CS162
Operating Systems and Systems Programming
Lecture 21

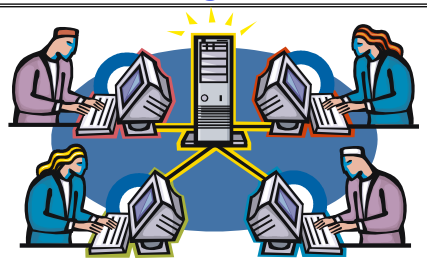
Distributed Decision Making, TCP/IP and Sockets

November 9th, 2016

Prof. Anthony D. Joseph

http://cs162.eecs.Berkeley.edu

Networking Definitions

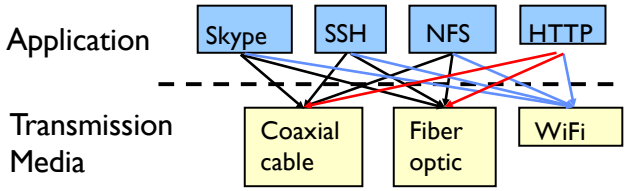


- Network: physical connection that allows two computers to communicate
- Packet: unit of transfer, sequence of bits carried over the network
  - Network carries packets from one CPU to another
  - Destination gets interrupt when packet arrives
- Protocol: agreement between two parties as to how information is to be transmitted

Recall: What Is A Protocol?

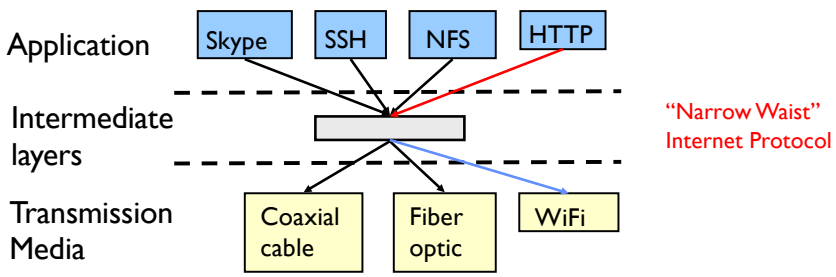
- A protocol is an agreement on how to communicate
- Includes
  - Syntax: how a communication is specified & structured
    - » Format, order messages are sent and received
  - Semantics: what a communication means
    - » Actions taken when transmitting, receiving, or when a timer expires
- Described formally by a state machine
  - Often represented as a message transaction diagram

Global Communication: The Problem



- Many different applications
  - email, web, P2P, etc.
- Many different network styles and technologies
  - Wireless vs. wired vs. optical, etc.
- How do we organize this mess?
  - Re-implement every application for every technology?
- No! But how does the Internet design avoid this?

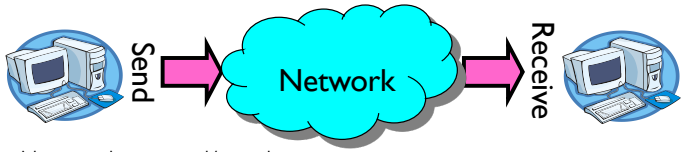
Solution: Intermediate Layers



- Introduce intermediate layers that provide set of abstractions for various network functionality & technologies
  - A new app/media implemented only once
  - Variation on "add another level of indirection"
- Goal: Reliable communication channels on which to build distributed applications

Distributed Applications

- How do you actually program a distributed application?
  - Need to synchronize multiple threads, running on different machines
    - » No shared memory, so cannot use test&set



  - One Abstraction: send/receive messages
    - » Already atomic: no receiver gets portion of a message and two receivers cannot get same message
- Interface:
  - Mailbox (`mbox`): temporary holding area for messages
    - » Includes both destination location and queue
  - `Send(message,mbox)`
    - » Send message to remote mailbox identified by `mbox`
  - `Receive(buffer,mbox)`
    - » Wait until `mbox` has message, copy into buffer, and return
    - » If threads sleeping on this `mbox`, wake up one of them

Using Messages: Send/Receive behavior

- When should `send(message, mbox)` return?
  - When receiver gets message? (i.e. ack received)
  - When message is safely buffered on destination?
  - Right away, if message is buffered on source node?
- Actually two questions here:
  - When can the sender be sure that receiver actually received the message?
  - When can sender reuse the memory containing message?
- Mailbox provides 1-way communication from T1 → T2
  - T1 → buffer → T2
  - Very similar to producer/consumer
    - » Send = V, Receive = P
    - » However, can't tell if sender/receiver is local or not!

Messaging for Producer-Consumer Style

- Using send/receive for producer-consumer style:

```
Producer:
   int msg1[1000];
   while(1) {
      prepare message;
      send(msg1, mbox);
   }
Consumer:
   int buffer[1000];
   while(1) {
      receive(buffer, mbox);
      process message;
   }
```

Send Message

Receive Message

- No need for producer/consumer to keep track of space in mailbox: handled by send/receive
  - Next time: will discuss fact that this is one of the roles of the window in TCP: window is size of buffer on far end
  - Restricts sender to forward only what will fit in buffer

Messaging for Request/Response Communication

- What about two-way communication?
  - Request/Response
    - » Read a file stored on a remote machine
    - » Request a web page from a remote web server
  - Also called: client-server
    - » Client ≡ requester, Server ≡ responder
    - » Server provides "service" (file storage) to the client
- Example: File service

```
Client: (requesting the file)
   char response[1000];

   send("read rutabaga", server_mbox);
   receive(response, client_mbox);

Server: (responding with the file)
   char command[1000], answer[1000];

   receive(command, server_mbox);
   decode command;
   read file into answer;
   send(answer, client_mbox);
```

Request File

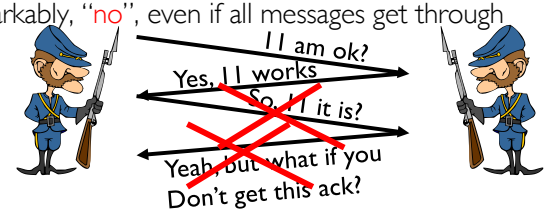
Get Response

Receive Request

Send Response

General's Paradox

- General's paradox:
  - Constraints of problem:
    - » Two generals, on separate mountains
    - » Can only communicate via messengers
    - » Messengers can be captured
  - Problem: need to coordinate attack
    - » If they attack at different times, they all die
    - » If they attack at same time, they win
  - Named after Custer, who died at Little Big Horn because he arrived a couple of days too early
- Can messages over an unreliable network be used to guarantee two entities do something simultaneously?
  - Remarkably, "no", even if all messages get through



  - No way to be sure last message gets through!

Two-Phase Commit

- Since we can't solve the General's Paradox (i.e. simultaneous action), let's solve a related problem
  - Distributed transaction: Two or more machines agree to do something, or not do it, atomically
- Two-Phase Commit protocol:
  - Persistent stable log on each machine: keep track of whether commit has happened
    - » If a machine crashes, when it wakes up it first checks its log to recover state of world at time of crash
  - Prepare Phase:
    - » The global coordinator requests that all participants will promise to commit or rollback the transaction
    - » Participants record promise in log, then acknowledge
    - » If anyone votes to abort, coordinator writes `"Abort"` in its log and tells everyone to abort; each records `"Abort"` in log
  - Commit Phase:
    - » After all participants respond that they are prepared, then the coordinator writes `"Commit"` to its log
    - » Then asks all nodes to commit; they respond with ACK
    - » After receive ACKs, coordinator writes `"Got Commit"` to log
  - Log can be used to complete this process such that all machines either commit or don't commit

2PC Algorithm

- Developed by Turing award winner Jim Gray (first Berkeley CS PhD, 1969)
- One coordinator
- N workers (replicas)
- High level algorithm description:
  - Coordinator asks all workers if they can commit
  - If all workers reply "VOTE-COMMIT", then coordinator broadcasts "GLOBAL-COMMIT"

    Otherwise coordinator broadcasts "GLOBAL-ABORT"
  - Workers obey the GLOBAL messages
- Use a persistent, stable log on each machine to keep track of what you are doing
  - If a machine crashes, when it wakes up it first checks its log to recover state of world at time of crash

Detailed Algorithm

Coordinator Algorithm

Coordinator sends **VOTE-REQ** to all workers

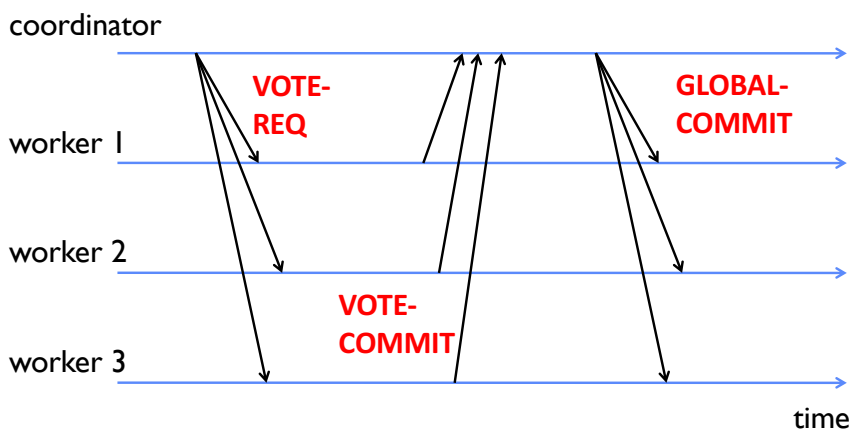
- If receive **VOTE-COMMIT** from all N workers, send **GLOBAL-COMMIT** to all workers
- If doesn't receive **VOTE-COMMIT** from all N workers, send **GLOBAL-ABORT** to all workers

Worker Algorithm

- Wait for **VOTE-REQ** from coordinator
- If ready, send **VOTE-COMMIT** to coordinator
- If not ready, send **VOTE-ABORT** to coordinator
  - And immediately abort

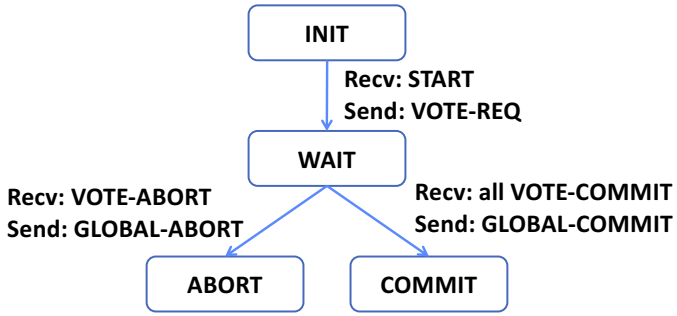
- If receive **GLOBAL-COMMIT** then commit
- If receive **GLOBAL-ABORT** then abort

Failure Free Example Execution

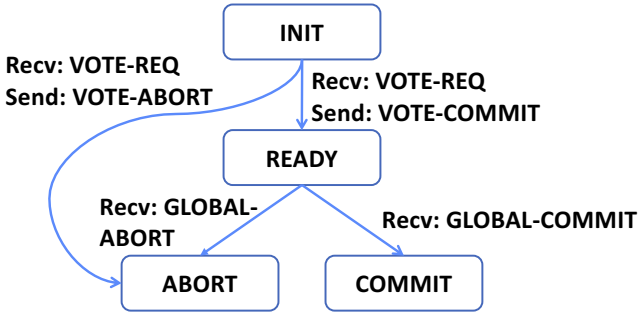


State Machine of Coordinator

- Coordinator implements simple state machine:

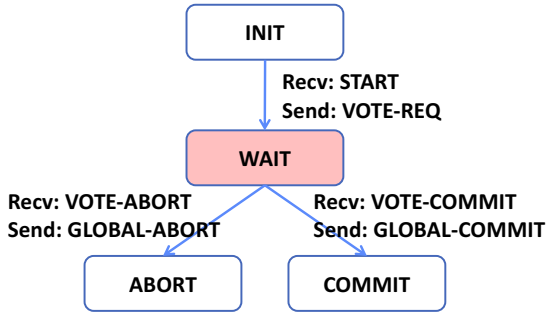


State Machine of Workers

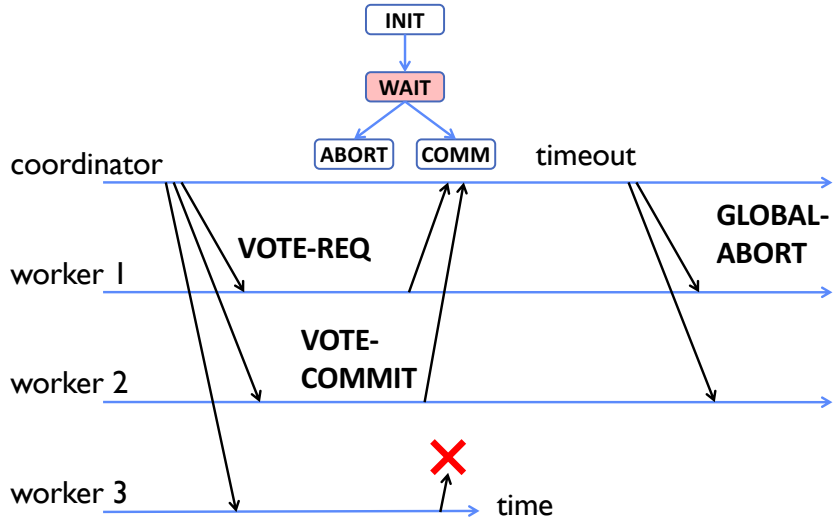


Dealing with Worker Failures

- Failure only affects states in which the coordinator is waiting for messages
- Coordinator only waits for votes in "WAIT" state
- In WAIT, if doesn't receive N votes, it times out and sends GLOBAL-ABORT

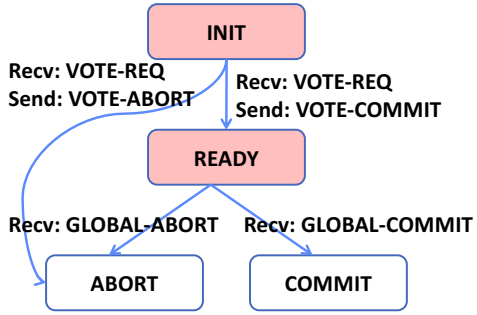


Example of Worker Failure

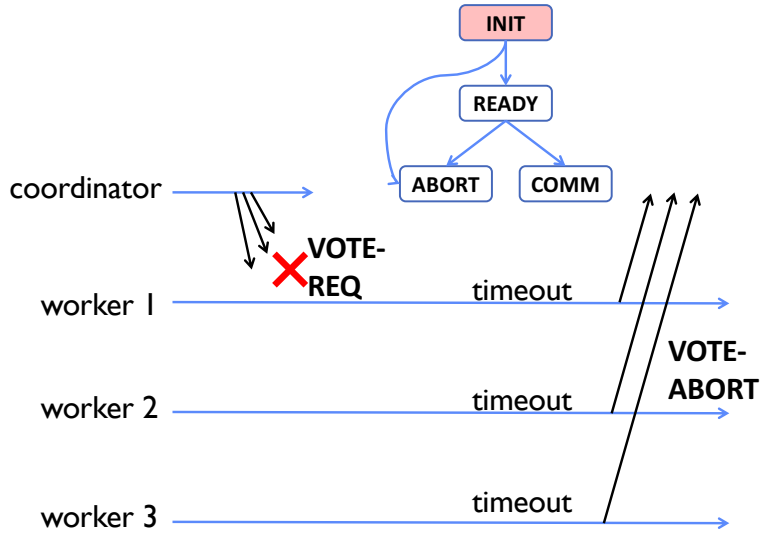


Dealing with Coordinator Failure

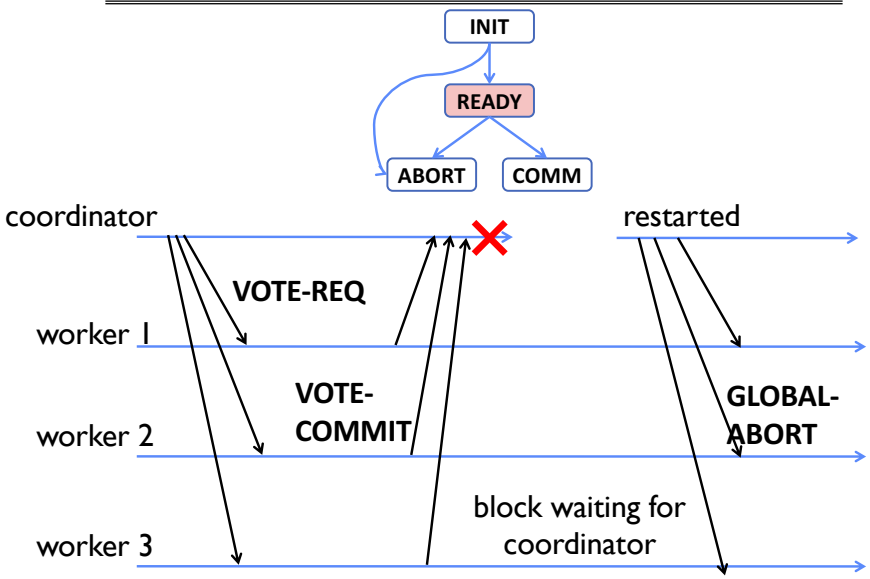
- Worker waits for VOTE-REQ in INIT
  - Worker can time out and abort (coordinator handles it)
- Worker waits for GLOBAL-* message in READY
  - If coordinator fails, workers must **BLOCK** waiting for coordinator to recover and send GLOBAL_* message



Example of Coordinator Failure #1



Example of Coordinator Failure #2

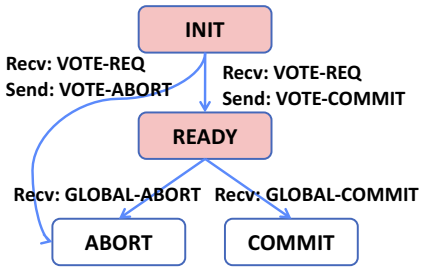


Durability

- All nodes use stable storage to store current state
  - stable storage is non-volatile storage (e.g. backed by disk) that guarantees atomic writes.
- Upon recovery, it can restore state and resume:
  - Coordinator aborts in INIT, WAIT, or ABORT
  - Coordinator commits in COMMIT
  - Worker aborts in INIT, ABORT
  - Worker commits in COMMIT
  - Worker asks Coordinator in READY

Blocking for Coordinator to Recover

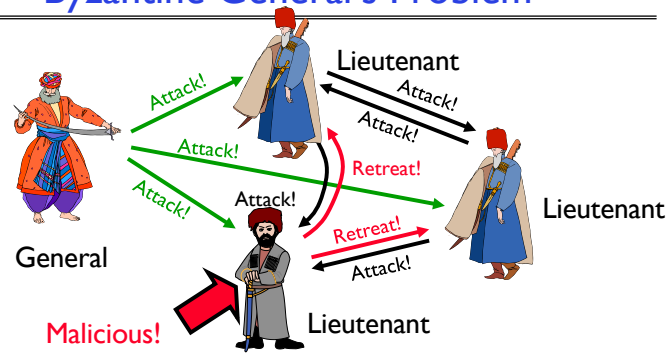
- A worker waiting for global decision can ask fellow workers about their state
  - If another worker is in ABORT or COMMIT state then coordinator must have sent GLOBAL-*
    - » Thus, worker can safely abort or commit, respectively
  - If another worker is still in INIT state then both workers can decide to abort
  - If all workers are in ready, need to **BLOCK** (don't know if coordinator wanted to abort or commit)



Distributed Decision Making Discussion

- Why is distributed decision making desirable?
  - Fault Tolerance!
  - A group of machines can come to a decision even if one or more of them fail during the process
    - » Simple failure mode called "failstop" (different modes later)
  - After decision made, result recorded in multiple places
- Undesirable feature of Two-Phase Commit: Blocking
  - One machine can be stalled until another site recovers:
    - » Site B writes `"prepared to commit"` record to its log, sends a `"yes"` vote to the coordinator (site A) and crashes
    - » Site A crashes
    - » Site B wakes up, check its log, and realizes that it has voted `"yes"` on the update. It sends a message to site A asking what happened. At this point, B cannot decide to abort, because update may have committed
    - » B is blocked until A comes back
  - A blocked site holds resources (locks on updated items, pages pinned in memory, etc) until learns fate of update
- PAXOS: An alternative used by Google and others that does not have this blocking problem
- What happens if one or more of the nodes is malicious?
  - Malicious: attempting to compromise the decision making

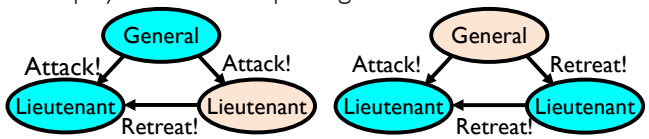
Byzantine General's Problem



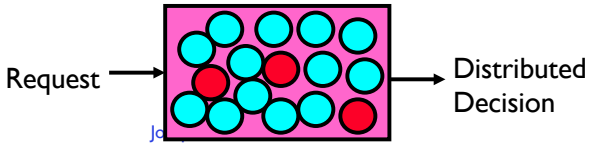
- Byazantine General's Problem (n players):
  - One General and n-1 Lieutenants
  - Some number of these (*f*) can be insane or malicious
- The commanding general must send an order to his n-1 lieutenants such that the following Integrity Constraints apply:
  - IC1: All loyal lieutenants obey the same order
  - IC2: If the commanding general is loyal, then all loyal lieutenants obey the order he sends

Byzantine General's Problem (con't)

- Impossibility Results:
  - Cannot solve Byzantine General's Problem with $n=3$ because one malicious player can mess up things



  - With f faults, need $n > 3f$ to solve problem
- Various algorithms exist to solve problem
  - Original algorithm has #messages exponential in n
  - Newer algorithms have message complexity $O(n^2)$
    - » One from MIT, for instance (Castro and Liskov, 1999)
- Use of BFT (Byzantine Fault Tolerance) algorithm
  - Allow multiple machines to make a coordinated decision even if some subset of them ($< n/3$) are malicious



Administrivia

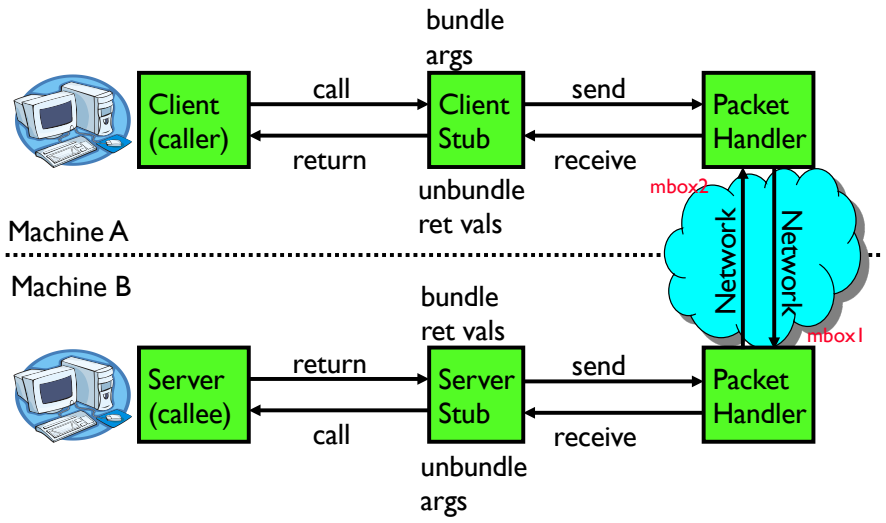
- Project 3 – Design Doc due Monday 11/14

BREAK

Remote Procedure Call

- Raw messaging is a bit too low-level for programming
  - Must wrap up information into message at source
  - Must decide what to do with message at destination
  - May need to sit and wait for multiple messages to arrive
- Another option: Remote Procedure Call (RPC)
  - Calls a procedure on a remote machine
  - Client calls:
    `remoteFileSystem→Read("rutabaga");`
  - Translated automatically into call on server:
    `fileSys→Read("rutabaga");`
- Implementation:
  - Request-response message passing (under covers!)
  - "Stub" provides glue on client/server
    - » Client stub is responsible for "marshalling" arguments and "unmarshalling" the return values
    - » Server-side stub is responsible for "unmarshalling" arguments and "marshalling" the return values.
- Marshalling involves (depending on system)
  - Converting values to a canonical form, serializing objects, copying arguments passed by reference, etc.

RPC Information Flow



RPC Details

- Equivalence with regular procedure call
  - Parameters ⇔ Request Message
  - Result ⇔ Reply message
  - Name of Procedure: Passed in request message
  - Return Address: mbox2 (client return mail box)
- Stub generator: Compiler that generates stubs
  - Input: interface definitions in an "interface definition language (IDL)"
    - » Contains, among other things, types of arguments/return
  - Output: stub code in the appropriate source language
    - » Code for client to pack message, send it off, wait for result, unpack result and return to caller
    - » Code for server to unpack message, call procedure, pack results, send them off
- Cross-platform issues:
  - What if client/server machines are different architectures or in different languages?
    - » Convert everything to/from some canonical form
    - » Tag every item with an indication of how it is encoded (avoids unnecessary conversions)

RPC Details (continued)

- How does client know which mbox to send to?
  - Need to translate name of remote service into network endpoint (Remote machine, port, possibly other info)
  - Binding: the process of converting a user-visible name into a network endpoint
    - » This is another word for "naming" at network level
    - » Static: fixed at compile time
    - » Dynamic: performed at runtime
- Dynamic Binding
  - Most RPC systems use dynamic binding via name service
    - » Name service provides dynamic translation of service → mbox
  - Why dynamic binding?
    - » Access control: check who is permitted to access service
    - » Fail-over: If server fails, use a different one
- What if there are multiple servers?
  - Could give flexibility at binding time
    - » Choose unloaded server for each new client
  - Could provide same mbox (router level redirect)
    - » Choose unloaded server for each new request
    - » Only works if no state carried from one call to next
- What if multiple clients?
  - Pass pointer to client-specific return mbox in request

Problems with RPC

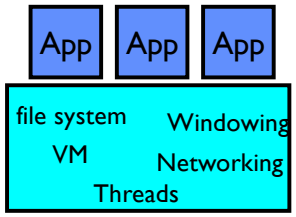
- Non-Atomic failures
  - Different failure modes in distributed system than on a single machine
  - Consider many different types of failures
    - » User-level bug causes address space to crash
    - » Machine failure, kernel bug causes all processes on same machine to fail
    - » Some machine is compromised by malicious party
  - Before RPC: whole system would crash/die
  - After RPC: One machine crashes/compromised while others keep working
  - Can easily result in inconsistent view of the world
    - » Did my cached data get written back or not?
    - » Did server do what I requested or not?
  - Answer? Distributed transactions/Byzantine Commit
- Performance
  - Cost of Procedure call « same-machine RPC « network RPC
  - Means programmers must be aware that RPC is not free
    - » Caching can help, but may make failure handling complex

Cross-Domain Communication/Location Transparency

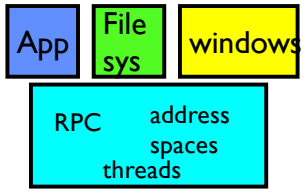
- How do address spaces communicate with one another?
  - Shared Memory with Semaphores, monitors, etc…
  - File System
  - Pipes (1-way communication)
  - "Remote" procedure call (2-way communication)
- RPC's can be used to communicate between address spaces on different machines or the same machine
  - Services can be run wherever it's most appropriate
  - Access to local and remote services looks the same
- Examples of modern RPC systems:
  - CORBA (Common Object Request Broker Architecture)
  - DCOM (Distributed COM)
  - RMI (Java Remote Method Invocation)

Microkernel operating systems

- Example: split kernel into application-level servers.
  - File system looks remote, even though on same machine



Monolithic Structure

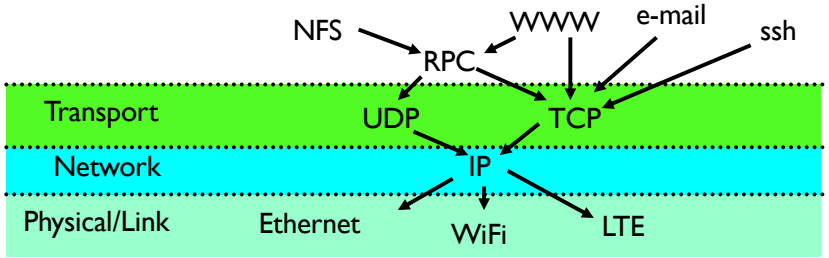


Microkernel Structure

- Why split the OS into separate domains?
  - Fault isolation: bugs are more isolated (build a firewall)
  - Enforces modularity: allows incremental upgrades of pieces of software (client or server)
  - Location transparent: service can be local or remote
    - » For example in the X windowing system:
      - Each X client can be on a separate machine from X server
      - Neither has to run on the machine with the frame buffer

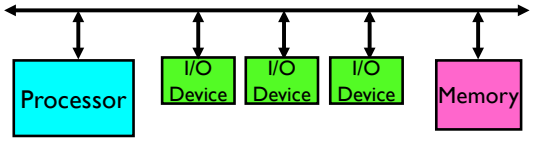
Network Protocols

- 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:

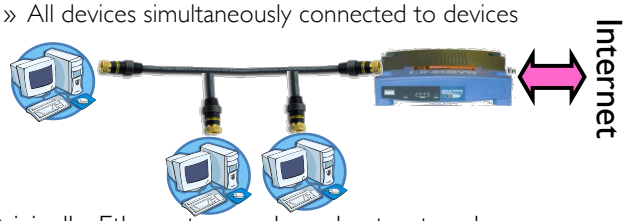


Broadcast Networks

- Broadcast Network: Shared Communication Medium

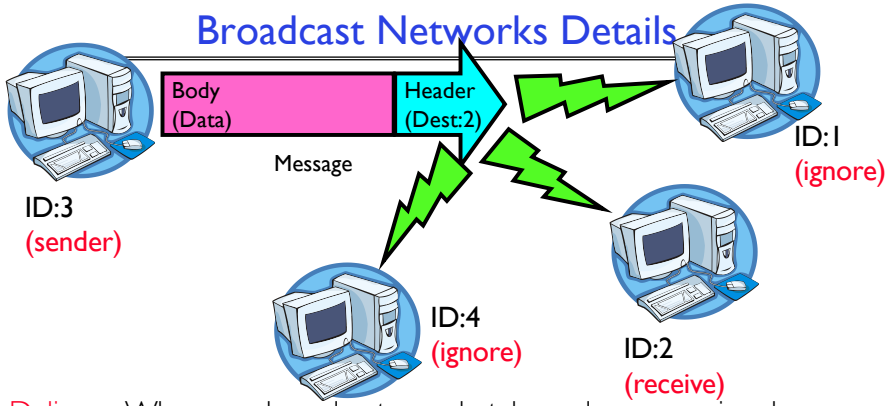


  – Shared Medium can be a set of wires
    » Inside a computer, this is called a bus
    » All devices simultaneously connected to devices



  – Originally, Ethernet was a broadcast network
    » All computers on local subnet connected to one another
  – More examples (wireless: medium is air): cellular phones (GSM, CDMA, and LTE), WiFi

Broadcast Networks Details

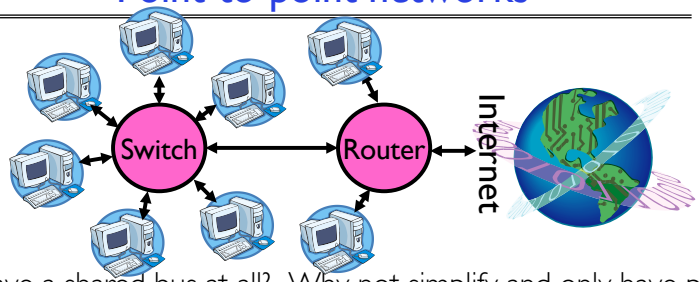


- Delivery: When you broadcast a packet, how does a receiver know who it is for? (packet goes to everyone!)
  – Put header on front of packet: [Destination | Packet]
  – Everyone gets packet, discards if not the target
  – In Ethernet and WiFi, this check is done in hardware
    » No OS interrupt if not for particular destination
  – This is layering: we're going to build complex network protocols by layering on top of the packet

Carrier Sense, Multiple Access/Collision Detection

- Ethernet (early 80's): first practical local area network
  – It is the most common LAN for UNIX, PC, and Mac
  – Use wire instead of radio, but still broadcast medium
- Key advance was in arbitration called CSMA/CD: Carrier sense, multiple access/collision detection
  – Carrier Sense: don't send unless idle
    » Don't mess up communications already in process
  – Collision Detect: sender checks if packet trampled.
    » If so, abort, wait, and retry.
  – Backoff Scheme: Choose wait time before trying again
- How long to wait after trying to send and failing?
  – What if everyone waits the same length of time? Then, they all collide again at some time!
  – Must find way to break up shared behavior with nothing more than shared communication channel
- Adaptive randomized waiting strategy:
  – Adaptive and Random: First time, pick random wait time with some initial mean. If collide again, pick random value from bigger mean wait time. Etc.
  – Randomness is important to decouple colliding senders
  – Scheme figures out how many people are trying to send!

Point-to-point networks



- Why have a shared bus at all? Why not simplify and only have point-to-point links + routers/switches?
  – Originally wasn't cost-effective
  – Now, easy to make high-speed switches and routers that can forward packets from a sender to a receiver
- Point-to-point network: a network in which every physical wire is connected to only two computers
- Switch: a bridge that transforms a shared-bus (broadcast) configuration into a point-to-point network
- Router: a device that acts as a junction between two networks to transfer data packets among them

The Internet Protocol: "IP"

- The Internet is a large network of computers spread across the globe
  - According to the Internet Systems Consortium, there were over 1.05 Billion computers as of July 2016
  - In principle, every host can speak with every other one under the right circumstances
- IP Packet: a network packet on the internet
- IP Address: a 32-bit integer used as the destination of an IP packet
  - Often written as four dot-separated integers, with each integer from 0—255 (thus representing 8x4=32 bits)
  - Example CS file server is: `169.229.60.83` ≡ `0xA9E53C53`
- Internet Host: a computer connected to the Internet
  - Host has one or more IP addresses used for routing
    - » Some of these may be private and unavailable for routing
  - Not every computer has a unique IP address
    - » Groups of machines may share a single IP address
    - » In this case, machines have private addresses behind a "Network Address Translation" (NAT) gateway

Address Subnets

- Subnet: A network connecting a set of hosts with related destination addresses
- With IP, all the addresses in subnet are related by a prefix of bits
  - Mask: The number of matching prefix bits
    - » Expressed as a single value (e.g., 24) or a set of ones in a 32-bit value (e.g., `255.255.255.0`)
- A subnet is identified by 32-bit value, with the bits which differ set to zero, followed by a slash and a mask
  - Example: `128.32.131.0/24` designates a subnet in which all the addresses look like `128.32.131.XX`
  - Same subnet: `128.32.131.0/255.255.255.0`
- Difference between subnet and complete network range
  - Subnet is always a subset of address range
  - Once, subnet meant single physical broadcast wire; now, less clear exactly what it means (virtualized by switches)

Address Ranges in IP

- IP address space divided into prefix-delimited ranges:
  - Class A: `NN.0.0.0/8`
    - » `NN` is 1–126 (126 of these networks)
    - » 16,777,214 IP addresses per network
    - » `10.xx.yy.zz` is private
    - » `127.xx.yy.zz` is loopback
  - Class B: `NN.MM.0.0/16`
    - » `NN` is 128–191, `MM` is 0-255 (16,384 of these networks)
    - » 65,534 IP addresses per network
    - » `172.[16-31].xx.yy` are private
  - Class C: `NN.MM.LL.0/24`
    - » `NN` is 192–223, `MM` and LL 0-255 (2,097,151 of these networks)
    - » 254 IP addresses per networks
    - » `192.168.xx.yy` are private
- Address ranges are often owned by organizations
  - Can be further divided into subnets

Summary (1/2)

- Protocol: Agreement between two parties as to how information is to be transmitted
- Two-phase commit: distributed decision making
  - First, make sure everyone guarantees they will commit if asked (prepare)
  - Next, ask everyone to commit
- Byzantine General's Problem: distributed decision making with malicious failures
  - One general, n-1 lieutenants: some number of them may be malicious (often "f" of them)
  - All non-malicious lieutenants must come to same decision
  - If general not malicious, lieutenants must follow general
  - Only solvable if $n \geq 3f+1$
- Remote Procedure Call (RPC): Call procedure on remote machine
  - Provides same interface as procedure
  - Automatic packing/unpacking of args without user programming (in stub)

Summary (2/2)

- Internet Protocol (IP)
  - Used to route messages through routes across globe
  - 32-bit addresses, 16-bit ports