (124.8.9.8)

DOMAIN NAME	IP ADDRESS
Cs.Rutgers.edu	12.1.2.3
Cs.Rutgers.edu	12.1.2.4
Cs.Rutgers.edu	12.1.2.5
Cs.Rutgers.edu	12.1.2.6

NS record delegates the choice of IP address to the CDN name server.

Custom logic to map ONE domain name to one of many IP addresses!



Popular CDNs:
CloudFlare
Akamai
Level3
…

128.6.4.2

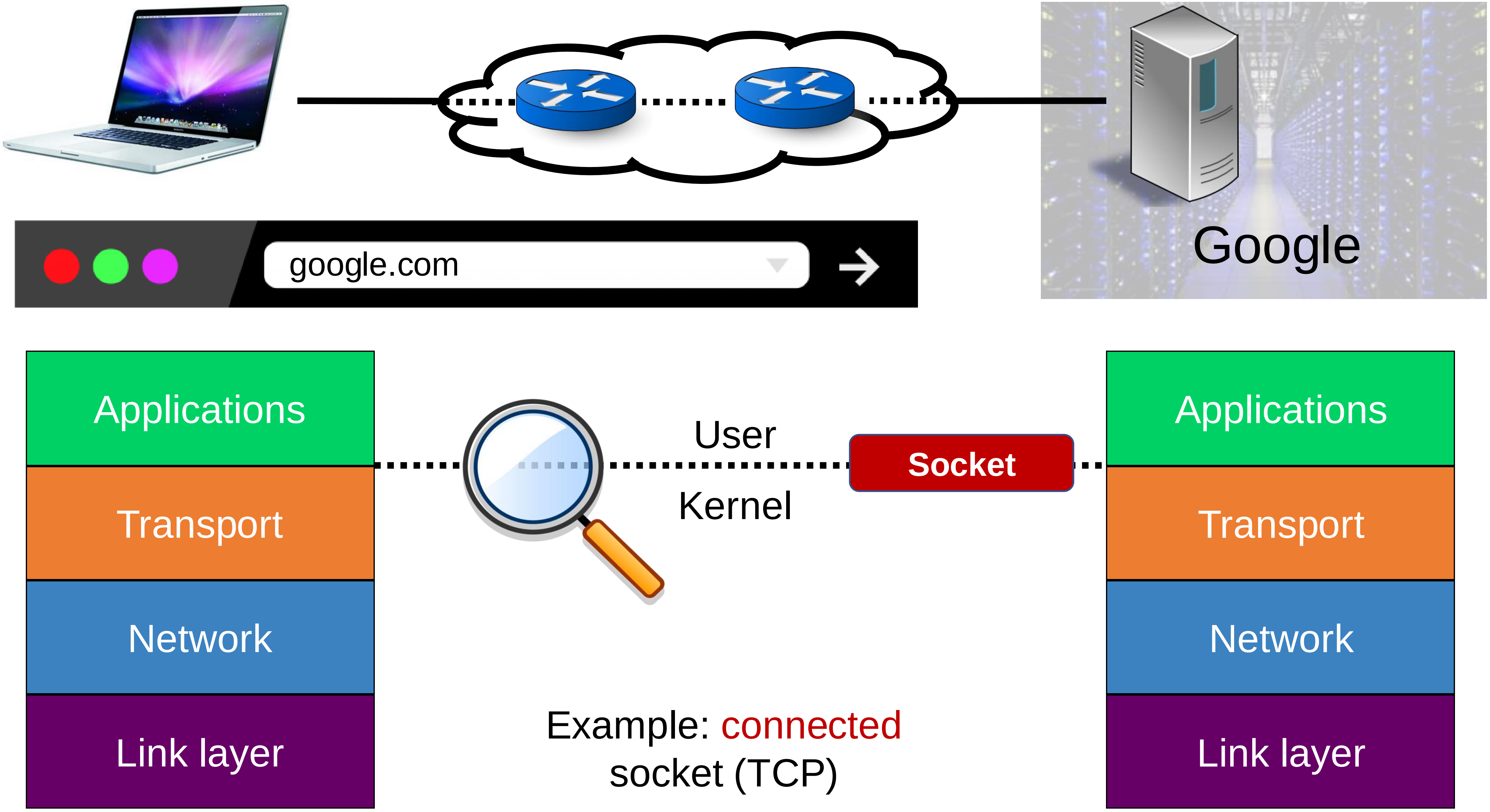
Origin server

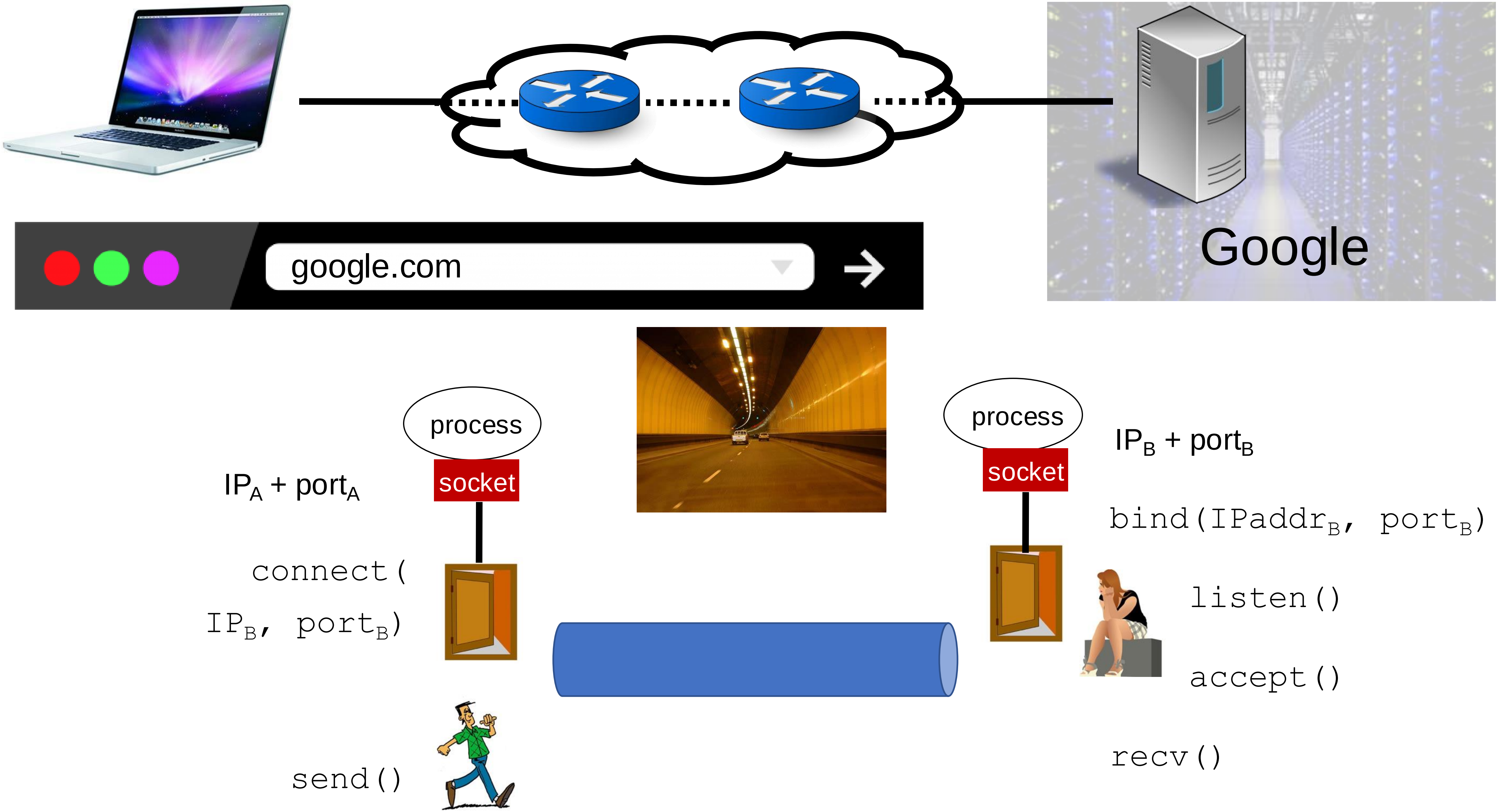
Most requests go to CDN servers (caches).
CDN servers may request object from origin
Few client requests go directly to origin server

Seeing a CDN in action

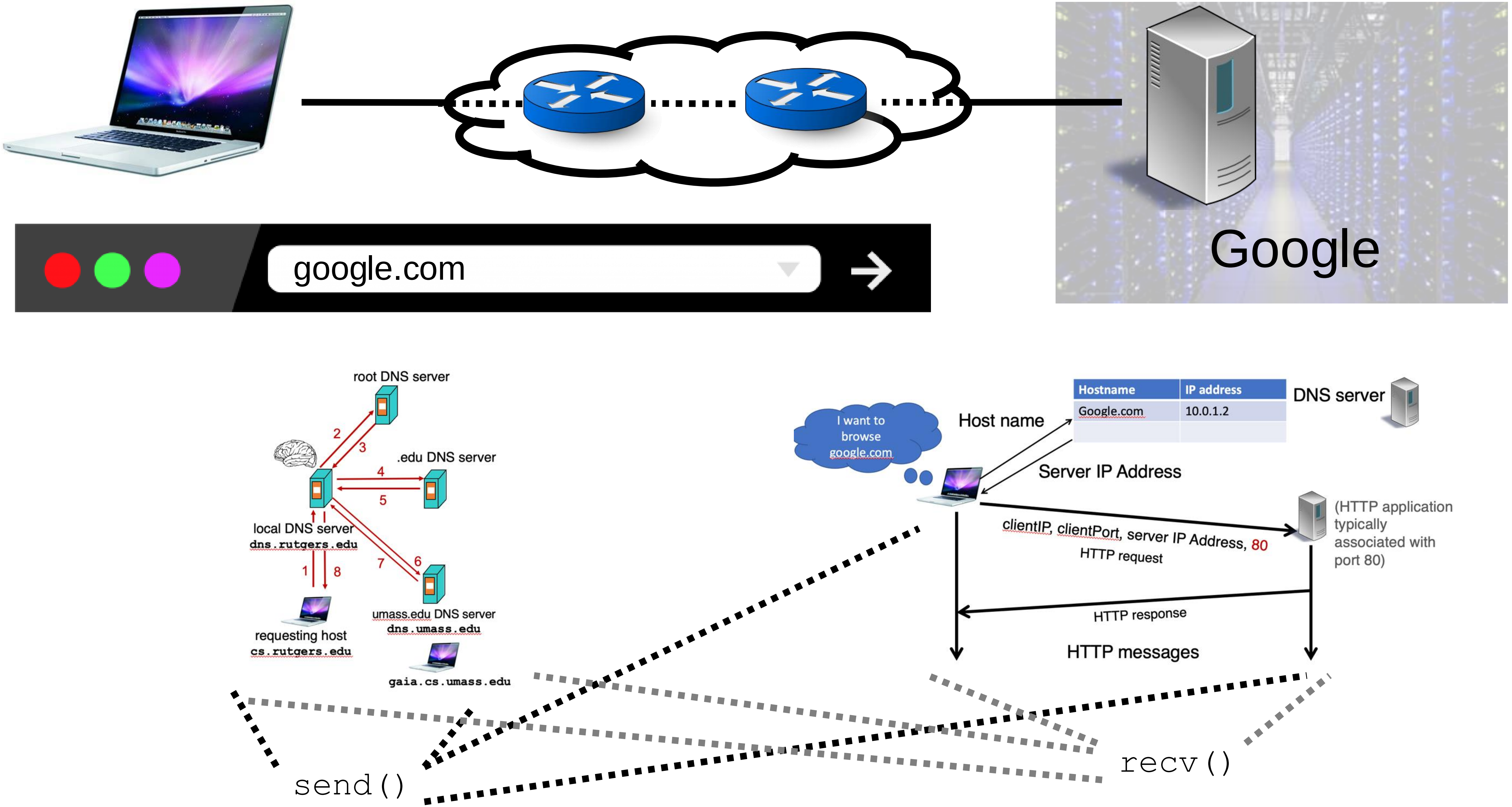
- `dig web.mit.edu` (or) `dig +trace web.mit.edu`
- `telnet web.mit.edu 80`

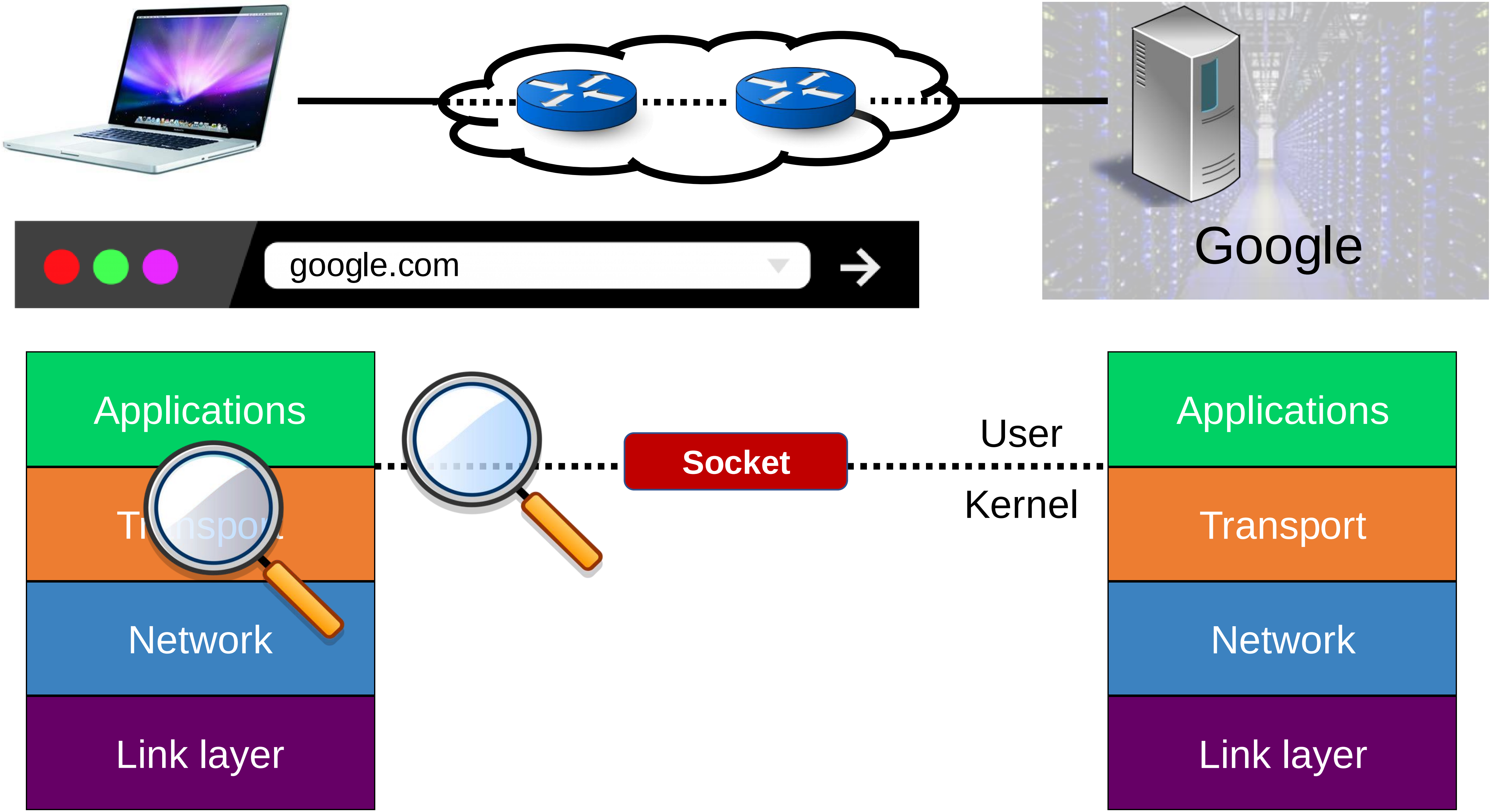
Application-OS interface


google.com
Google
Applications
Transport
Network
Link layer
User
Socket
Kernel
Applications
Transport
Network
Link layer
Example: connected
socket (TCP)



google.com
Google
process
socket
IP_A + port_A
connect(
IP_B, port_B)
send()
process
socket
IP_B + port_B
bind(IPaddr_B, port_B)
listen()
accept()
recv()

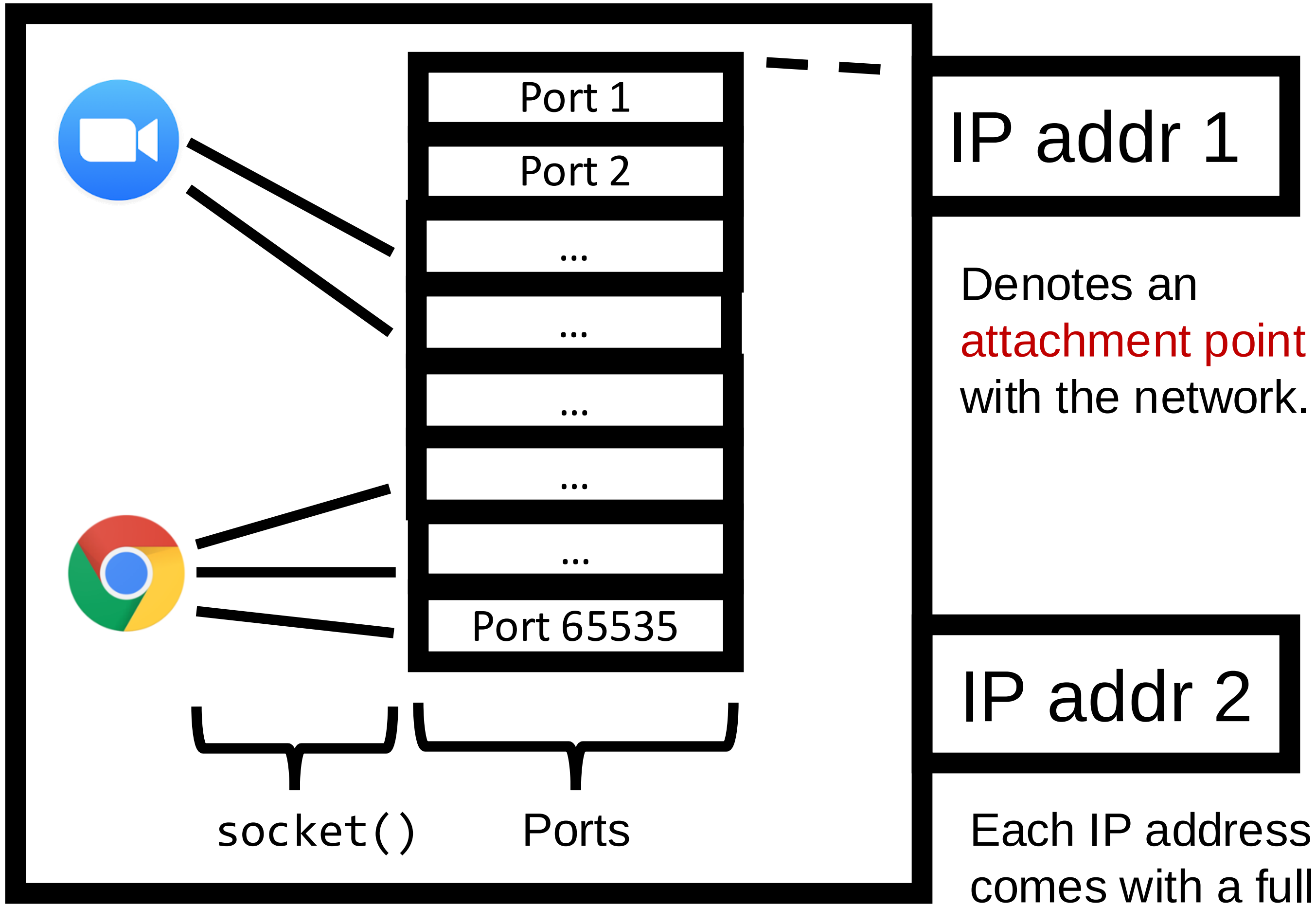

google.com
Google
root DNS server
.edu DNS server
local DNS server
dns.rutgers.edu
requesting host
cs.rutgers.edu
umass.edu DNS server
dns.umass.edu
gaia.cs.umass.edu
1
2
3
4
5
6
7
8
I want to browse google.com
Host name
Hostname
IP address
Google.com
10.0.1.2
DNS server
Server IP Address
clientIP, clientPort, server IP Address, 80
HTTP request
(HTTP application typically associated with port 80)
HTTP response
HTTP messages
send()
recv()



google.com
Google
Applications
Network
Link layer
Socket
User
Kernel
Applications
Transport
Network
Link layer

Transport

(1) (De)multiplexing

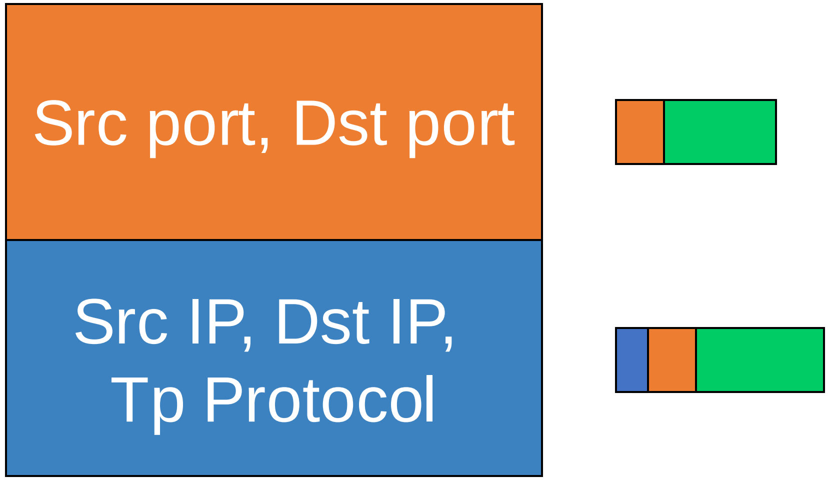


Machine

Denotes an attachment point with the network.

Each IP address comes with a full copy of its own ports.

Connection lookup: The operating system does a lookup using these data to determine the right socket and app.



UDP or TCP listening:
(dst IP, dst port, TCP)

TCP established:
(dst IP, dst port, src IP, src port, TCP)

TCP sockets of different types

Listening (bound but unconnected)

```
# On server side
ls = socket(AF_INET, SOCK_STREAM)
ls.bind(serv_ip, serv_port)
ls.listen() # no accept() yet
```

(dst IP, dst port)

➔

Socket (ss)

Enables new connections to be demultiplexed correctly

Connected (Established)

```
# On server side
cs, addr = ls.accept()
```

`accept()` creates a new socket with the 4-tuple (established) mapping

```
# On client side
connect(serv_ip, serv_port)
```

(src IP, dst IP, src port, dst port)

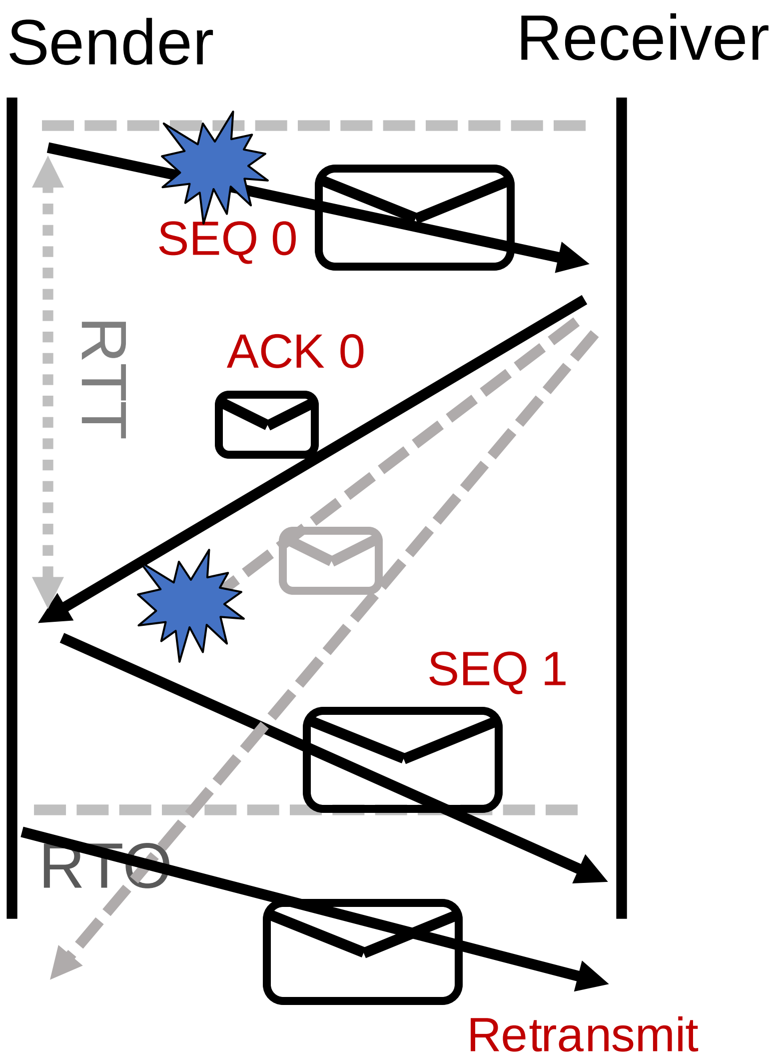
➔

Socket (cs NOT ls)

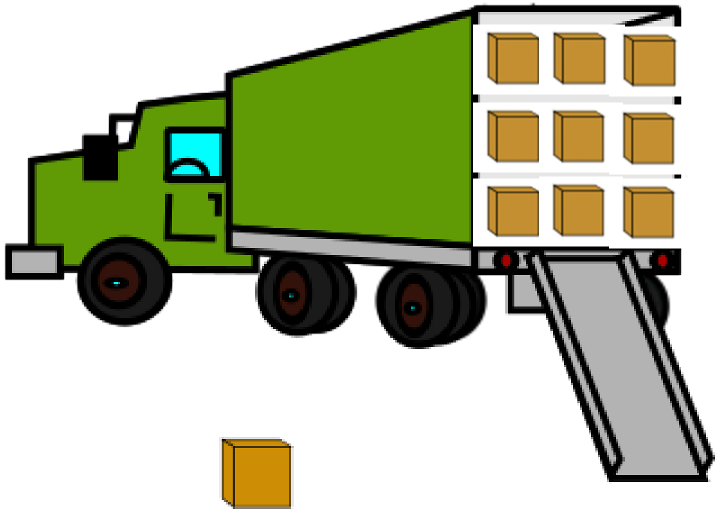
Enables established connections to be demultiplexed correctly

(2) Reliability: Stop and Wait. 3 Ideas

- ACKs: Sender sends a single packet, then waits for an ACK to know the packet was successfully received. Then the sender transmits the next packet.
- RTO: If ACK is not received until a timeout, sender retransmits the packet
- Seq: Disambiguate duplicate vs. fresh packets using sequence numbers that change on "adjacent" packets



Sending one packet per RTT makes the data transfer rate limited by the time between the endpoints, rather than the bandwidth.



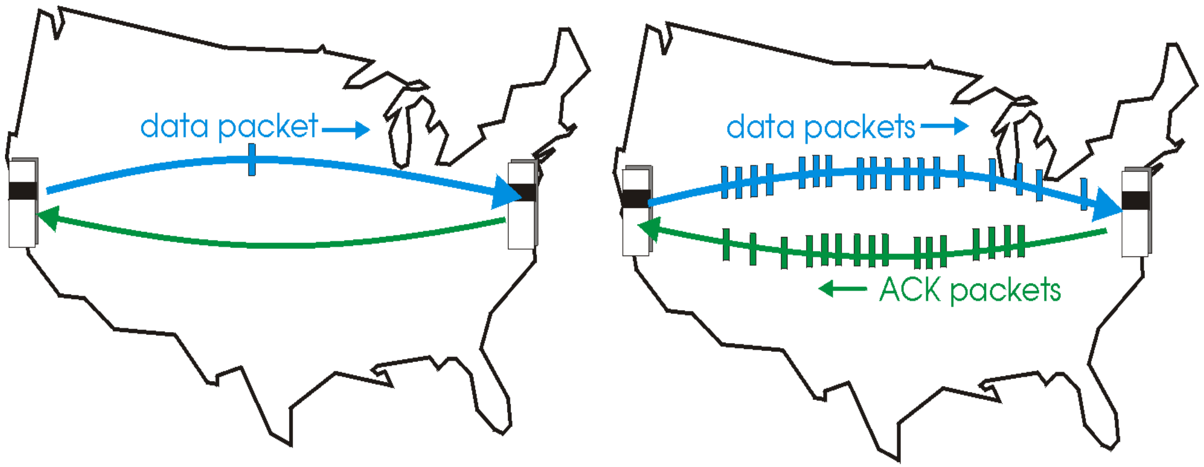
Ensure you got the (one) box safely; make N trips

Ensure you get N boxes safely; make just 1 trip!

Keep many packets in flight

Pipelined reliability

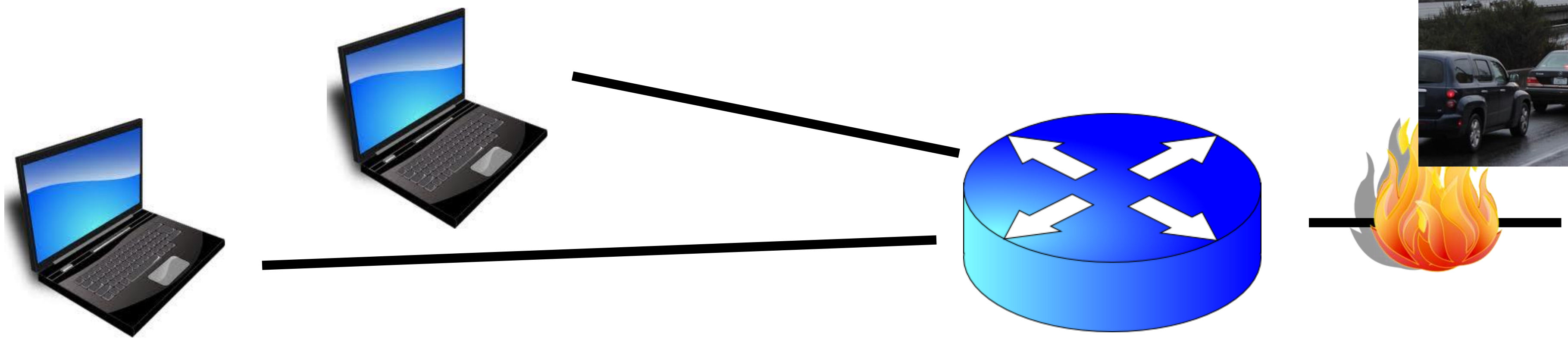
- Data in flight: data that has been sent, but sender hasn't yet received ACKs from the receiver
  - Note: can refer to packets in flight or bytes in flight
- New packets sent at the same time as older ones still in flight
- New packets sent at the same time as ACKs are returning
- More data moving in same time!
- Improves throughput
  - Rate of data transfer



(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

(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.

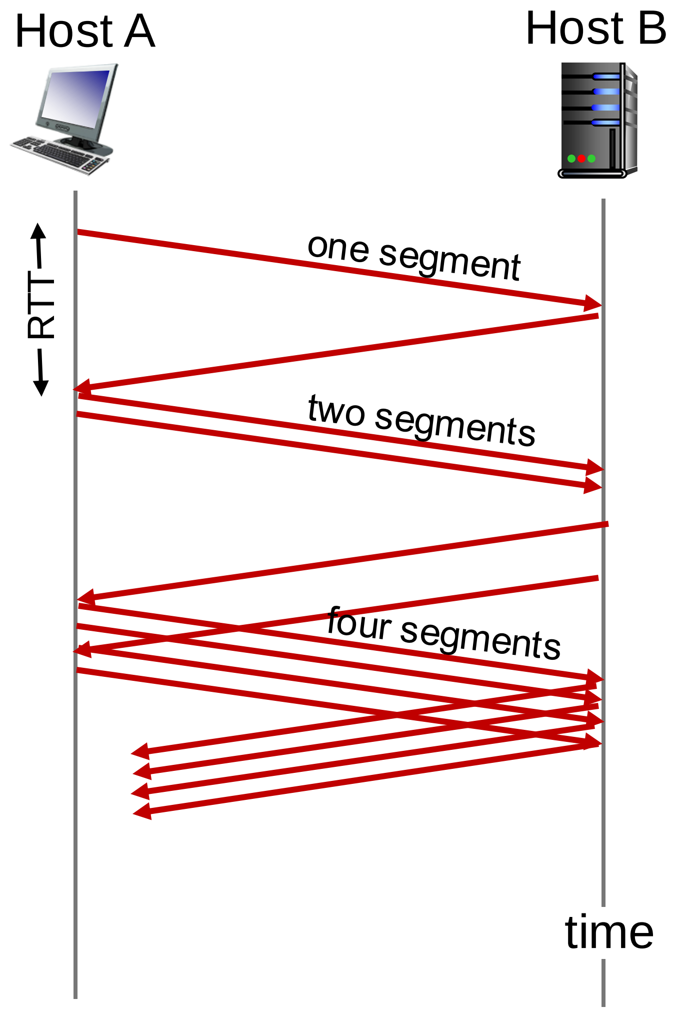
- If sender sends less than the bottleneck link rate:
  - all packets get through; successful ACKs

- Congestion window (`cwnd`): amount of data in flight

Quickly finding a rate: TCP slow start

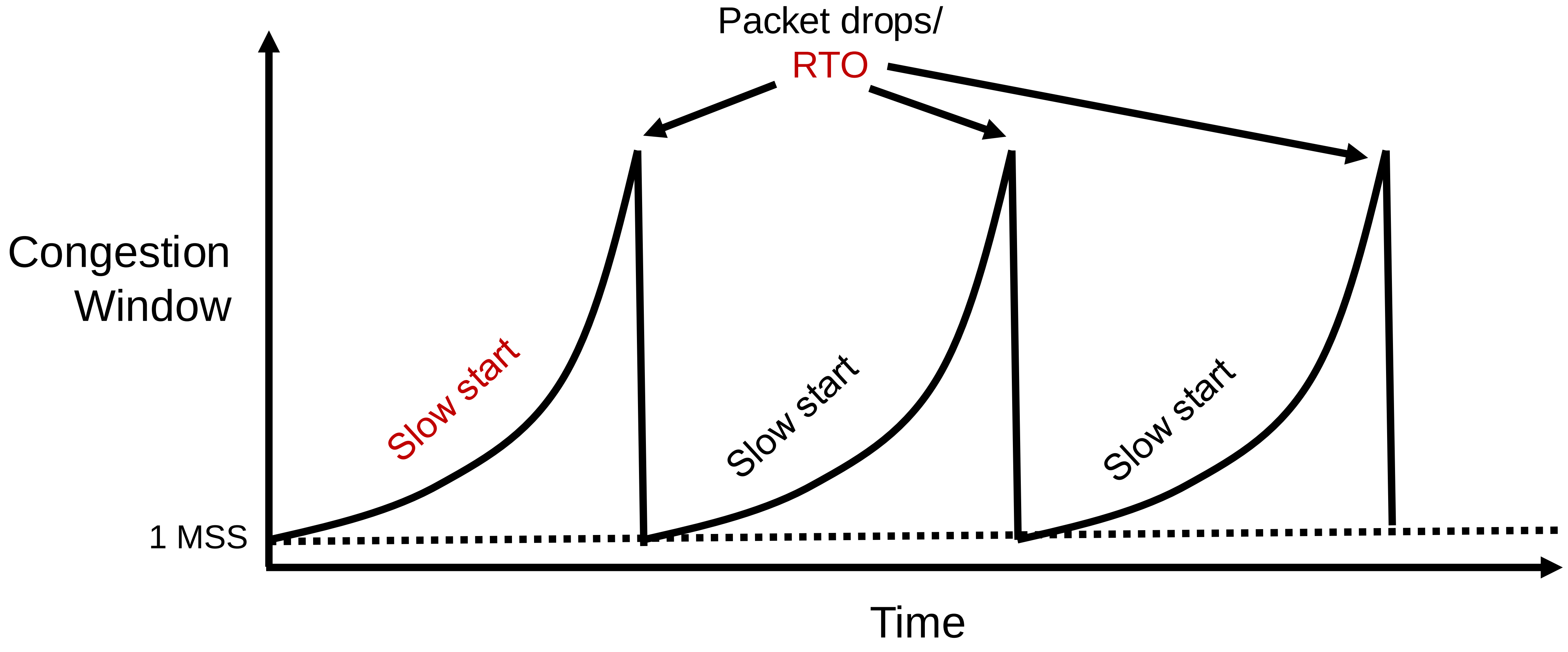
- Initially `cwnd` = 1 MSS
  - MSS is "maximum segment size"

- 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

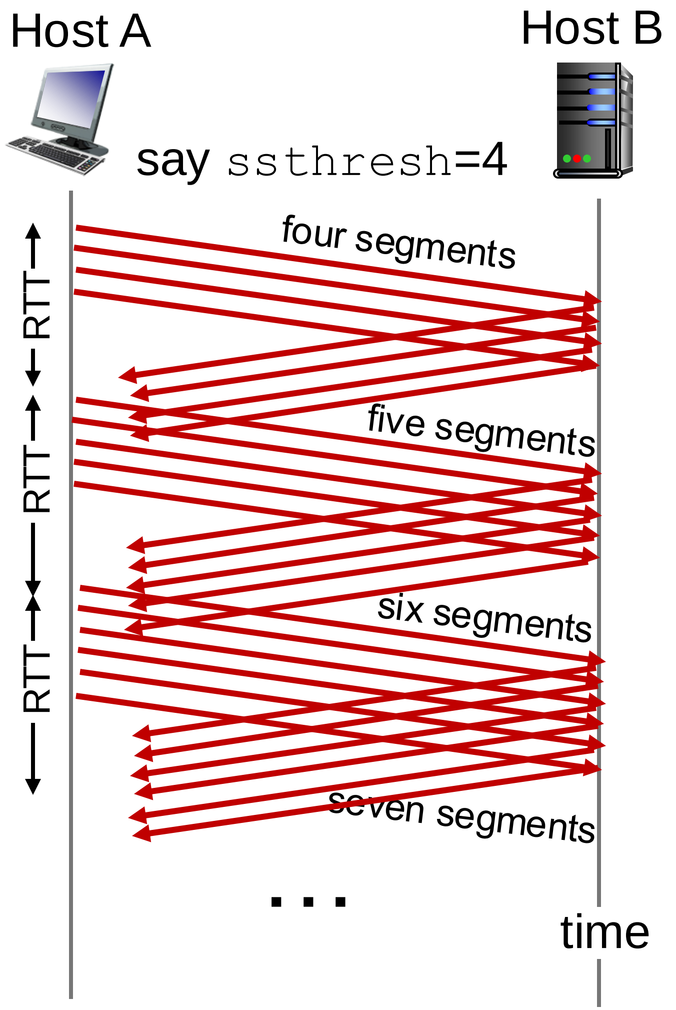


Slow start has problems

- 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

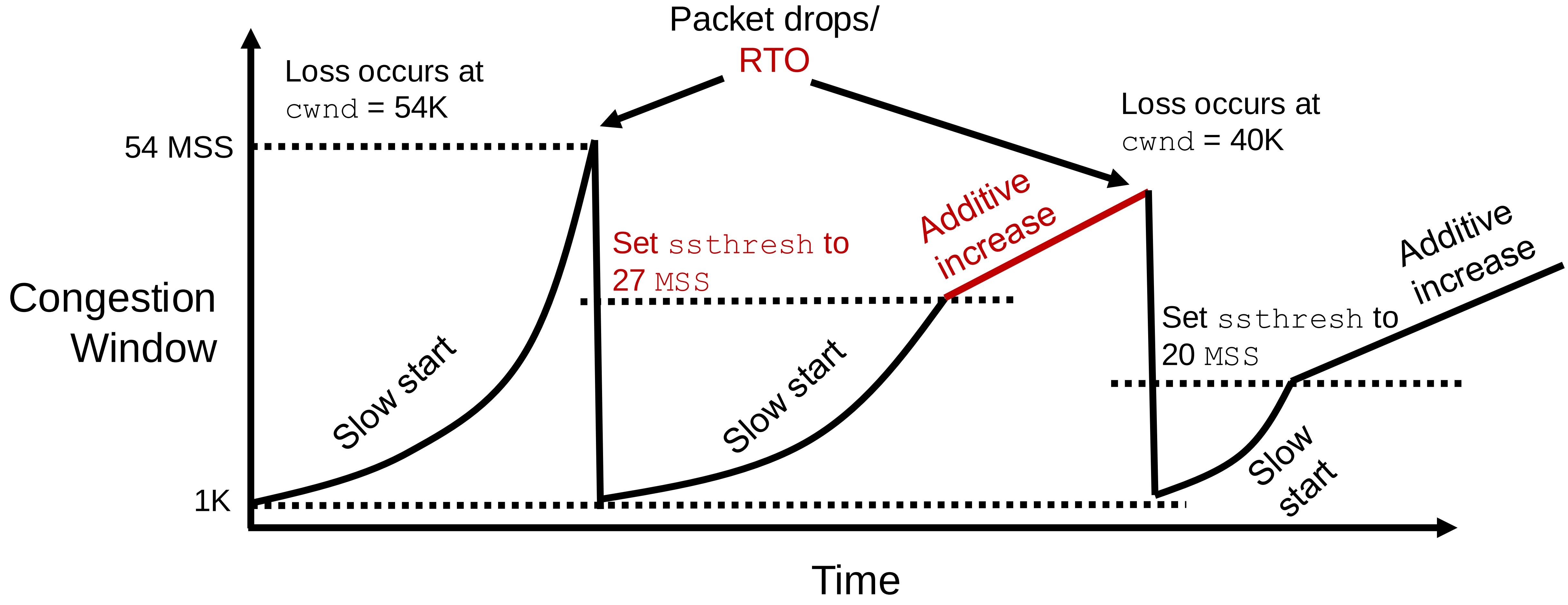
Default `ssthresh` = 64KB = 64 `MSS`

AI is slow.

Persistent connections

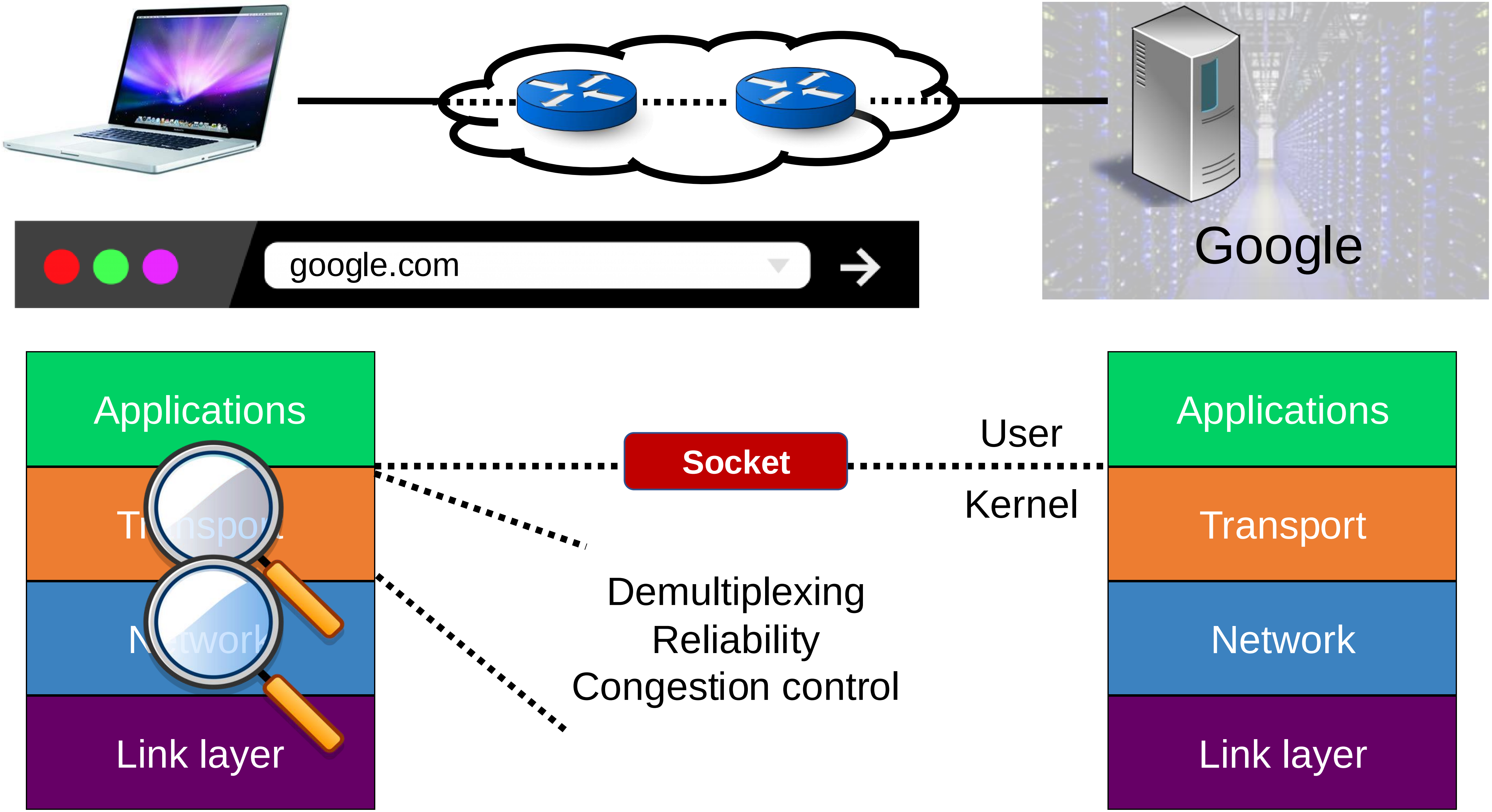
Large window sizes

Different laws to evolve congestion window



Routing


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


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

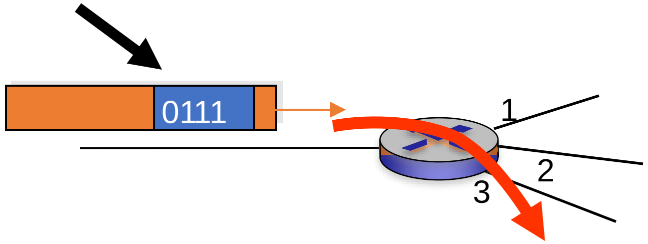
- Routing: process of planning trip from source to destination

Control/Data Planes

Data plane = Forwarding

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

values in arriving packet header



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