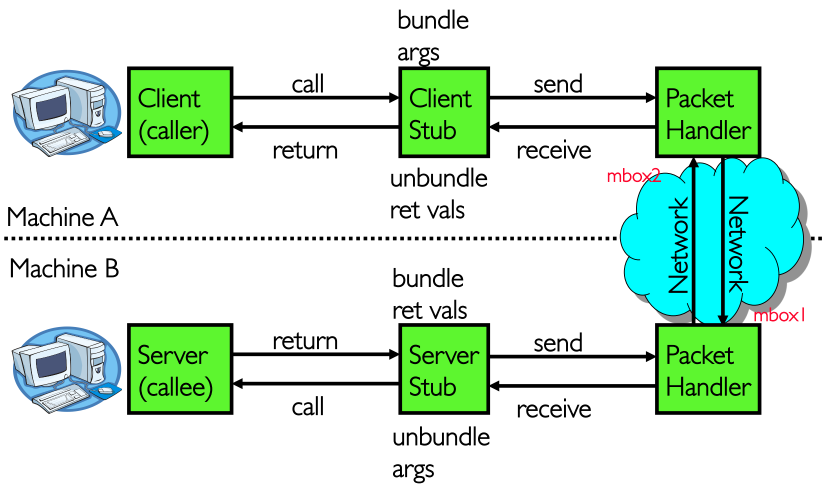
CS162
Operating Systems and Systems Programming
Lecture 22

TCP/IP (Continued)

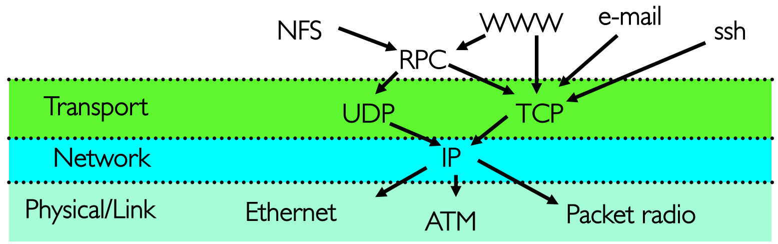
April 18th, 2016
Prof. Anthony D. Joseph
http://cs162.eecs.Berkeley.edu

Recall: RPC Information Flow



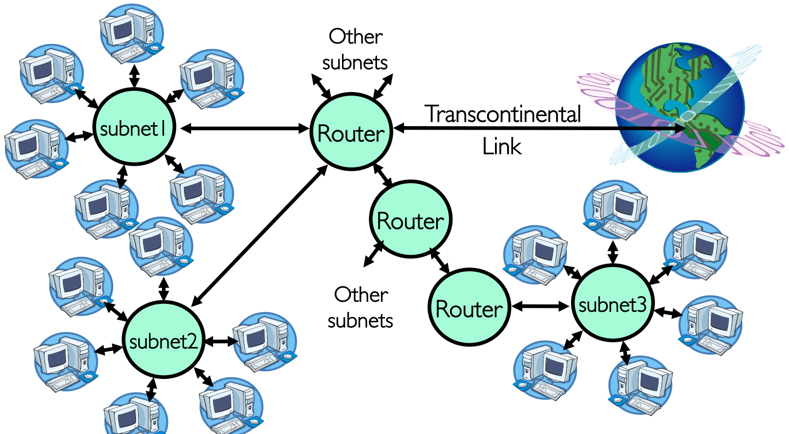
Recall: Network Protocols

- Networking protocols: many levels
  - Physical level: mechanical and electrical network (e.g., how are 0 and 1 represented)
  - Link level: packet formats/error control (for instance, the CSMA/CD protocol)
  - Network level: network routing, addressing
  - Transport Level: reliable message delivery
- Protocols on today's Internet:



Hierarchical Networking: The Internet

- How can we build a network with millions of hosts?
  - Hierarchy! Not every host connected to every other one
  - Use a network of Routers to connect subnets together
    - » Routing is often by prefix: e.g. first router matches first 8 bits of address, next router matches more, etc.



Simple Network Terminology

- Local-Area Network (LAN) – designed to cover small geographical area
  - Multi-access bus, ring, or star network
  - Speed ≈ 100 – 10,000 Megabits/second (even 40-100Gb/s)
  - Broadcast is fast and cheap
  - In small organization, a LAN could consist of a single subnet. In large organizations (like UC Berkeley), a LAN contains many subnets

- Wide-Area Network (WAN) – links geographically separated sites
  - Point-to-point connections over long-haul lines (often leased from a phone company)
  - Speed ≈ 1.544 – 155 Megabits/second (even 100Gb/s to 8,800Tb/s)
  - Broadcast usually requires multiple messages

Routing

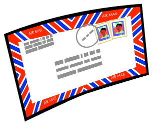
- Routing: the process of forwarding packets hop-by-hop through routers to reach their destination
  - Need more than just a destination address!
    - » Need a path
  - Post Office Analogy:
    - » Destination address on each letter is not sufficient to get it to the destination
    - » To get a letter from here to Florida, must route to local post office, sorted and sent on plane to somewhere in Florida, be routed to post office, sorted and sent with carrier who knows where street and house is…
- Internet routing mechanism: routing tables
  - Each router does table lookup to decide which link to use to get packet closer to destination
  - Don't need 4 billion entries in table: routing is by subnet
  - Could packets be sent in a loop? Yes, if tables incorrect
- Routing table contains:
  - Destination address range → output link closer to destination
  - Default entry (for subnets without explicit entries)

Setting up Routing Tables

- How do you set up routing tables?
  - Internet has no centralized state!
    - » No single machine knows entire topology
    - » Topology constantly changing (faults, reconfiguration, etc.)
  - Need dynamic algorithm that acquires routing tables
    - » Ideally, have one entry per subnet or portion of address
    - » Could have "default" routes that send packets for unknown subnets to a different router that has more information
- Possible algorithm for acquiring routing table
  - Routing table has "cost" for each entry
    - » Includes number of hops to destination, congestion, etc.
    - » Entries for unknown subnets have infinite cost
  - Neighbors periodically exchange routing tables
    - » If neighbor knows cheaper route to a subnet, replace your entry with neighbors entry (+1 for hop to neighbor)
- In reality:
  - Internet has networks of many different scales
  - Different algorithms run at different scales

Naming in the Internet

Name ⇒ Address

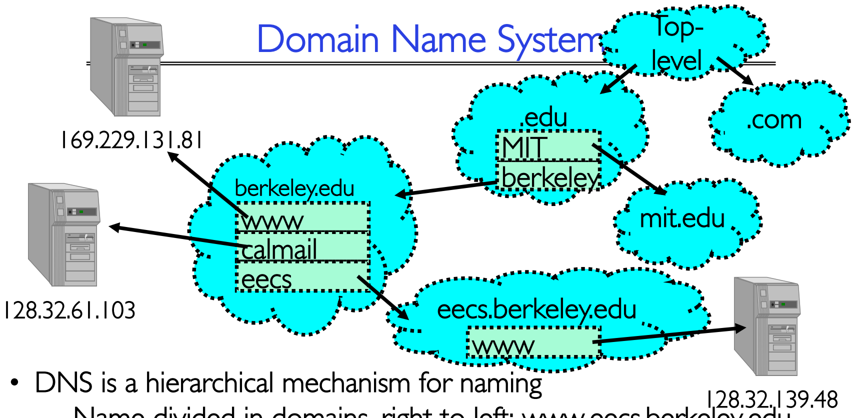


- How to map human-readable names to IP addresses?
  - E.g., www.berkeley.edu ⇒ 128.32.139.48
  - E.g., www.google.com ⇒ different addresses depending on location, load

- Why is this necessary?
  - IP addresses are hard to remember
  - IP addresses change:
    - » Say, Server 1 crashes and gets replaced by Server 2
    - » Or – www.google.com handled by different servers

- Mechanism: Domain Naming System (DNS)

Domain Name System



- DNS is a hierarchical mechanism for naming
  - Name divided in domains, right to left: www.eecs.berkeley.edu
- Each domain owned by a particular organization
  - Top level handled by ICANN (Internet Corporation for Assigned Numbers and Names)
  - Subsequent levels owned by organizations
- Resolution: series of queries to successive servers
- Caching: queries take time, so results cached for period of time

How Important is Correct Resolution?

- If attacker manages to give incorrect mapping:
  - Cause someone to route to server, thinking they're routing to different one
    - » Trick them into logging into "bank" – give up username and password
- DNS is insecure (a weak link)
  - What if "response" is returned from different server than original query?
  - Cause person to use incorrect IP address!
- In July 2008, hole in DNS security identified!
  - Security researcher Dan Kaminsky discovered attack that broke DNS
    - » One person in an ISP convinced to load particular web page, then *all* users of that ISP end up pointing at wrong address
  - High profile, highly advertised need for patching DNS
    - » Big press release, lots of mystery
    - » Security researchers told not speculate on cause until patches applied
- A solution? Domain Name System Security Extensions (DNSSEC)
  - Several IETF RFCs for using Public Key Cryptography to sign DNS records
  - Many deployment challenges!
    - » ISPs, Domain registrars, clients OSes, DNS servers, …
  - Many protocol challenges – backward compatibility, info leakage, …

Network Layering

- Layering: building complex services from simpler ones
  - Each layer provides services needed by higher layers by utilizing services provided by lower layers
- The physical/link layer is pretty limited
  - Packets are of limited size (called the "Maximum Transfer Unit or MTU: often 200-1500 bytes in size)
  - Routing is limited to within a physical link (wire) or through a switch
- Our goal in the following is to show how to construct a secure, ordered, message service routed to anywhere:

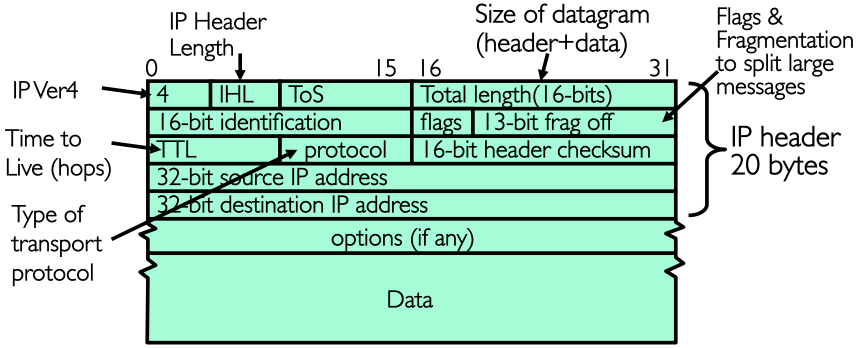
Physical Reality: Packets	Abstraction: Messages
Limited Size	Arbitrary Size
Unordered (sometimes)	Ordered
Unreliable	Reliable
Machine-to-machine	Process-to-process
Only on local area net	Routed anywhere
Asynchronous	Synchronous
Insecure	Secure

Building a Messaging Service

- Handling Arbitrary Sized Messages:
  - Must deal with limited physical packet size
  - Split big message into smaller ones (called fragments)
    - » Must be reassembled at destination
  - Checksum computed on each fragment or whole message
- Internet Protocol (IP): Must find way to send packets to arbitrary destination in network
  - Deliver messages unreliably ("best effort") from one machine in Internet to another
  - Since intermediate links may have limited size, must be able to fragment/reassemble packets on demand
  - Includes 256 different "sub-protocols" build on top of IP
    - » Examples: ICMP(1), TCP(6), UDP (17), IPSEC(50,51)

IP Packet Format

- IP Packet Format:



IP Header Length

Size of datagram (header+data)

Flags & Fragmentation to split large messages

0 15 16 31

IP Ver4

Time to Live (hops)

Type of transport protocol

4 | IHL | ToS | Total length(16-bits)

16-bit identification | flags | 13-bit frag off

TTL | protocol | 16-bit header checksum

32-bit source IP address

32-bit destination IP address

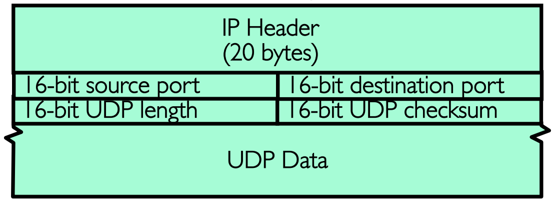
options (if any)

Data

IP header 20 bytes

Building a Messaging Service

- Process to process communication
  - Basic routing gets packets from machine→machine
  - What we really want is routing from process→process
    - » Add "ports", which are 16-bit identifiers
    - » A communication channel (connection) defined by 5 items: [source addr, source port, dest addr, dest port, protocol]
- UDP: The Unreliable Datagram Protocol (called UDP/IP)
  - Layered on top of basic IP (IP Protocol 17)
    - » Datagram: unreliable, unordered, packet sent from source user → dest user



IP Header (20 bytes)

16-bit source port | 16-bit destination port

16-bit UDP length | 16-bit UDP checksum

UDP Data

  - Important aspect: low overhead!
    - » Often used for high-bandwidth bi-directional audio/video streams
    - » Many uses of UDP considered "anti-social" – none of the "well-behaved" aspects of (say) TCP/IP

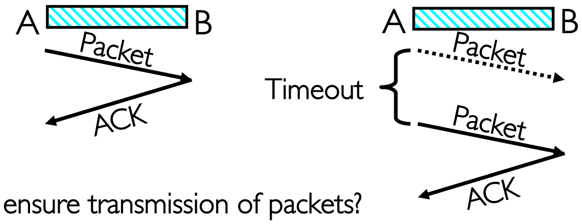
Ordered Messages

- Ordered Messages
  - Several network services are best constructed by ordered messaging
    - » Ask remote machine to first do x, then do y, etc.
  - Unfortunately, underlying network is packet based:
    - » Packets are routed one at a time through the network
    - » Can take different paths or be delayed individually
  - IP can reorder packets! P_0,P_1 might arrive as P_1,P_0
- Solution requires queuing at destination
  - Need to hold onto packets to undo out of order arrivals
  - Total degree of reordering impacts queue size
- Ordered messages on top of unordered ones:
  - Assign sequence numbers to packets
    - » 0,1,2,3,4…..
    - » If packets arrive out of order, reorder before delivering to user application
    - » For instance, hold onto #3 until #2 arrives, etc.
  - Sequence numbers are specific to particular connection
    - » Reordering among connections normally doesn't matter
  - If restart connection, need to make sure use different range of sequence numbers than previously…

Reliable Message Delivery: the Problem

- All physical networks can garble and/or drop packets
  - Physical media: packet not transmitted/received
    - » If transmit close to maximum rate, get more throughput – even if some packets get lost
    - » If transmit at lowest voltage such that error correction just starts correcting errors, get best power/bit
  - Congestion: no place to put incoming packet
    - » Point-to-point network: insufficient queue at switch/router
    - » Broadcast link: two host try to use same link
    - » In any network: insufficient buffer space at destination
    - » Rate mismatch: what if sender send faster than receiver can process?
- Reliable Message Delivery on top of Unreliable Packets
  - Need some way to make sure that packets actually make it to receiver
    - » Every packet received at least once
    - » Every packet received at most once
  - Can combine with ordering: every packet received by process at destination exactly once and in order

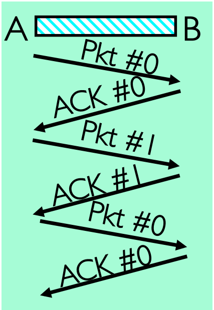
Using Acknowledgements



- How to ensure transmission of packets?
  - Detect garbling at receiver via checksum, discard if bad
  - Receiver acknowledges (by sending "ACK") when packet received properly at destination
  - Timeout at sender: if no ACK, retransmit
- Some questions:
  - If the sender doesn't get an ACK, does that mean the receiver didn't get the original message?
    - » No
  - What if ACK gets dropped? Or if message gets delayed?
    - » Sender doesn't get ACK, retransmits, Receiver gets message twice, ACK each

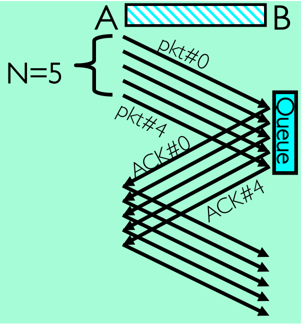
How to Deal with Message Duplication?

- Solution: put sequence number in message to identify re-transmitted packets
  - Receiver checks for duplicate number's; Discard if detected
- Requirements:
  - Sender keeps copy of unACK'd messages
    - » Easy: only need to buffer messages
  - Receiver tracks possible duplicate messages
    - » Hard: when ok to forget about received message?
- Alternating-bit protocol:
  - Send one message at a time; don't send next message until ACK received
  - Sender keeps last message; receiver tracks sequence number of last message received
- Pros: simple, small overhead
- Con: Poor performance
  - Wire can hold multiple messages; want to fill up at (wire latency × throughput)
- Con: doesn't work if network can delay or duplicate messages arbitrarily



Better Messaging: Window-based Acknowledgements

- Windowing protocol (not quite TCP):
  - Send up to N packets without ack
    - » Allows pipelining of packets
    - » Window size (N) < queue at destination
  - Each packet has sequence number
    - » Receiver acknowledges each packet
    - » ACK says "received all packets up to sequence number X"/send more
- ACKs serve dual purpose:
  - Reliability: Confirming packet received
  - Ordering: Packets can be reordered at destination
- What if packet gets garbled/dropped?
  - Sender will timeout waiting for ACK packet
    - » Resend missing packets ⇒ Receiver gets packets out of order!
  - Should receiver discard packets that arrive out of order?
    - » Simple, but poor performance
  - Alternative: Keep copy until sender fills in missing pieces?
    - » Reduces # of retransmits, but more complex
- What if ACK gets garbled/dropped?
  - Timeout and resend just the un-acknowledged packets

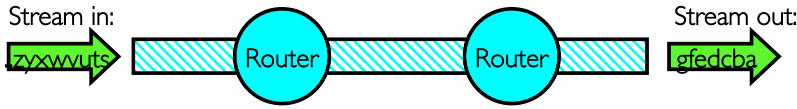


Administrivia

- Midterm II: this Wednesday! (4/20) [no lecture]
  - 6-7:30PM (aa-eh 10 Evans, ej-oa 155 Dwinelle)
  - Covers lectures #13 to 21 (assumes knowledge of #1 – 12)
    - » Address Translation/TLBs/Paging
    - » I/O subsystems, Storage Layers, Disks/SSD
    - » Performance and Queuing Theory
    - » File systems
    - » Distributed systems, 2PC, RPC
  - Closed book, no calculators
  - 1 page of hand-written notes, both sides
- Review session: Today 6:30-8:00 PM in 245 Li Ka Shing

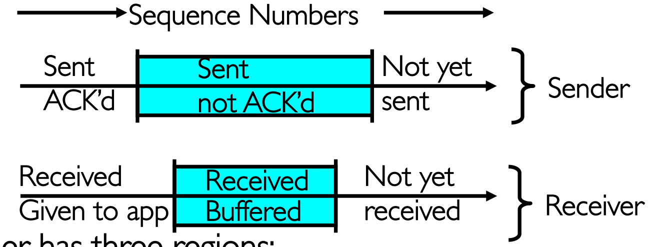
BREAK

Transmission Control Protocol (TCP)



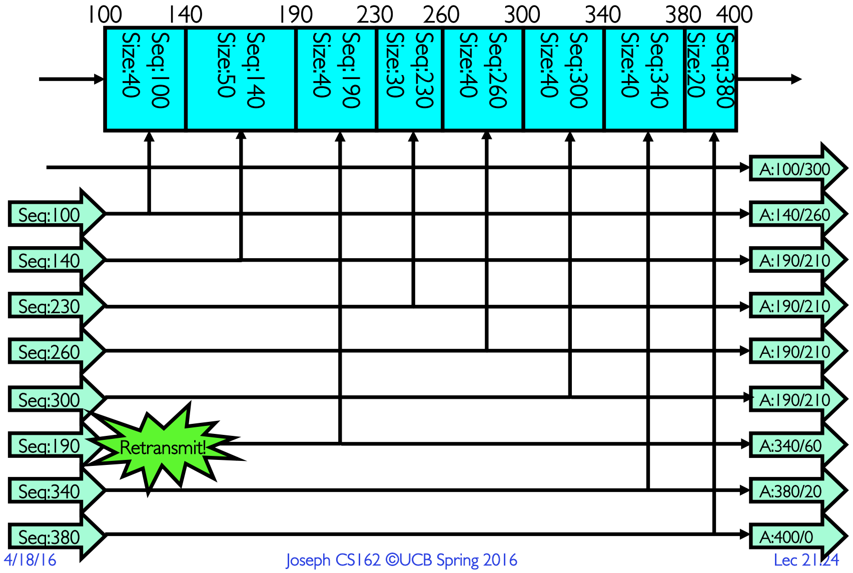
- Transmission Control Protocol (TCP)
  - TCP (IP Protocol 6) layered on top of IP
  - Reliable byte stream between two processes on different machines over Internet (read, write, flush)
- TCP Details
  - Fragments byte stream into packets, hands packets to IP
    - » IP may also fragment by itself
  - Uses window-based acknowledgement protocol (to minimize state at sender and receiver)
    - » "Window" reflects storage at receiver – sender shouldn't overrun receiver's buffer space
    - » Also, window should reflect speed/capacity of network – sender shouldn't overload network
  - Automatically retransmits lost packets
  - Adjusts rate of transmission to avoid congestion
    - » A "good citizen"

TCP Windows and Sequence Numbers

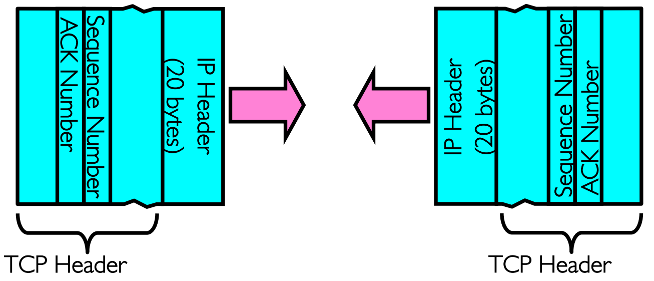


- Sender has three regions:
  - Sequence regions
    - » sent and ACK'd
    - » Sent and not ACK'd
    - » not yet sent
  - Window (colored region) adjusted by sender
- Receiver has three regions:
  - Sequence regions
    - » received and ACK'd (given to application)
    - » received and buffered
    - » not yet received (or discarded because out of order)

Window-Based Acknowledgements (TCP)



Selective Acknowledgement Option (SACK)



- Vanilla TCP Acknowledgement
  - Every message encodes Sequence number and ACK
  - Can include data for forward stream and/or ACK for reverse stream
- Selective Acknowledgement (SACK)
  - Acknowledgement information includes not just one number, but rather ranges of received packets
  - Must be specially negotiated at beginning of TCP setup
    - » SACK is widely used — all popular TCP stacks support it

Congestion Avoidance

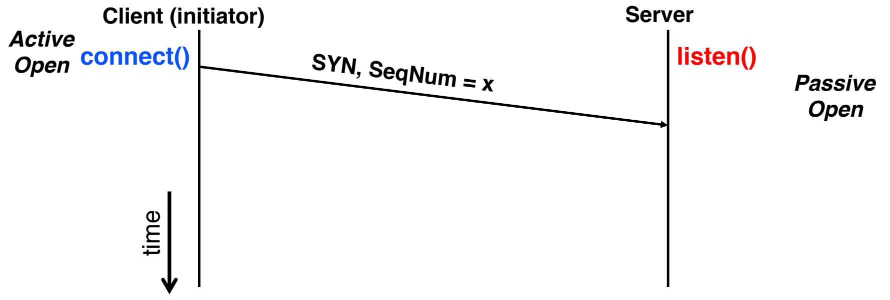
- Congestion
  - How long should timeout be for re-sending messages?
    - » Too long→wastes time if message lost
    - » Too short→retransmit even though ack will arrive shortly
  - Stability problem: more congestion ⇒ ack is delayed ⇒ unnecessary timeout ⇒ more traffic ⇒ more congestion
    - » Closely related to window size at sender: too big means putting too much data into network
- How does the sender's window size get chosen?
  - Must be less than receiver's advertised buffer size
  - Try to match the rate of sending packets with the rate that the slowest link can accommodate
  - Sender uses an adaptive algorithm to decide size of N
    - » Goal: fill network between sender and receiver
    - » Basic technique: slowly increase size of window until acknowledgements start being delayed/lost
- TCP solution: "slow start" (start sending slowly)
  - If no timeout, slowly increase window size (throughput) by 1 for each ack received
  - Timeout ⇒ congestion, so cut window size in half
  - "*Additive Increase, Multiplicative Decrease*"

Open Connection: 3-Way Handshaking

- Goal: agree on a set of parameters, i.e., the start sequence number for each side
  - Starting sequence number (first byte in stream)
  - Must be unique!
    - » If it is possible to predict sequence numbers, might be possible for attacker to hijack TCP connection
- Some ways of choosing an initial sequence number:
  - Time to live: each packet has a deadline.
    - » If not delivered in X seconds, then is dropped
    - » Thus, can re-use sequence numbers if wait for all packets in flight to be delivered or to expire
  - Epoch #: uniquely identifies *which* set of sequence numbers are currently being used
    - » Epoch # stored on disk, Put in every message
    - » Epoch # incremented on crash and/or when run out of sequence #
  - Pseudo-random increment to previous sequence number
    - » Used by several protocol implementations

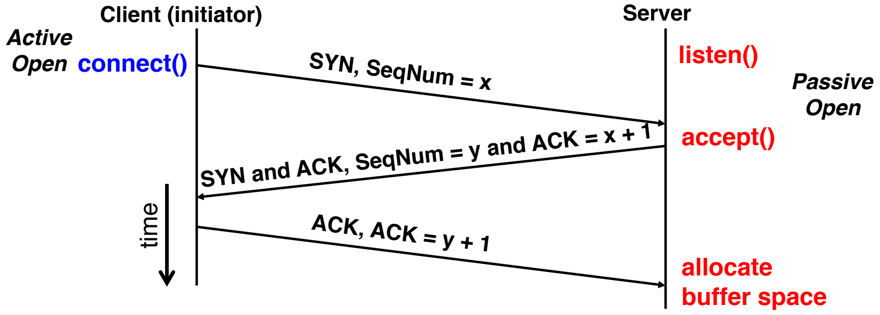
Open Connection: 3-Way Handshaking

- Server waits for new connection calling listen()
- Sender call connect() passing socket which contains server's IP address and port number
  - OS sends a special packet (SYN) containing a proposal for first sequence number, x

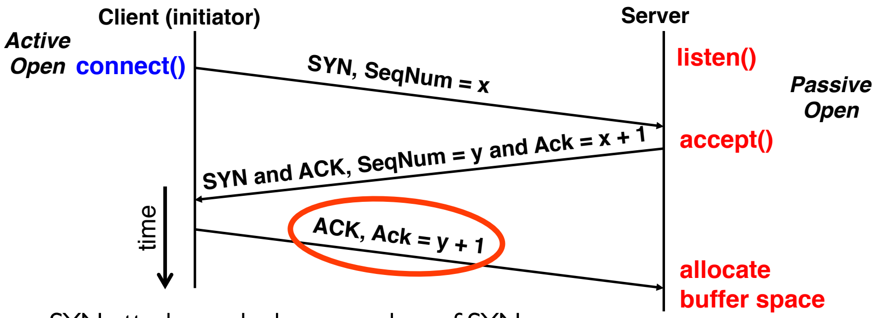


Open Connection: 3-Way Handshaking

- If it has enough resources, server calls accept() to accept connection, and sends back a SYN ACK packet containing
  - Client's sequence number incremented by one, (x + 1)
    - » Why is this needed?
  - A sequence number proposal, y, for first byte server will send



Denial of Service Vulnerability



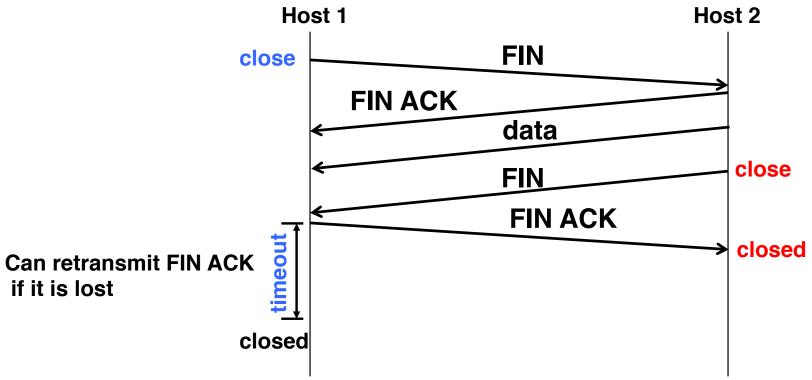
- SYN attack: send a huge number of SYN messages
  - Causes victim to commit resources (768 byte TCP/IP data structure)
- Alternatives: Do not commit resources until receive final ACK
  - SYN Cache: when SYN received, put small entry into cache and send SYN/ACK, If receive ACK, then put into listening socket
  - SYN Cookie: when SYN received, encode connection info into sequence number/other TCP header blocks, decode on ACK

3-Way Handshaking (cont'd)

- Three-way handshake adds 1 RTT delay
- Why do it this way?
  - Congestion control: SYN (40 byte) acts as cheap probe
  - Protects against delayed packets from a previous connection (would confuse receiver)

Close Connection

- Goal: both sides agree to close the connection
- 4-way connection tear down

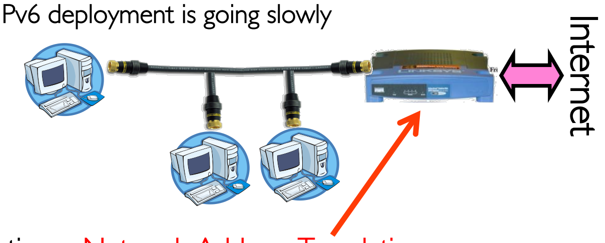


Recall: Using TCP Sockets

- Socket: an abstraction of a network I/O queue
  - Embodies one side of a communication channel
    - » Same interface regardless of location of other end
    - » Could be local machine (called "UNIX socket") or remote machine (called "network socket")
  - First introduced in 4.2 BSD UNIX: big innovation at time
    - » Now most operating systems provide some notion of socket
- Using Sockets for Client-Server (C/C++ interface):
  - On server: set up "server-socket"
    - » Create socket, Bind to protocol (TCP), local address, port
    - » Call listen(): tells server socket to accept incoming requests
    - » Perform multiple accept() calls on socket to accept incoming connection request
    - » Each successful accept() returns a new socket for a new connection; can pass this off to handler thread
  - On client:
    - » Create socket, Bind to protocol (TCP), remote address, port
    - » Perform connect() on socket to make connection
    - » If connect() successful, have socket connected to server

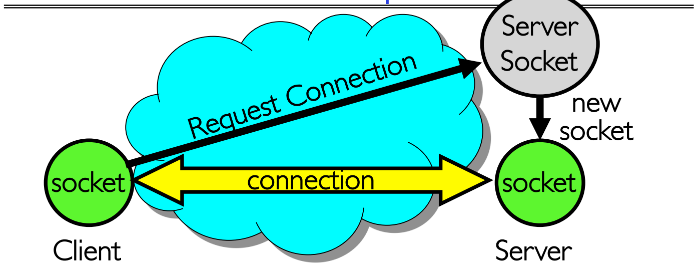
Network Address Translation (NAT)

- Problem:
  - IPv4 supports 2^{32} hosts, but allocation classes mean not all addresses can practically be used
  - Stanford, MIT each have class A allocation: 16,777,216 addresses!
  - IPv6 deployment is going slowly



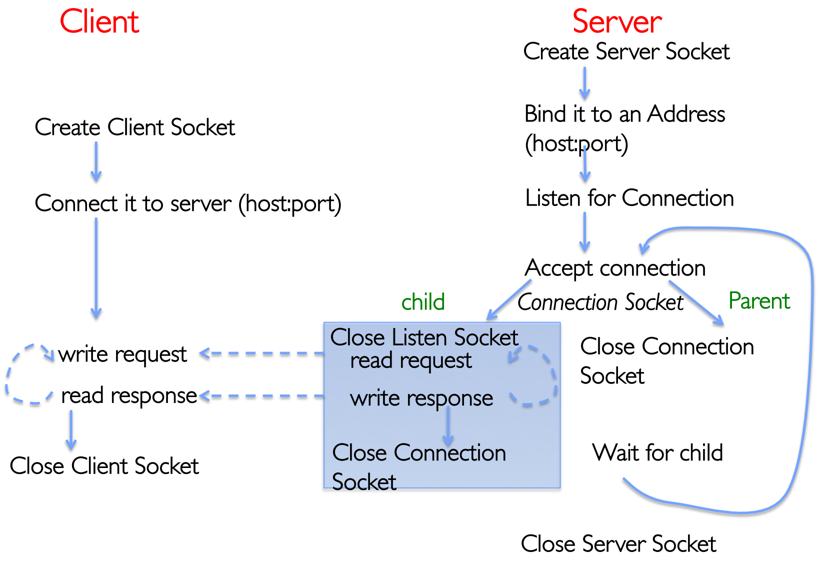
- Solution – Network Address Translation
  - Local subnet (uses non-routable IP addresses) ⇒ External IP
  - Router/firewall replaces local IP address/port combinations with external IP address/new port combinations
  - Router/firewall maintains translation table of current connections

Recall: Socket Setup over TCP/IP



- Things to remember:
  - Connection involves 5 values: [Client Addr, Client Port, Server Addr, Server Port, Protocol]
  - Often, Client Port "randomly" assigned
  - Server Port often "well known"
    - » 80 (web), 443 (secure web), 25 (sendmail), etc
    - » Well-known ports from 0—1023
- Network Address Translation (NAT) allows many internal connections (and/or hosts) with a single external IP address

Recall: Sockets With Protection/Parallelism



Client

Create Client Socket

Connect it to server (host:port)

write request

read response

Close Client Socket

Server

Create Server Socket

Bind it to an Address (host:port)

Listen for Connection

Accept connection

Connection Socket

child

Close Listen Socket

read request

write response

Close Connection Socket

Parent

Close Connection Socket

Wait for child

Close Server Socket

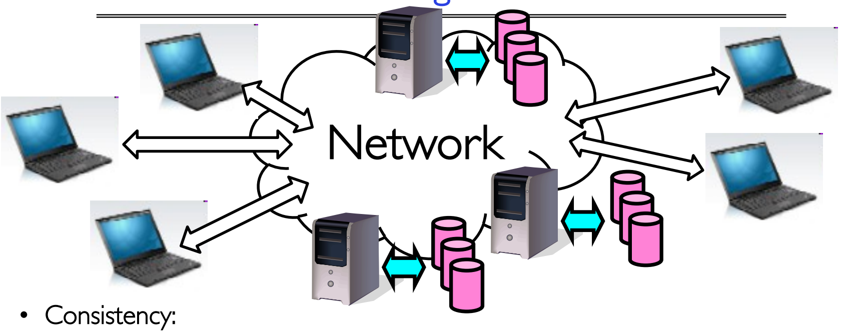
Recall: Client Protocol

```
char *hostname;
int sockfd, portno;
struct sockaddr_in serv_addr;
struct hostent *server;

server = buildServerAddr(&serv_addr, hostname, portno);

/* Create a TCP socket */
sockfd = socket(AF_INET, SOCK_STREAM, 0)

/* Connect to server on port */
connect(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)
printf("Connected to %s:%d\n",server->h_name, portno);

/* Carry out Client-Server protocol */
client(sockfd);

/* Clean up on termination */
close(sockfd);
```

Recall: Server Protocol (v2)

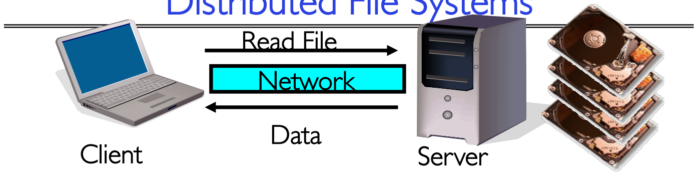
```
/* Create Socket to receive requests*/
lstnsockfd = socket(AF_INET, SOCK_STREAM, 0);
/* Bind socket to port */
bind(lstnsockfd, (struct sockaddr)&serv_addr,sizeof(serv_addr));
/* Set up socket to listen for incoming connections */
listen(lstnsockfd, MAXQUEUE);
while (1) {
    consockfd = accept(lstnsockfd, (struct sockaddr *) &cli_addr,
                                   &clilen);
    cpid = fork();              /* new process for connection */
    if (cpid > 0) {             /* parent process */
      close(consockfd);
      tcpid = wait(&cstatus);
    } else if (cpid == 0) {      /* child process */
      close(lstnsockfd);        /* let go of listen socket */

      server(consockfd);

      close(consockfd);
      exit(EXIT_SUCCESS);         /* exit child normally */
    }
  }
close(lstnsockfd);
```

Network-Attached Storage and the CAP Theorem

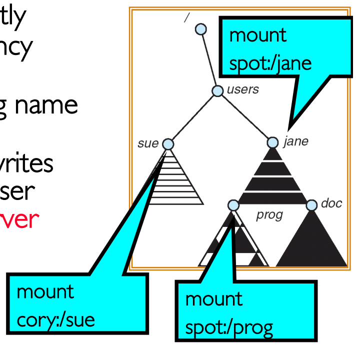


- Consistency:
  - Changes appear to everyone in the same serial order
- Availability:
  - Can get a result at any time
- Partition-Tolerance
  - System continues to work even when network becomes partitioned
- Consistency, Availability, Partition-Tolerance (CAP) Theorem: Cannot have all three at same time
  - Otherwise known as "Brewer's Theorem"

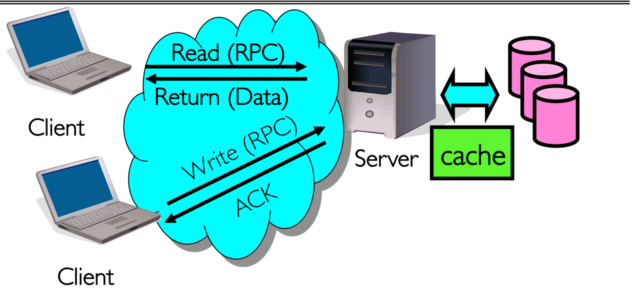
Distributed File Systems



- Transparent access to files stored on a remote disk
- Naming choices (always an issue):
  - *Hostname*:localname: Name files explicitly
    - » No location or migration transparency
  - *Mounting* of remote file systems
    - » Mounts remote file system by giving name and local mount point
    - » Transparent to user: all reads and writes look like local reads and writes to user e.g. /users/sue/foo→/sue/foo on server
  - *A single, global name space:* every file in the world has unique name
    - » Location Transparency: servers/files can move without involving user

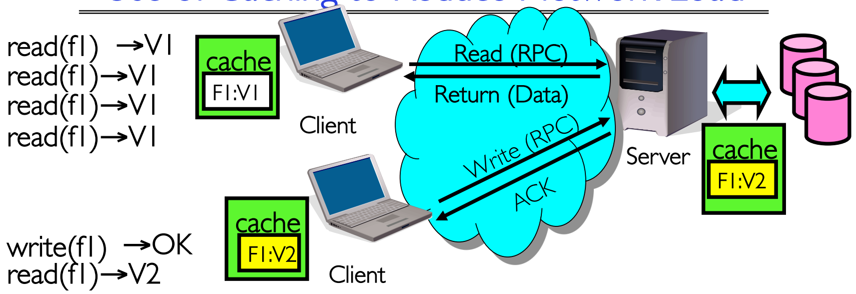


Simple Distributed File System



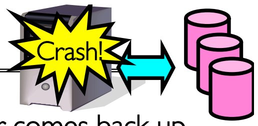
- Remote Disk: Reads and writes forwarded to server
  - Use Remote Procedure Calls (RPC) to translate file system calls into remote requests
  - No local caching/can be caching at server-side
- Advantage: Server provides completely consistent view of file system to multiple clients
- Problems? Performance!
  - Going over network is slower than going to local memory
  - Lots of network traffic/not well pipelined
  - Server can be a bottleneck

Use of Caching to Reduce Network Load



- Idea: Use caching to reduce network load
  - In practice: use buffer cache at source and destination
- Advantage: if open/read/write/close can be done locally, don't need to do any network traffic…fast!
- Problems:
  - Failure:
    - » Client caches have data not committed at server
  - Cache consistency!
    - » Client caches not consistent with server/each other

Failures



- What if server crashes? Can client wait until server comes back up and continue as before?
  - Any data in server memory but not on disk can be lost
  - Shared state across RPC: What if server crashes after seek? Then, when client does "read", it will fail
  - Message retries: suppose server crashes after it does UNIX "`rm foo`", but before acknowledgment?
    - » Message system will retry: send it again
    - » How does it know not to delete it again? (could solve with two-phase commit protocol, but NFS takes a more ad hoc approach)
- Stateless protocol: A protocol in which all information required to process a request is passed with request
  - Server keeps no state about client, except as hints to help improve performance (e.g., a cache)
  - Thus, if server crashes and restarted, requests can continue where left off (in many cases)
- What if client crashes?
  - Might lose modified data in client cache

Network File System (NFS)

- Three Layers for NFS system
  - UNIX filesystem API: open, read, write, close calls + file descriptors
  - VFS layer: distinguishes local from remote files
    - » Calls the NFS protocol procedures for remote requests
  - NFS service layer: bottom layer of the architecture
    - » Implements the NFS protocol
- NFS Protocol: RPC for file operations on server
  - Reading/searching a directory
  - Manipulating links and directories
  - Accessing file attributes/reading and writing files
- Write-through caching: Modified data committed to server's disk before results are returned to the client
  - Lose some of the advantages of caching
  - Time to perform write() can be long
  - Need some mechanism for readers to eventually notice changes! (more on this later)

NFS Continued

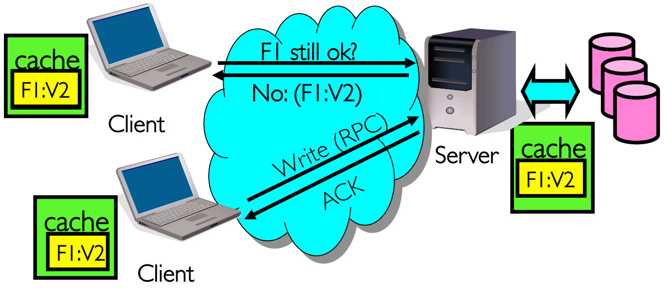
- NFS servers are stateless; each request provides all arguments require for execution
  - E.g. reads include information for entire operation, such as **`ReadAt(inumber,position)`**, not **`Read(openfile)`**
  - No need to perform network open() or close() on file – each operation stands on its own
- Idempotent: Performing requests multiple times has same effect as performing it exactly once
  - Example: Server crashes between disk I/O and message send, client resend read, server does operation again
  - Example: Read and write file blocks: just re-read or re-write file block – no side effects
  - Example: What about "remove"? NFS does operation twice and second time returns an advisory error

NFS Failure Model

- Transparent to client system
- Is this a good idea? What if you are in the middle of reading a file and server crashes?
- Options – NFS provides both choices:
  - Hang until server comes back up (next week?)
  - Return an error (of course, most applications don't know they are talking over a network)

NFS Cache Consistency

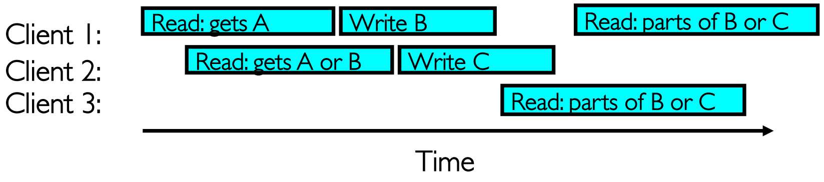
- NFS protocol: weak consistency
  - Client polls server periodically to check for changes
    - » Polls server for changes to data in last 3-30 seconds (tunable parameter)
    - » Thus, when file is changed on one client, server is notified, but other clients use old version of file until timeout



  - What if multiple clients write to same file?
    - » In NFS, can get either version (or parts of both)
    - » Completely arbitrary

Sequential Ordering Constraints

- What sort of cache coherence might we expect?
  - One CPU changes file, and before it's done, another CPU reads file
- Example: Start with file contents = "A"



Client 1: Read: gets A | Write B | Read: parts of B or C

Client 2: Read: gets A or B | Write C

Client 3: Read: parts of B or C

Time

- What would we actually want?
  - Assume we want distributed system to behave exactly the same as if all processes are running on single system
    - » If read finishes before write starts, get old copy
    - » If read starts after write finishes, get new copy
    - » Otherwise, get either new or old copy
  - For NFS:
    - » If read starts more than 30 seconds after write, get new copy; otherwise, could get partial update

NFS Pros and Cons

- NFS Pros:
  - Simple, Highly portable
- NFS Cons:
  - Sometimes inconsistent!
  - Doesn't scale to large numbers of clients
    - » Must keep checking to see if caches out of date
    - » Server becomes bottleneck due to polling traffic

Summary

- Internet Protocol (IP)
  - Used to route messages through routes across globe
  - 32-bit addresses, 16-bit ports
- DNS: System for mapping from names⇒IP addresses
  - Hierarchical mapping from authoritative domains
  - Recent flaws discovered
- Datagram: a self-contained message whose arrival, arrival time, and content are not guaranteed
- Ordered messages:
  - Use sequence numbers and reorder at destination
- Reliable messages:
  - Use Acknowledgements
- TCP: Reliable byte stream between two processes on different machines over Internet (read, write, flush)
  - Uses window-based acknowledgement protocol
  - Congestion-avoidance dynamically adapts sender window to account for congestion in network