CS162
Operating Systems and Systems Programming
Lecture 16

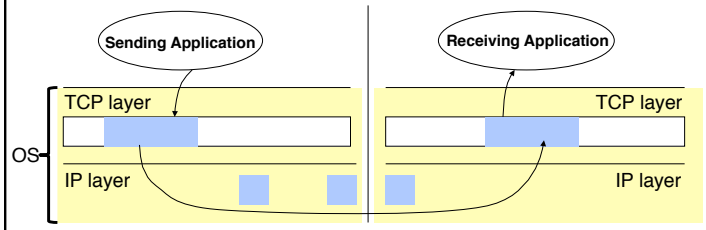
Flow Control, DNS

March 28, 2011
Ion Stoica
http://inst.eecs.berkeley.edu/~cs162

TCP Flow Control

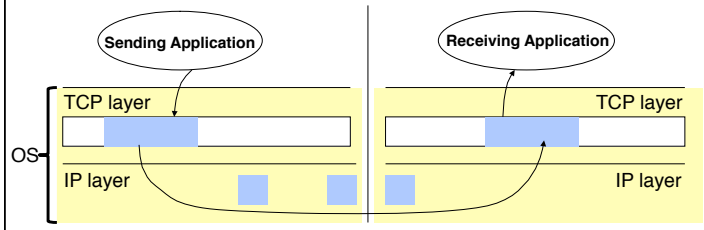
- TCP: stream oriented protocol
  - Sender sends a stream of bytes, not packets (e.g., no need to tell TCP how much you send)
  - Receiver reads a stream of bytes
- TCP flow control:
  - Sliding window protocol at byte (not packet) level
    - » Go-back-N: TCP Tahoe, Reno, New Reno
    - » Selective acknowledgement: TCP Sack
  - Receiver tells sender how many more bytes it can receive without overflowing its buffer (i.e., AdvertisedWindow)
  - The ack(nowledgement) contains sequence number N of next byte the receiver expects, i.e., receiver has received all bytes in sequence up to and including N-1

TCP Flow Control



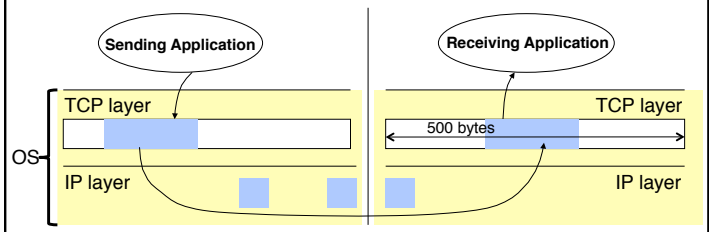
- TCP/IP implemented by OS (Kernel)
  - TCP and application run in different processes
  - Cannot do context switching on sending/receiving every packet
    - » At 1Gbps, it takes 12 usec to send an 1500 bytes, and 0.8usec to send an 100 byte packet
- Need buffers to match
  - Sending app with sending TCP
  - Receiving TCP with receiving app

TCP Flow Control



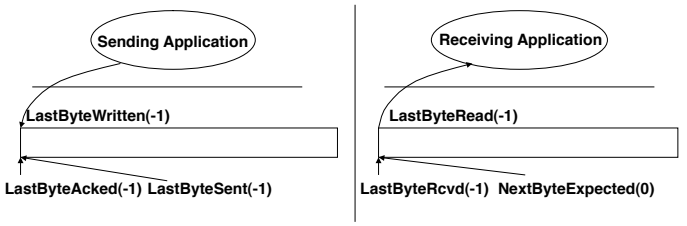
- Three pairs of producer-consumer's
  - sending app → sending TCP
  - sending TCP → receiving TCP
  - receiving TCP → receiving app
- How is mutual exclusion implemented?

TCP Flow Control



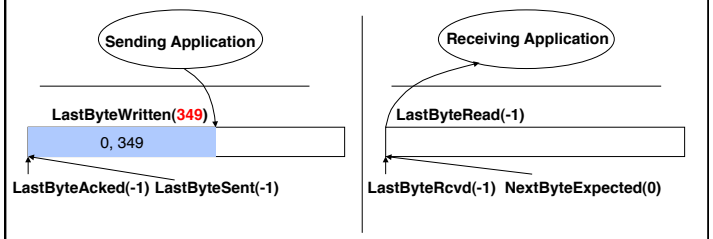
- Example assumptions:
  - Maximum IP packet size = 100 bytes
  - Size of the receiving buffer (MaxRcvBuf) = 500bytes
- Use circular buffers, i.e., N's byte is stored at (N mod MaxRcvBuf) in the buffer
- Recall, ack indicates the next expected byte in-sequence, not the last received byte

TCP Flow Control



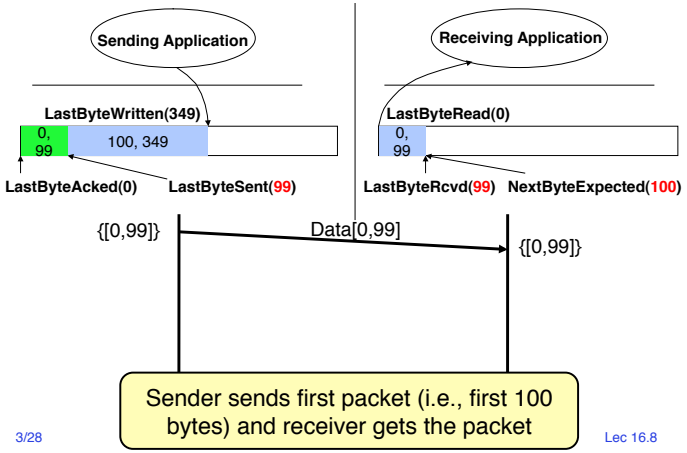
- LastByteWritten: last byte written by the sending app
- LastByteSent: last byte sent by the sender
- LastByteAcked: last byte acked at the sender
- LastByteRcvd: last byte received at receiver
- NextByteExpected: last in-sequence byte expected by receiver
- LastByteRead: last byte read by the receiving app

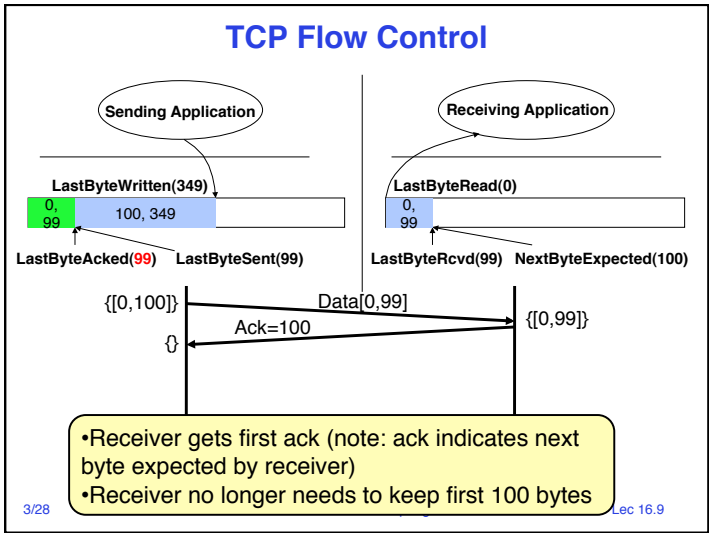
TCP Flow Control

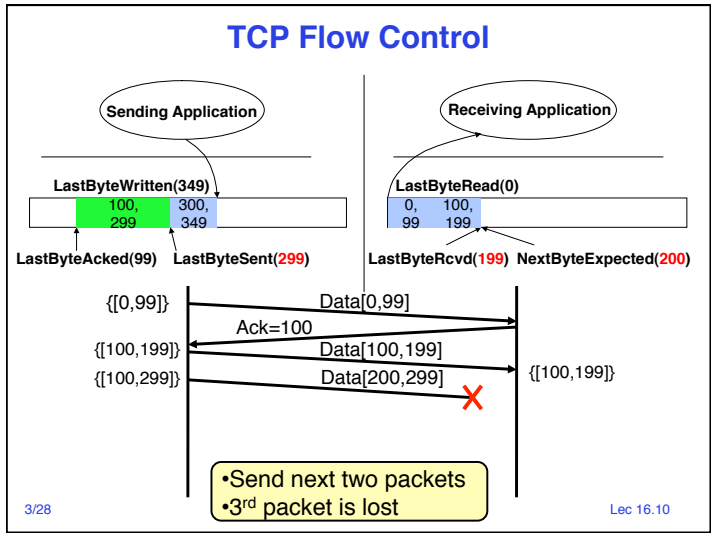


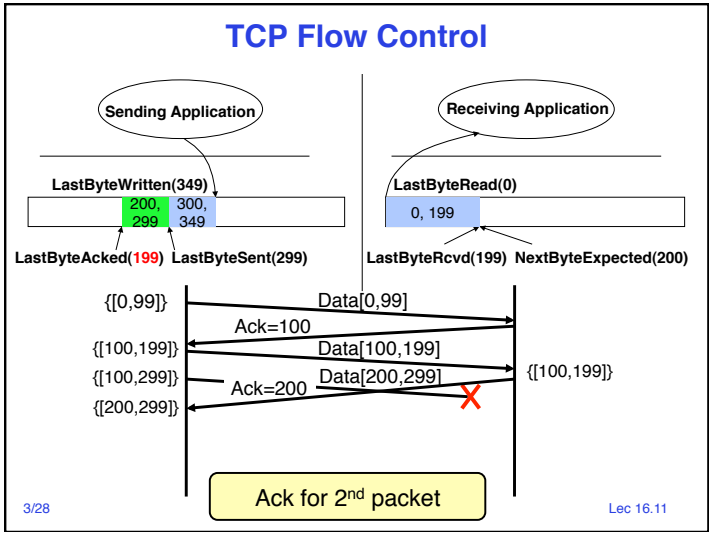
- Sending app sends 350 bytes
- Recall: we assume IP only accepts packets no larger than 100 bytes

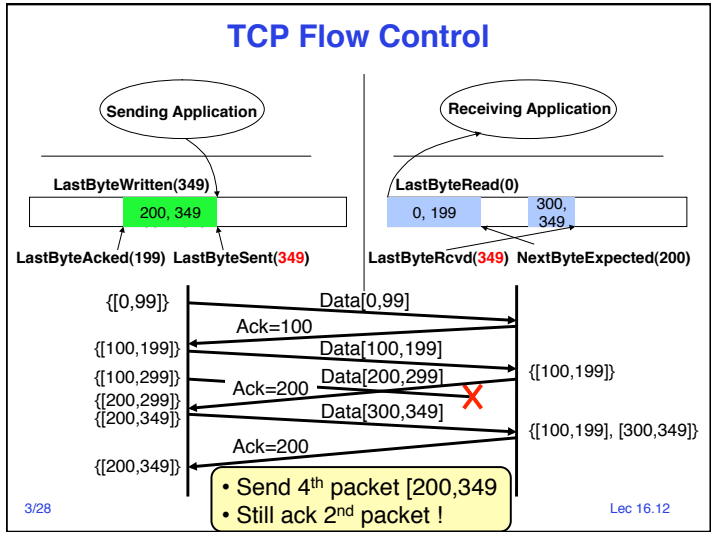
TCP Flow Control



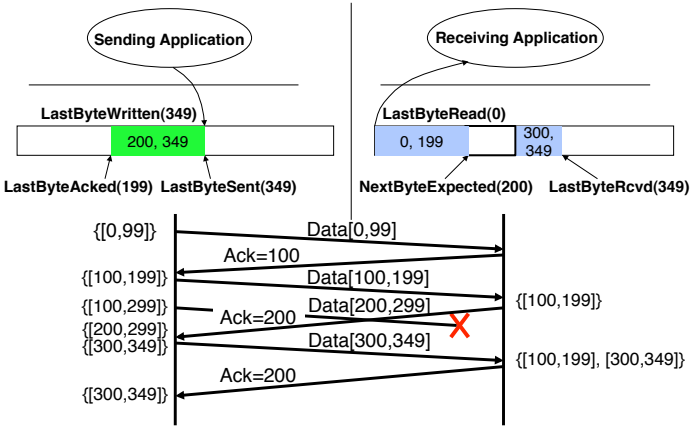

TCP Flow Control
Sending Application
Receiving Application
LastByteWritten(349)
0, 99
100, 349
LastByteRead(0)
0, 99
LastByteAcked(99)
LastByteSent(99)
LastByteRcvd(99)
NextByteExpected(100)
{[0,100]}
Data[0,99]
{[0,99]}
Ack=100
{}
•Receiver gets first ack (note: ack indicates next byte expected by receiver)
•Receiver no longer needs to keep first 100 bytes
3/28
Lec 16.9


TCP Flow Control
Sending Application
Receiving Application
LastByteWritten(349)
100, 299
300, 349
LastByteRead(0)
0, 99
100, 199
LastByteAcked(99)
LastByteSent(299)
LastByteRcvd(199)
NextByteExpected(200)
{[0,99]}
Data[0,99]
Ack=100
{[100,199]}
Data[100,199]
{[100,299]}
Data[200,299]
{[100,199]}
•Send next two packets
•3rd packet is lost
3/28
Lec 16.10

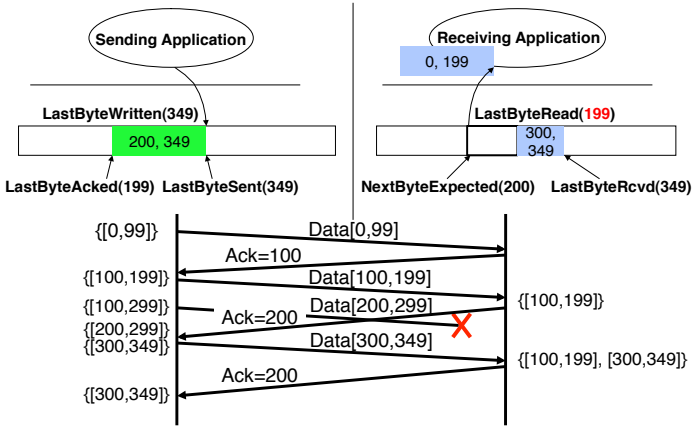

TCP Flow Control
Sending Application
Receiving Application
LastByteWritten(349)
200, 299
300, 349
LastByteRead(0)
0, 199
LastByteAcked(199)
LastByteSent(299)
LastByteRcvd(199)
NextByteExpected(200)
{[0,99]}
Data[0,99]
Ack=100
{[100,199]}
Data[100,199]
{[100,299]}
Ack=200
Data[200,299]
{[100,199]}
{[200,299]}
Ack for 2nd packet
3/28
Lec 16.11


TCP Flow Control
Sending Application
Receiving Application
LastByteWritten(349)
200, 349
LastByteRead(0)
0, 199
300, 349
LastByteAcked(199)
LastByteSent(349)
LastByteRcvd(349)
NextByteExpected(200)
{[0,99]}
Data[0,99]
Ack=100
{[100,199]}
Data[100,199]
{[100,299]}
Data[200,299]
{[100,199]}
Ack=200
{[200,299]}
{[200,349]}
Data[300,349]
{[100,199], [300,349]}
Ack=200
{[200,349]}
• Send 4th packet [200,349
• Still ack 2nd packet !
3/28
Lec 16.12

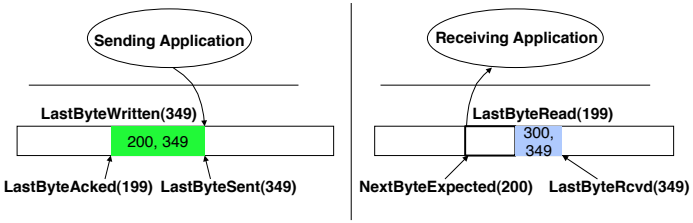
TCP Flow Control



TCP Flow Control



TCP Flow Control



- AdvertisedWindow: number of bytes the receiver can receive

  AdvertisedWindow = MaxRcvBuffer – (LastByteRcvd – LastByteRead)

- Sender window: number of bytes the sender can send

  Sender window = AdvertisedWindow – (LastByteSent – LastByteAcked)
  MaxSendBuffer >= LastByteWritten - LastByteAcked

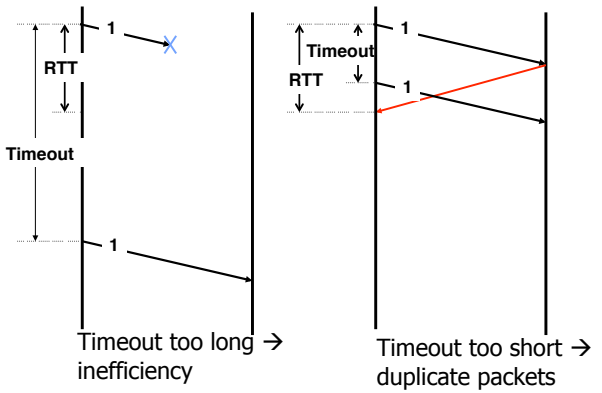
What if Receiving App Stops Receiving Data?

- LastByteRead stops advancing → receiving buffer eventually fills with undelivered data, i.e.,
  LastByteRcvd = MaxRcvBuffer + LastByteRead →
  AdvertisedWindow = 0
- Sending TCP stops sending data (as AdvertisedWindow = 0) → LastByteSent and LastByteAcked stop advancing
- Sending TCP buffer fills in when
  LastByteWritten = LastByteAcked + MaxSendBuffer
- Sending app stops sending data when sending TCP buffer fills in

Retransmission Timeout

- If haven't received ack by timeout, retransmit packet after last acked packet
- How to set timeout?

Timing Illustration



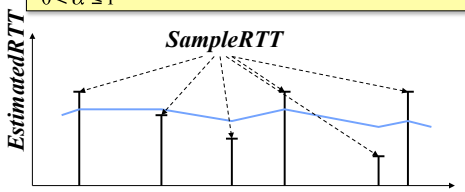
Retransmission Timeout (cont'd)

- If haven't received ack by timeout, retransmit packet after last acked packet
- How to set timeout?
  - Too long: connection has low throughput
  - Too short: retransmit packet that was just delayed
    - » Packet was probably delayed because of congestion
    - » Sending another packet too soon just makes congestion worse
- Solution: make timeout proportional to RTT

RTT Estimation

- Use exponential averaging:

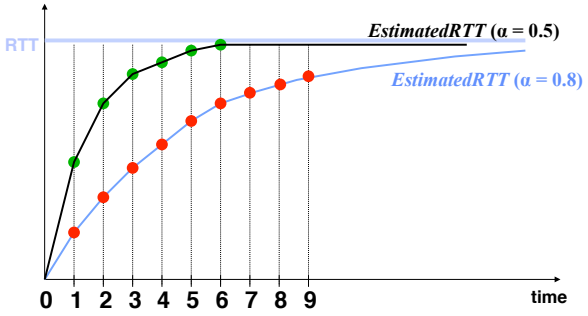
$$SampleRTT = AckRcvdTime - SendPacketTime$$
$$EstimatedRTT = \alpha \times EstimatedRTT + (1-\alpha) \times SampleRTT$$
$$TimeOut = 2 \times EstimatedRTT$$
$$0 < \alpha \leq 1$$



Exponential Averaging Example

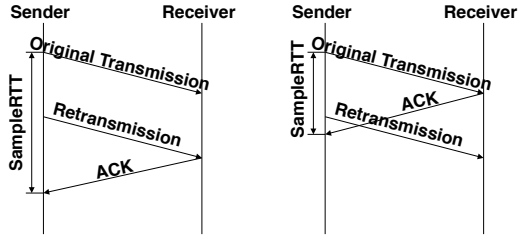
$$EstimatedRTT = \alpha * EstimatedRTT + (1 - \alpha) * SampleRTT$$

Assume RTT is constant → $SampleRTT$ = RTT



Problem

- How to differentiate between the real ACK, and ACK of the retransmitted packet?



Karn/Partridge Algorithm

- Measure *SampleRTT* only for original transmissions
- Exponential backoff → for each retransmission, double *EstimatedRTT*

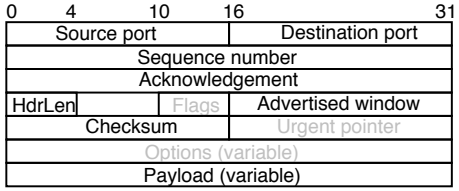
Jacobson/Karels Algorithm

- Problem: exponential average is not enough
  - One solution: use standard deviation (requires expensive square root computation)
  - Use mean deviation instead

$$Difference = SampleRTT - EstimatedRTT$$
$$EstimatedRTT = EstimatedRTT + \delta \times Difference$$
$$Deviation = Deviation + \delta \times (|Difference| - Deviation)$$
$$TimeOut = \mu \times EstimatedRTT + \phi \times Deviation$$
$$0 < \delta \leq 1$$
$$\mu = 1$$
$$\phi = 4$$

TCP Header

- Sequence number, acknowledgement, and advertised window – used by sliding-window based flow control
- HdrLen: TCP header length in 4-byte words
- Checksum: checksum of TCP header + payload



What did We Learn so Far?

- Packet switching (vs. circuit switching)
  - Store & forwarding: a packet is stored before being forwarded
  - Each packet is independently forwarded
- Statistical multiplexing:
  - Un-correlated bursty traffic → aggregate average is close to the peak aggregate bandwidth
- Layering: network organization
- E2E argument:
  - Think twice before in adding functionality at a lower layer unless that functionality can be fully implemented at that layer

What did We Learn so Far? (cont'd)

- Opening & closing a connection
- Flow control
- Reliability
  - Stop & wait
  - Sliding window (Go-back-n, selective repeat)
  - Retransmission timeout

Administrivia

- Project 2 due:
  - Code: Thursday, March 31st
  - Final document, peer evaluation: Friday, April 1st
- Project 3 starts after you are done with Project 2
- This Wednesday (March 30th), invited lecture
  - Sam Madden (MIT): Introduction in Databases
- Please answer class survey:
  - http://www.surveymonkey.com/s/MBF7TCK

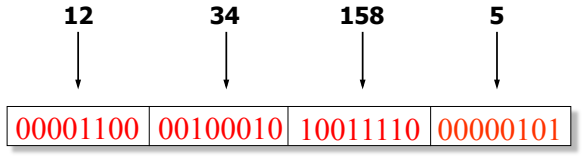
5min Break

Domain Name System (DNS)

- Concepts & principles underlying the Domain Name System (DNS)
  - **Indirection:** names in place of addresses
  - **Hierarchy:** in names, addresses, and servers
  - **Caching:** of mappings from names to/from addresses

IP Addresses (IPv4)

- A unique 32-bit number
- Identifies an interface (on a host, on a router, …)
- Represented in dotted-quad notation. E.g, **12.34.158.5**:



Host Names vs. IP addresses

- Host names
  - Mnemonic name appreciated by humans
  - Variable length, full alphabet of characters
  - Provide little (if any) information about location
  - Examples: www.cnn.com and bbc.co.uk
- IP addresses
  - Numerical address appreciated by routers
  - Fixed length, binary number
  - Hierarchical, related to host location
  - Examples: 64.236.16.20 and 212.58.224.131

Separating Naming and Addressing

- Names are easier to remember
  - www.cnn.com vs. 64.236.16.20
- Addresses can change underneath
  - Move www.cnn.com to 64.125.91.21
  - E.g., renumbering when changing providers
- Name could map to multiple IP addresses
  - www.cnn.com to multiple (8) replicas of the Web site
  - Enables
    - » Load-balancing
    - » Reducing latency by picking nearby servers
    - » Tailoring content based on requester's location/identity
- Multiple names for the same address
  - E.g., aliases like www.cnn.com and cnn.com

Scalable (Name ↔ Address) Mappings

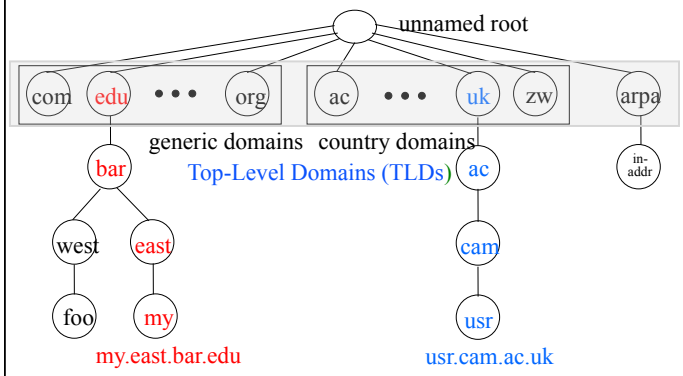
- Originally: per-host file
  - Flat namespace
  - **`/etc/hosts`** (what is this on your computer today?)
  - SRI (Menlo Park) kept master copy
  - Downloaded regularly
- Single server doesn't scale
  - Traffic implosion (lookups & updates)
  - Single point of failure
  - Amazing politics

Need a distributed, hierarchical collection of servers

Domain Name System (DNS)

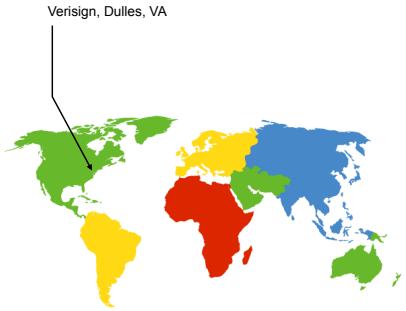
- Properties of DNS
  - Hierarchical name space divided into zones
  - Zones distributed over collection of DNS servers
- Hierarchy of DNS servers
  - Root (hardwired into other servers)
  - Top-level domain (TLD) servers
  - Authoritative DNS servers
- Performing the translations
  - Local DNS servers
  - Resolver software

Distributed Hierarchical Database



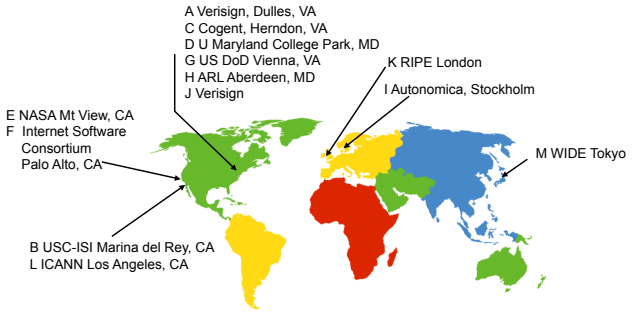
DNS Root

- Located in Virginia, USA
- How do we make the root scale?



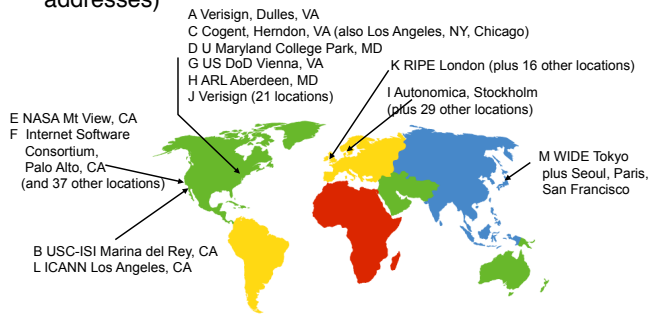
DNS Root Servers

- 13 root servers (see http://www.root-servers.org/)
  - Labeled A through M
- Does this scale?



DNS Root Servers

- 13 root servers (see http://www.root-servers.org/)
  - Labeled A through M
- Replication via any-casting (localized routing for addresses)



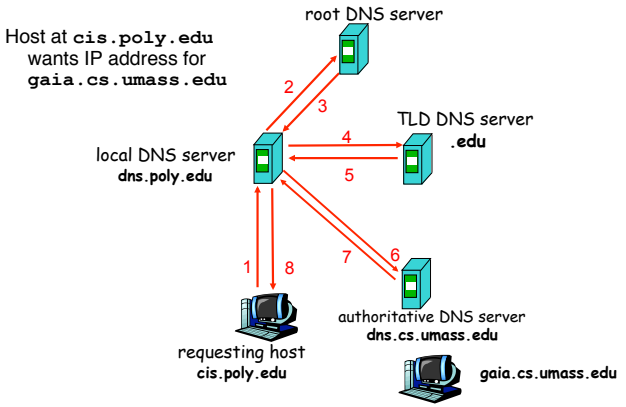
TLD and Authoritative DNS Servers

- Top-level domain (TLD) servers
  - Generic domains (e.g., com, org, edu)
  - Country domains (e.g., uk, fr, cn, jp)
  - Special domains (e.g., arpa)
  - Typically managed professionally
    - » Network Solutions maintains servers for "**com**"
    - » Educause maintains servers for "**edu**"
- Authoritative DNS servers
  - Provide public records for hosts at an organization
    - » Private records may differ, though not part of original design's intent
  - For the organization's servers (e.g., Web and mail)
  - Can be maintained locally or by a service provider

Using DNS

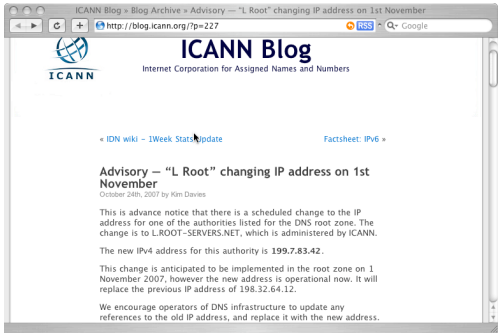
- Local DNS server ("default name server")
  - Usually near the endhosts that use it
  - Local hosts configured with local server (e.g., `/etc/resolv.conf`) or learn server via DHCP
- Client application
  - Extract server name (e.g., from the URL)
  - Do *gethostbyname()* to trigger resolver code
- Server application
  - Extract client IP address from socket
  - Optional *gethostbyaddr()* to translate into name

Example



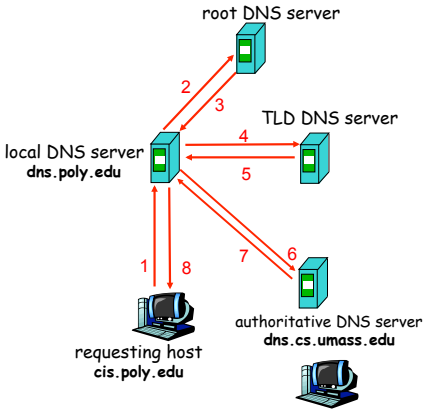
How did it know the root server IP?

- Hard-coded
- What if it changes?



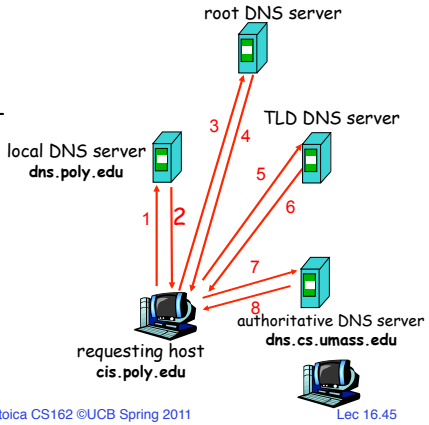
Recursive vs. Iterative Queries

- Recursive query
  - Ask server to get answer for you
  - E.g., request 1 and response 8



Recursive vs. Iterative Queries

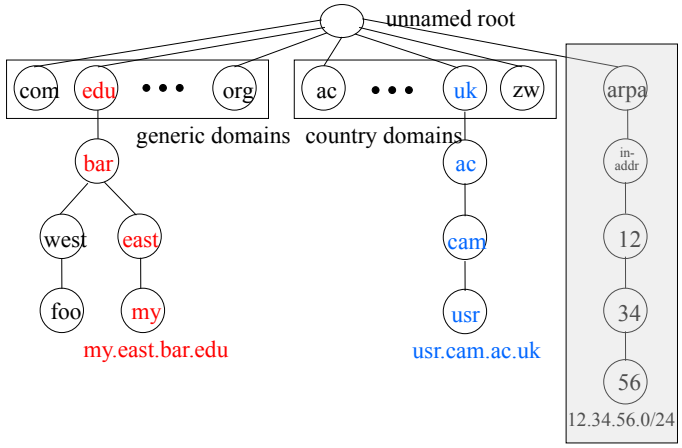
- Iterative query
  - Ask server who to ask next
  - E.g., all other request-response pairs



Reverse Mapping (Address → Host)

- How do we go the other direction, from an IP address to the corresponding hostname?
- Addresses already have natural "quad" hierarchy:
  - 12.34.56.78
- But: quad notation has most-sig. hierarchy element on left, while www.cnn.com has it on the right
- Idea: reverse the quads = 78.56.34.12 …
  - … and look that up in the DNS
- Under what TLD?
  - Convention: **in-addr.arpa**
  - So lookup is for `78.56.34.12.in-addr.arpa`

Distributed Hierarchical Database



DNS Caching

- Performing all these queries takes time
  - And all this before actual communication takes place
  - E.g., 1-second latency before starting Web download
- Caching can greatly reduce overhead
  - The top-level servers very rarely change
  - Popular sites (e.g., www.cnn.com) visited often
  - Local DNS server often has the information cached
- How DNS caching works
  - DNS servers cache responses to queries
  - Responses include a "time to live" (TTL) field
  - Server deletes cached entry after TTL expires

Negative Caching

- Remember things that don't work
  - Misspellings like *www.cnn.comm* and *www.cnnn.com*
  - These can take a long time to fail the first time
  - Good to remember that they don't work
  - … so the failure takes less time the next time around
- But: negative caching is optional
  - And not widely implemented

DNS Summary

- Distributed, hierarchical database
- Indirection gets us human-readable names, ability to change address, etc.
- Caching to improve performance