CS162
Operating Systems and Systems Programming
Lecture 12

Flow Control, DNS

October 10, 2011
Anthony D. Joseph and Ion Stoica
http://inst.eecs.berkeley.edu/~cs162

Goals for Today

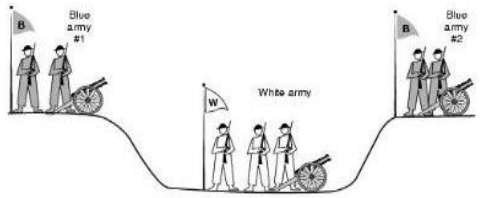
- Closing connection
- Flow control
- Retransmission timeout
- Domain Name Service (DNS)

TCP Service

1) Open connection: 3-way handshaking
2) Reliable byte stream transfer from (IPa, TCP_Port1) to (IPb, TCP_Port2)
3) Close (tear-down) connection

Close Connection (Two Generals Problem)

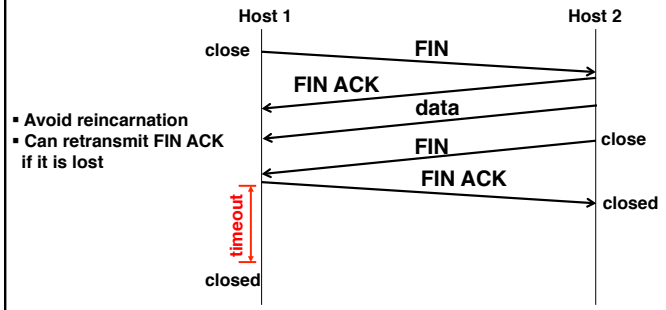
- Goal: both sides agree to close the connection
- Two-army problem:
  - "Two blue armies need to simultaneously attack the white army to win; otherwise they will be defeated. The blue army can communicate only across the area controlled by the white army which can intercept the messengers."



- What is the solution?

Close Connection

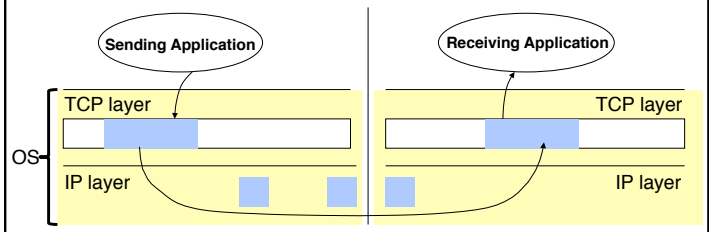
- 4-ways tear down connection



TCP Flow Control

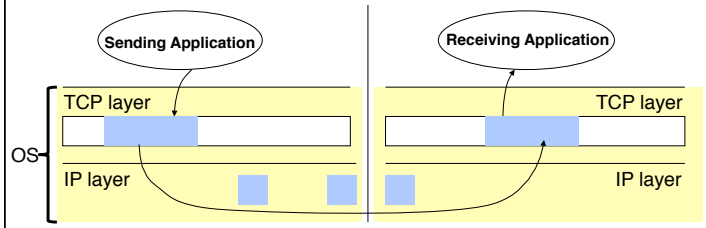
- TCP: stream oriented protocol
  - Sender sends a stream of bytes, not packets (e.g., no need to tell TCP how much you send)
  - Receiver reads a stream of bytes
- TCP flow control:
  - Sliding window protocol at byte (not packet) level
    - » Go-back-N: TCP Tahoe, Reno, New Reno
    - » Selective acknowledgement (SR): 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



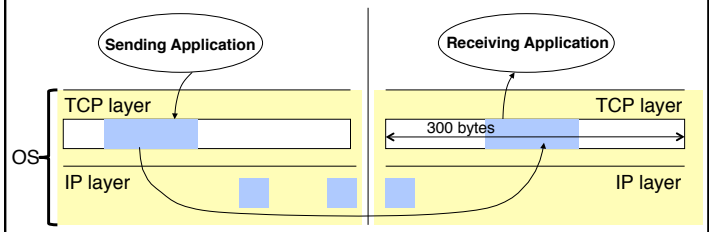
- TCP/IP implemented by OS (Kernel)
  - TCP and application run in different processes
  - Cannot do context switching on sending/receiving every packet
    - » At 1Gbps, it takes 12 usec to send an 1500 bytes, and 0.8usec to send an 100 byte packet
- Need buffers to match …
  - sending app with sending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

TCP Flow Control



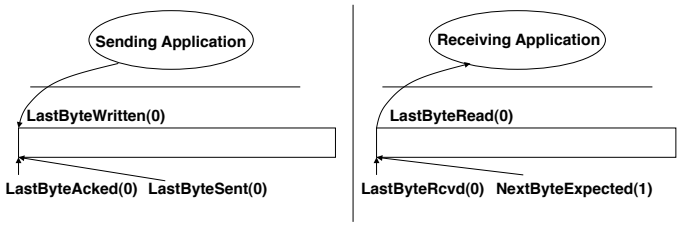
- Three pairs of producer-consumer's
  - sending app → sending TCP
  - sending TCP → receiving TCP
  - receiving TCP → receiving app
- How is mutual exclusion implemented?

TCP Flow Control



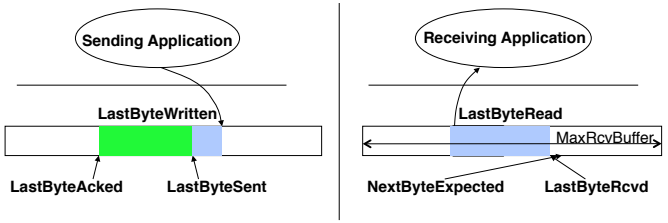
- Example assumptions:
  - Maximum IP packet size = 100 bytes
  - Size of the receiving buffer (MaxRcvBuf) = 300bytes
- Use circular buffers, i.e., N's byte is stored at (N mod MaxRcvBuf) in the buffer
- Recall, ack indicates the next expected byte in-sequence, not the last received byte

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



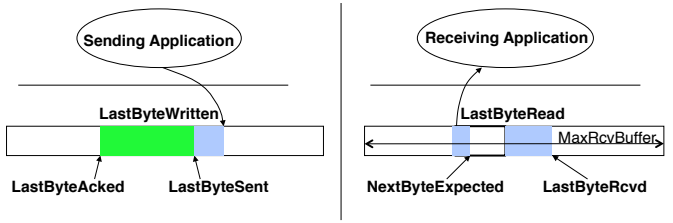
- AdvertisedWindow: number of bytes the receiver can receive

AdvertisedWindow = MaxRcvBuffer – (LastByteRcvd – LastByteRead)

- Sender window: number of bytes the sender can send

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

TCP Flow Control



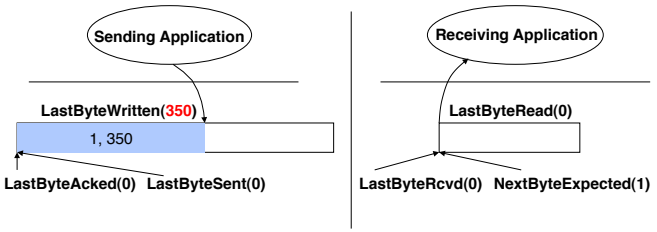
- Still true if receiver missed data….

AdvertisedWindow = MaxRcvBuffer – (LastByteRcvd – LastByteRead)

- Sender window: number of bytes the sender can send

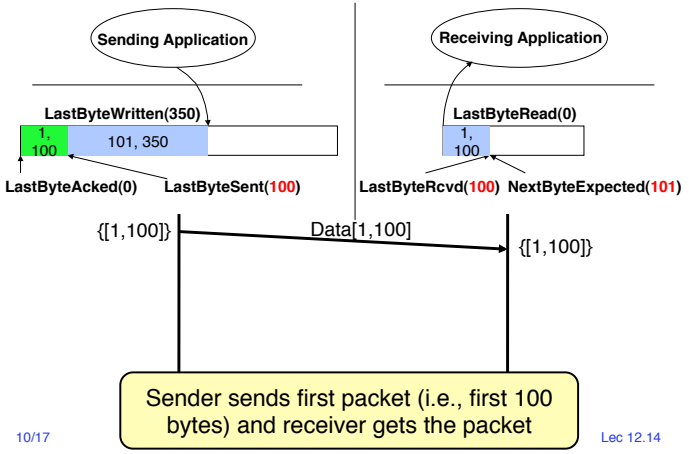
Sender window = AdvertisedWindow – (LastByteSent – LastByteAcked)
MaxSendBuffer >= LastByteWritten - LastByteAcked

TCP Flow Control

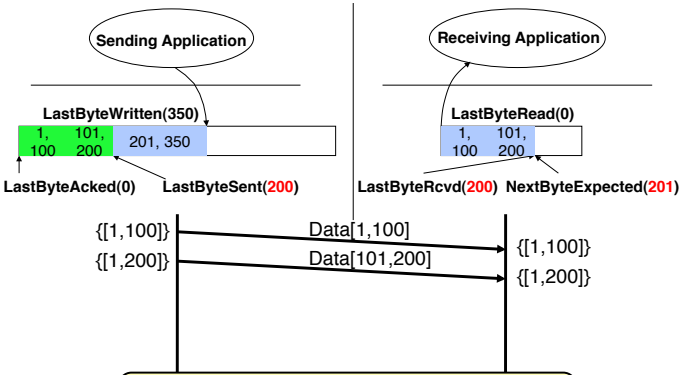


- Sending app sends 350 bytes
- Recall: we assume IP only accepts packets no larger than 100 bytes

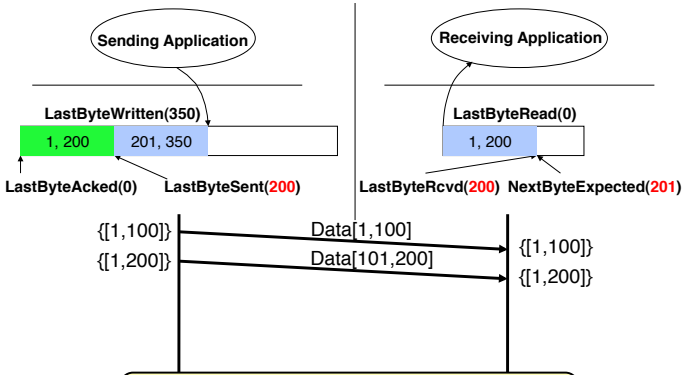
TCP Flow Control

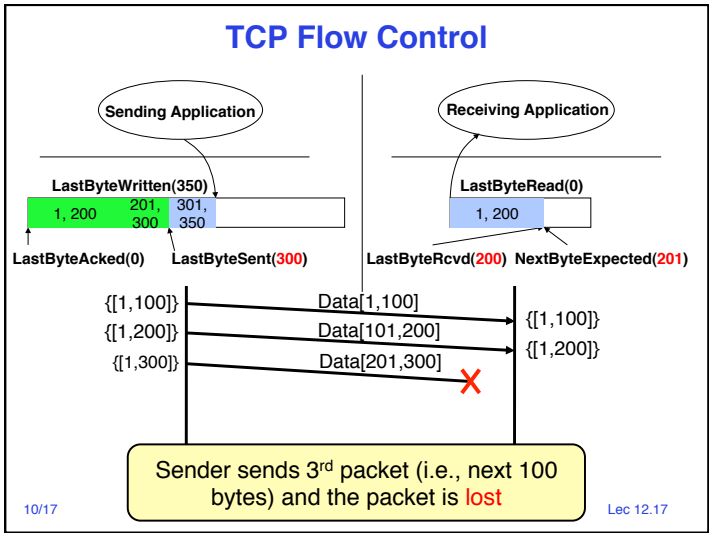


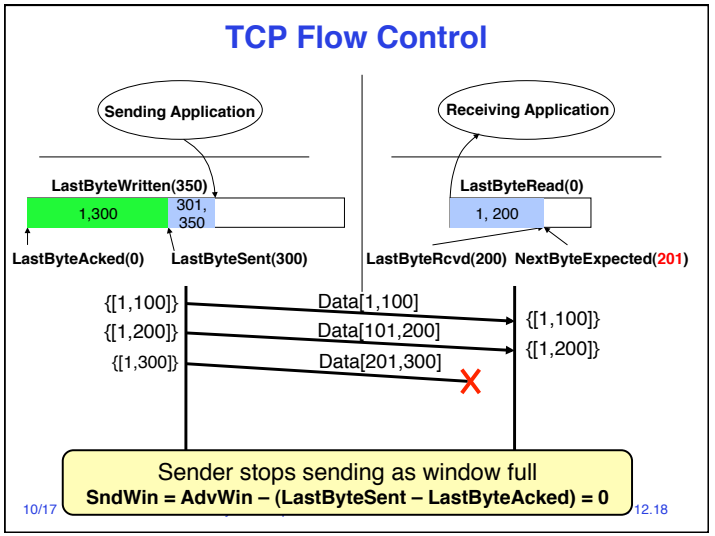
TCP Flow Control

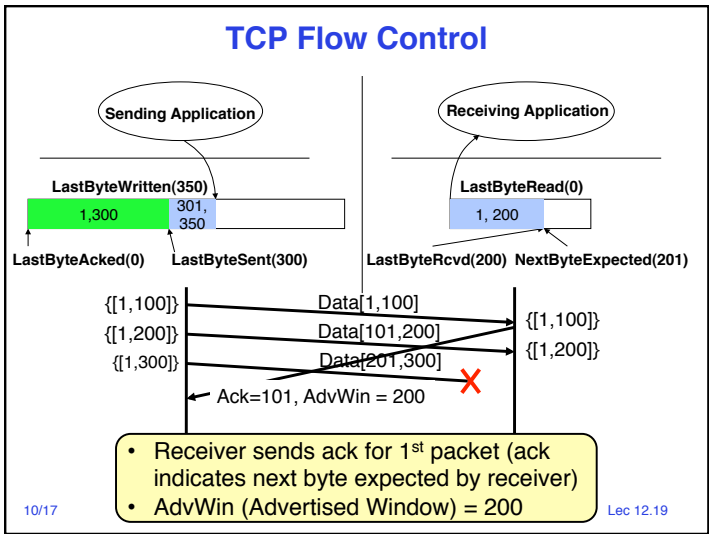


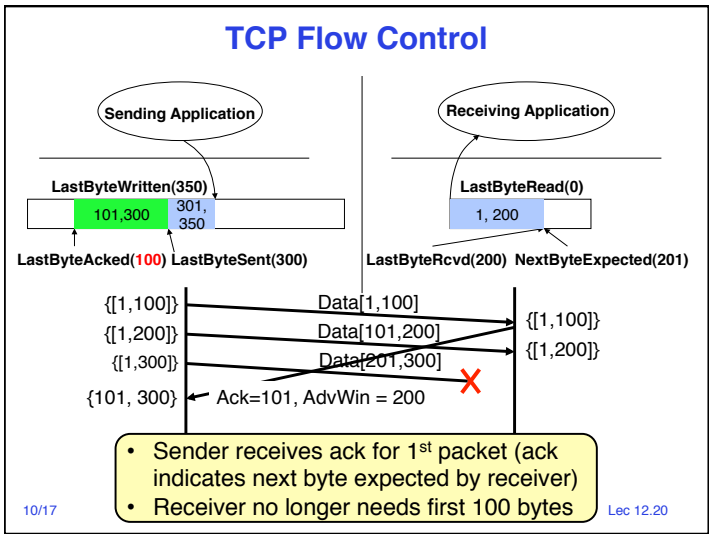
TCP Flow Control

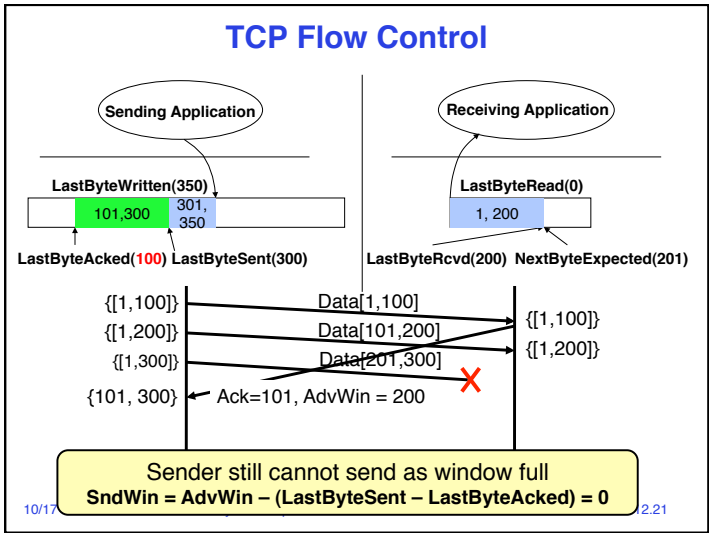


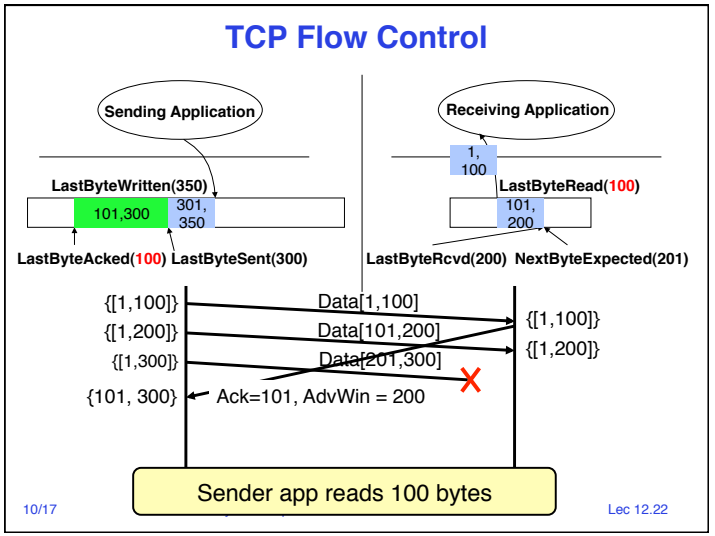

TCP Flow Control
Sending Application
Receiving Application
LastByteWritten(350)
1, 200
201, 300
301, 350
LastByteRead(0)
1, 200
LastByteAcked(0)
LastByteSent(300)
LastByteRcvd(200)
NextByteExpected(201)
{[1,100]}
{[1,200]}
{[1,300]}
Data[1,100]
Data[101,200]
Data[201,300]
{[1,100]}
{[1,200]}
Sender sends 3rd packet (i.e., next 100 bytes) and the packet is lost
10/17
Lec 12.17

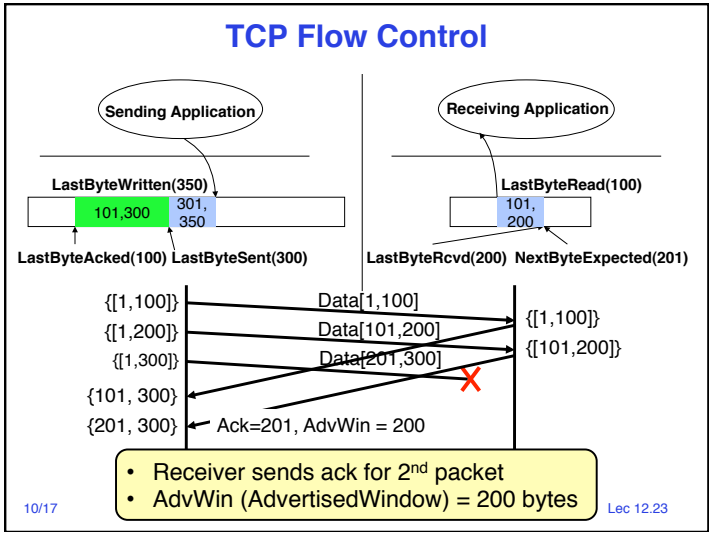

TCP Flow Control
Sending Application
Receiving Application
LastByteWritten(350)
1,300
301, 350
LastByteRead(0)
1, 200
LastByteAcked(0)
LastByteSent(300)
LastByteRcvd(200)
NextByteExpected(201)
{[1,100]}
{[1,200]}
{[1,300]}
Data[1,100]
Data[101,200]
Data[201,300]
{[1,100]}
{[1,200]}
Sender stops sending as window full
SndWin = AdvWin – (LastByteSent – LastByteAcked) = 0
10/17
12.18

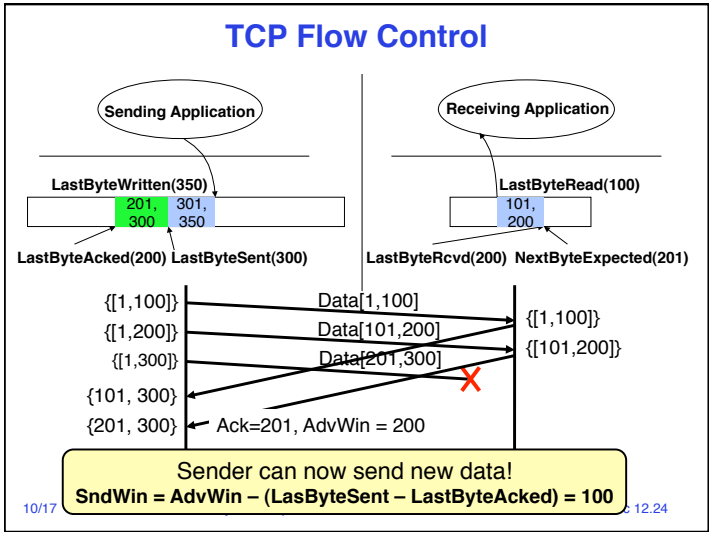

TCP Flow Control
Sending Application
Receiving Application
LastByteWritten(350)
1,300
301, 350
LastByteRead(0)
1, 200
LastByteAcked(0)
LastByteSent(300)
LastByteRcvd(200)
NextByteExpected(201)
{[1,100]}
{[1,200]}
{[1,300]}
Data[1,100]
Data[101,200]
Data[201,300]
{[1,100]}
{[1,200]}
Ack=101, AdvWin = 200
• Receiver sends ack for 1st packet (ack indicates next byte expected by receiver)
• AdvWin (Advertised Window) = 200
10/17
Lec 12.19

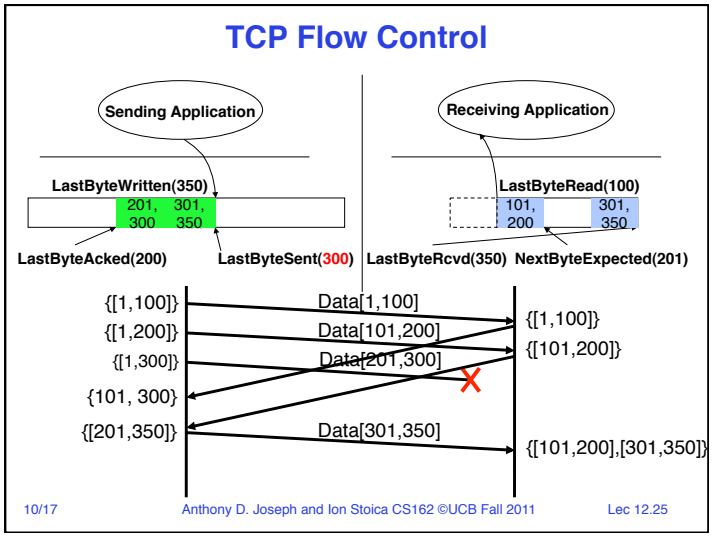

TCP Flow Control
Sending Application
Receiving Application
LastByteWritten(350)
101,300
301, 350
LastByteRead(0)
1, 200
LastByteAcked(100)
LastByteSent(300)
LastByteRcvd(200)
NextByteExpected(201)
{[1,100]}
{[1,200]}
{[1,300]}
Data[1,100]
Data[101,200]
Data[201,300]
{[1,100]}
{[1,200]}
{101, 300}
Ack=101, AdvWin = 200
• Sender receives ack for 1st packet (ack indicates next byte expected by receiver)
• Receiver no longer needs first 100 bytes
10/17
Lec 12.20

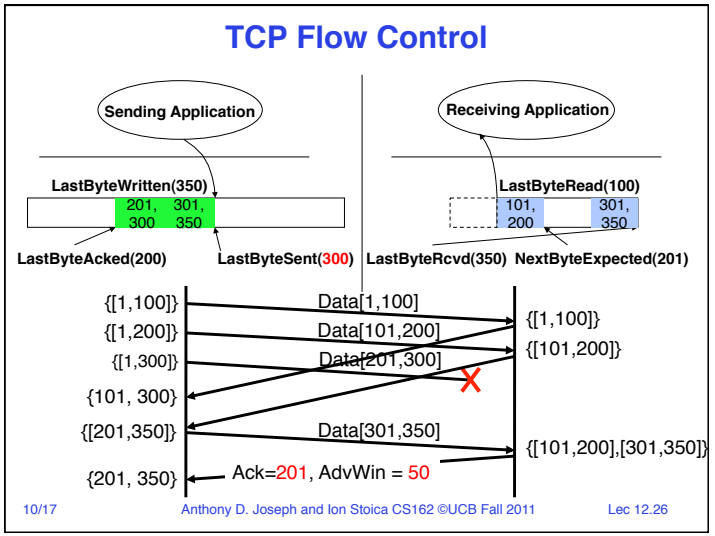

TCP Flow Control
Sending Application
Receiving Application
LastByteWritten(350)
101,300
301, 350
LastByteRead(0)
1, 200
LastByteAcked(100) LastByteSent(300)
LastByteRcvd(200) NextByteExpected(201)
{[1,100]}
{[1,200]}
{[1,300]}
{101, 300}
Data[1,100]
Data[101,200]
Data[201,300]
Ack=101, AdvWin = 200
{[1,100]}
{[1,200]}
Sender still cannot send as window full
SndWin = AdvWin – (LastByteSent – LastByteAcked) = 0
10/17
2.21

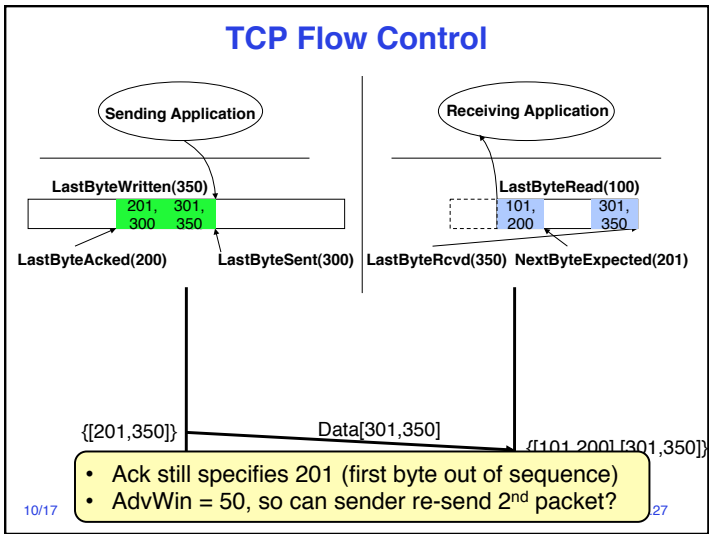

TCP Flow Control
Sending Application
Receiving Application
1, 100
LastByteWritten(350)
101,300
301, 350
LastByteRead(100)
101, 200
LastByteAcked(100) LastByteSent(300)
LastByteRcvd(200) NextByteExpected(201)
{[1,100]}
{[1,200]}
{[1,300]}
{101, 300}
Data[1,100]
Data[101,200]
Data[201,300]
Ack=101, AdvWin = 200
{[1,100]}
{[1,200]}
Sender app reads 100 bytes
10/17
Lec 12.22

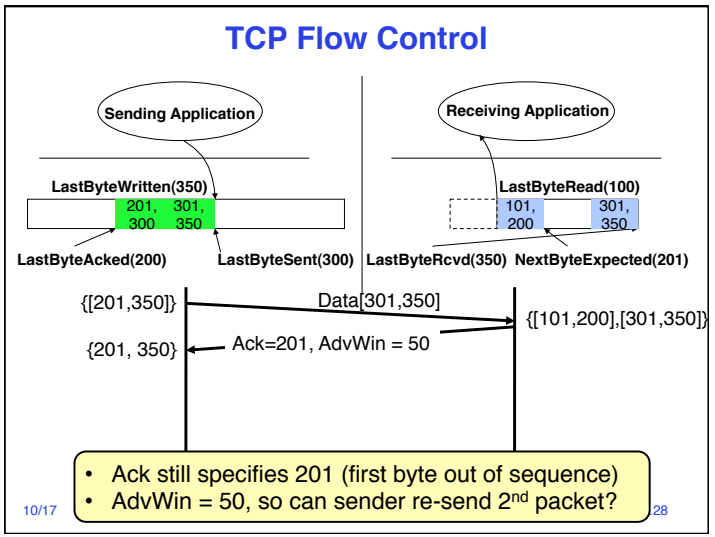

TCP Flow Control
Sending Application
Receiving Application
LastByteWritten(350)
101,300
301, 350
LastByteRead(100)
101, 200
LastByteAcked(100) LastByteSent(300)
LastByteRcvd(200) NextByteExpected(201)
{[1,100]}
{[1,200]}
{[1,300]}
{101, 300}
{201, 300}
Data[1,100]
Data[101,200]
Data[201,300]
Ack=201, AdvWin = 200
{[1,100]}
{[101,200]}
• Receiver sends ack for 2nd packet
• AdvWin (AdvertisedWindow) = 200 bytes
10/17
Lec 12.23


TCP Flow Control
Sending Application
Receiving Application
LastByteWritten(350)
201, 300
301, 350
LastByteRead(100)
101, 200
LastByteAcked(200) LastByteSent(300)
LastByteRcvd(200) NextByteExpected(201)
{[1,100]}
{[1,200]}
{[1,300]}
{101, 300}
{201, 300}
Data[1,100]
Data[101,200]
Data[201,300]
Ack=201, AdvWin = 200
{[1,100]}
{[101,200]}
Sender can now send new data!
SndWin = AdvWin – (LasByteSent – LastByteAcked) = 100
10/17
12.24

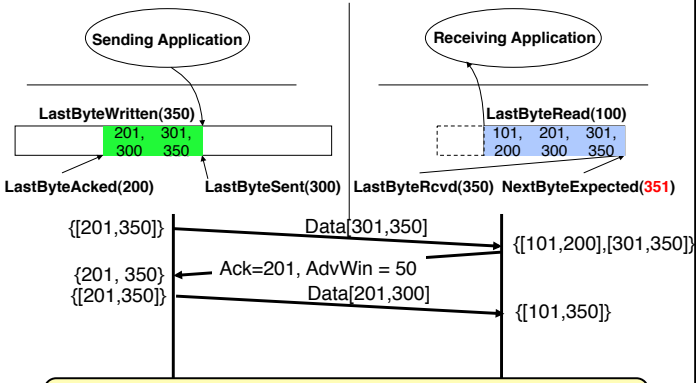

TCP Flow Control
Sending Application
Receiving Application
LastByteWritten(350)
201, 300
301, 350
LastByteRead(100)
101, 200
301, 350
LastByteAcked(200)
LastByteSent(300)
LastByteRcvd(350)
NextByteExpected(201)
{[1,100]}
{[1,200]}
{[1,300]}
{101, 300}
{[201,350]}
Data[1,100]
Data[101,200]
Data[201,300]
Data[301,350]
{[1,100]}
{[101,200]}
{[101,200],[301,350]}
10/17
Anthony D. Joseph and Ion Stoica CS162 ©UCB Fall 2011
Lec 12.25


TCP Flow Control
Sending Application
Receiving Application
LastByteWritten(350)
201, 300
301, 350
LastByteRead(100)
101, 200
301, 350
LastByteAcked(200)
LastByteSent(300)
LastByteRcvd(350)
NextByteExpected(201)
{[1,100]}
{[1,200]}
{[1,300]}
{101, 300}
{[201,350]}
{201, 350}
Data[1,100]
Data[101,200]
Data[201,300]
Data[301,350]
Ack=201, AdvWin = 50
{[1,100]}
{[101,200]}
{[101,200],[301,350]}
10/17
Anthony D. Joseph and Ion Stoica CS162 ©UCB Fall 2011
Lec 12.26


TCP Flow Control
Sending Application
Receiving Application
LastByteWritten(350)
201, 300
301, 350
LastByteRead(100)
101, 200
301, 350
LastByteAcked(200)
LastByteSent(300)
LastByteRcvd(350)
NextByteExpected(201)
{[201,350]}
Data[301,350]
{[101,200],[301,350]}
• Ack still specifies 201 (first byte out of sequence)
• AdvWin = 50, so can sender re-send 2nd packet?
10/17
27

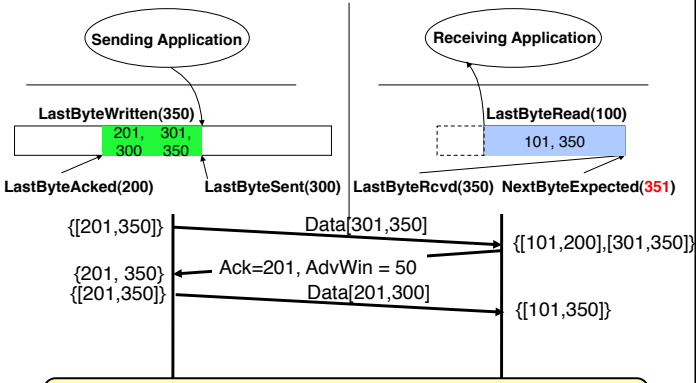

TCP Flow Control
Sending Application
Receiving Application
LastByteWritten(350)
201, 300
301, 350
LastByteRead(100)
101, 200
301, 350
LastByteAcked(200)
LastByteSent(300)
LastByteRcvd(350)
NextByteExpected(201)
{[201,350]}
{201, 350}
Data[301,350]
Ack=201, AdvWin = 50
{[101,200],[301,350]}
• Ack still specifies 201 (first byte out of sequence)
• AdvWin = 50, so can sender re-send 2nd packet?
10/17
28

TCP Flow Control



Yes! Sender can re-send 2nd packet since it's in existing window – won't cause receiver window to grow

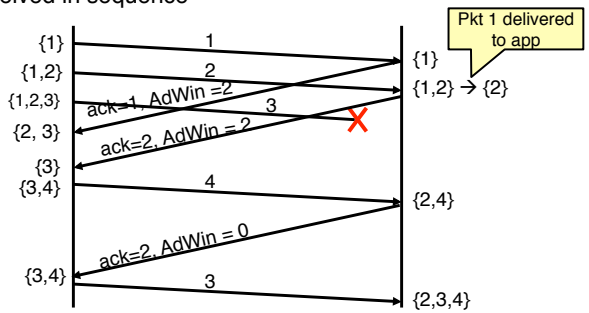
TCP Flow Control



Yes! Sender can re-send 2nd packet since it's in existing window – won't cause receiver window to grow

Recap: Packet-level Flow Control

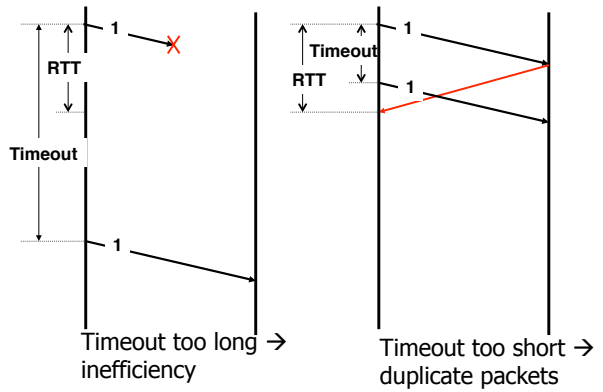
- Implement flow control at packet level (see last lecture)
- Advertised window is 3 packets
- Ack packet specifies the seq. number of last packet received in sequence



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
  - Use exponential averaging to estimate RTT

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 adding functionality at a lower layer, do it if and only if
  - Improves dramatically the performance of apps that need it
  - Doesn't hurt performance of apps that don't need it

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

Midterm

- Midterm: Thursday, October 13, 5-6:30pm in 155 Dwinelle
  - Up to and including lecture 11
  - Closed book, 1 cheat sheet (two sides)
- Materials: everything up to laste lecture, i.e., Lecture 11 (Reliability, TCP Connection Setup)
- Midterm review: Today, 7:30-9:30pm, 306 Soda Hall
- Ion's office hour change:
  - 11-12am → 10-11am, Wednesday, October 12
  - Additional office hour: 6:30-7:30pm, Wednesday, October 12

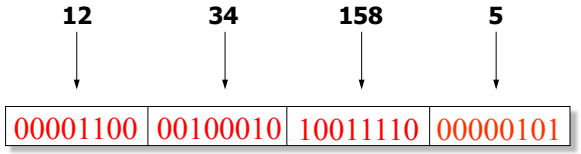
5min Break

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

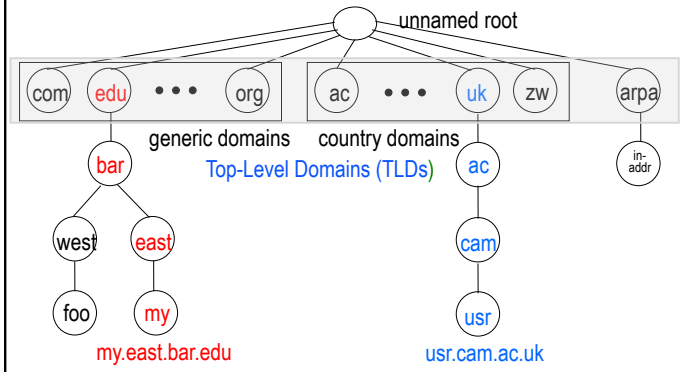
- Originally: per-host file
  - Flat namespace
  - `/etc/hosts` (what is this on your computer today?)
  - SRI (Menlo Park) kept master copy
  - Downloaded regularly
- Single server doesn't scale
  - Traffic implosion (lookups & updates)
  - Single point of failure
  - Amazing politics

Need a distributed, hierarchical collection of servers

Domain Name System (DNS)

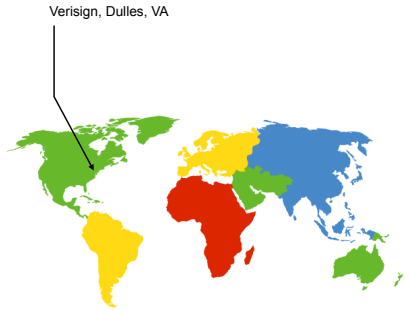
- Properties of DNS
  - Hierarchical name space divided into zones
  - Zones distributed over collection of DNS servers
- Hierarchy of DNS servers
  - Root (hardwired into other servers)
  - Top-level domain (TLD) servers
  - Authoritative DNS servers
- Performing the translations
  - Local DNS servers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

Distributed Hierarchical Database



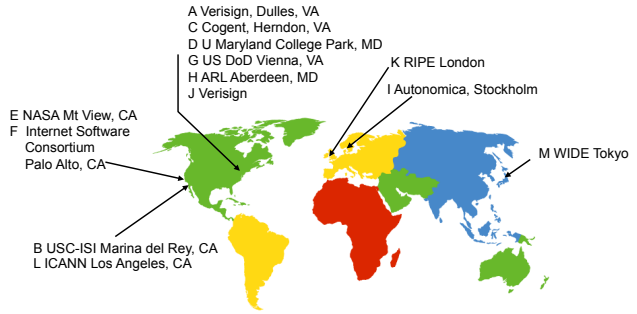
DNS Root

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



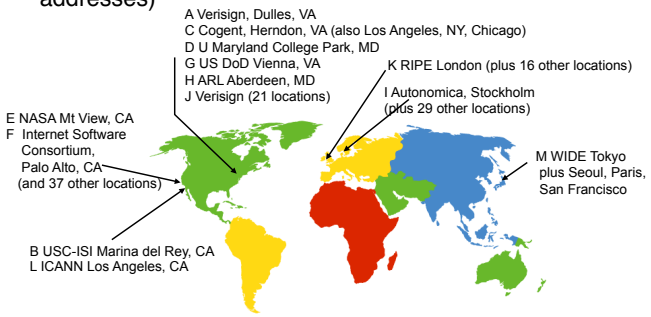
DNS Root Servers

- 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)



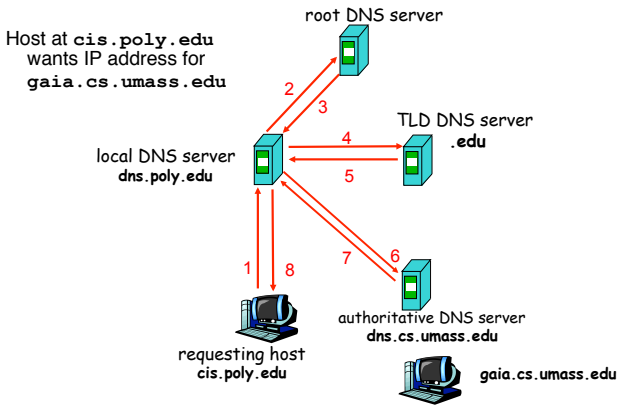
TLD and Authoritative DNS Servers

- 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

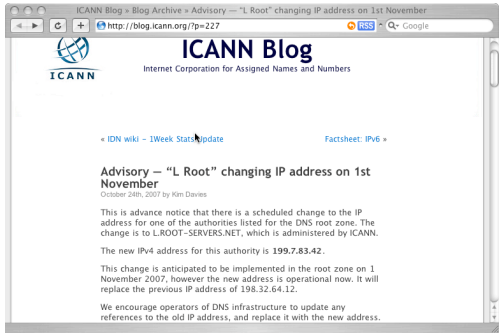
- Local DNS server ("default name server")
  - Usually near the endhosts that use it
  - Local hosts configured with local server (e.g., `/etc/resolv.conf`)
- Extract server name (e.g., from the URL)
  - Do *gethostbyname()* to trigger resolver code

Example



How did it know the root server IP?

- Hard-coded
- What if it changes?



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