Networking and TCP/IP (Con’t), DNS, RPC, Distributed File Systems

April 25th, 2023
Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu
Recall: Distributed Consensus Making

• Consensus problem
  – All nodes propose a value
  – Some nodes might crash and stop responding
  – Eventually, all remaining nodes decide on the same value from set of proposed values

• Distributed Decision Making
  – Choose between “true” and “false”
  – Or Choose between “commit” and “abort”

• Equally important (but often forgotten!): make it durable!
  – How do we make sure that decisions cannot be forgotten?
    » This is the “D” of “ACID” in a regular database
  – In a global-scale system?
    » What about erasure coding or massive replication?
    » Like BlockChain applications!
Recall: Wide Area Network

- **Wide Area Network** (WAN): network that covers a broad area (e.g., city, state, country, entire world)
  - E.g., Internet is a WAN
- WAN connects multiple Local Area Networks (LANs)
- Datalink layer networks are connected by **routers**
  - Different LANs can use different communication technology (e.g., wireless, cellular, optics, wired)
Recall: Routers

- **Forward** each packet received on an **incoming link** to an **outgoing link** based on packet’s destination IP address (towards its destination)
- **Store & forward**: packets are buffered before being forwarded
- **Forwarding table**: mapping between IP address and the output link
Recall: Packet Forwarding

- Upon receiving a packet, a router
  - read the IP destination address of the packet
  - consults its forwarding table → output port
  - forwards packet to corresponding output port

- Default route (for subnets without explicit entries)
  - Forward to more authoritative router
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 (e.g. BGP globally, OSPF locally,…)

Naming in the Internet

- 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, and load

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

- Mechanism: Domain Naming System (DNS)
• 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:
  – Can get someone to route to server, thinking that they are routing to a different server
    » Get them to log into “bank” – give up username and password

• Is DNS Secure?
  – Definitely a weak link
    » What if “response” returned from different server than original query?
    » Get person to use incorrect IP address!
  – Attempt to avoid substitution attacks:
    » Query includes random number which must be returned

• In July 2008, hole in DNS security located!
  – Dan Kaminsky (security researcher) discovered an attack that broke DNS globally
    » 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 no speculation until patches applied
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 perhaps through a switch
- Our goal in the following is to show how to construct a secure, ordered, message service routed to anywhere:

<table>
<thead>
<tr>
<th>Physical Reality: Packets</th>
<th>Abstraction: Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limited Size (MTU)</td>
<td>Arbitrary Size</td>
</tr>
<tr>
<td>Unordered (sometimes)</td>
<td>Ordered</td>
</tr>
<tr>
<td>Unreliable</td>
<td>Reliable</td>
</tr>
<tr>
<td>Machine-to-machine</td>
<td>Process-to-process</td>
</tr>
<tr>
<td>Only on local area net</td>
<td>Routed anywhere</td>
</tr>
<tr>
<td>Asynchronous</td>
<td>Synchronous</td>
</tr>
<tr>
<td>Insecure</td>
<td>Secure</td>
</tr>
</tbody>
</table>
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): Provides way to send datagrams to arbitrary destination
  – 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)
Recall: IPv4 Packet Format

- **IP Packet Format:**

  - **IP Datagram:** an unreliable, unordered, packet sent from source to destination
    - Function of network – deliver datagrams!

  - **IP Header Length**
  - **Total length (16-bits)**
  - **Flags & Fragmentation** to split large messages
  - **20 bytes**
  - **IP header**
  - **IP Ver4**
  - **Time to Live (hops)**
  - **Type of transport protocol**
  - **16-bit identification**
  - **TTL**
  - **protocol**
  - **13-bit frag off**
  - **options (if any)**
  - **32-bit source IP address**
  - **32-bit destination IP address**
  - **16-bit header checksum**
  - **protocol**

  - **Data**

  - **4 5 1 6 3 1**
Building a messaging service on IP

- 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]

- For example: The Unreliable Datagram Protocol (UDP)
  - Layered on top of basic IP (IP Protocol 17)
  - Datagram: an unreliable, unordered, packet sent from source user → dest user (Call it UDP/IP)

```plaintext
<table>
<thead>
<tr>
<th>IP Header (20 bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16-bit source port</td>
</tr>
<tr>
<td>16-bit destination port</td>
</tr>
<tr>
<td>16-bit UDP length</td>
</tr>
<tr>
<td>16-bit UDP checksum</td>
</tr>
</tbody>
</table>
```

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

• Midterm 3: *This Thursday*
  – No class on Thursday. I’ll have special office hours during class time.
  – Three double-sided pages of notes
  – Watch for Ed post about where you should go: we have multiple exam rooms
• All material up to today’s lecture is fair game
• Final deadlines during RRR week:
  – Yes, there will be office hour – watch for specifics
• Also – we have a special lecture (just for fun) next Tuesday
  – During normal class time!
Administrivia (Con’t)

• You need to know your units as CS/Engineering students!

• Units of Time: “s”: Second, “min”: 60s, “h”: 3600s, (of course)
  – Millisecond: \(1\text{ms} \Rightarrow 10^{-3}\text{s}\)
  – Microsecond: \(1\mu\text{s} \Rightarrow 10^{-6}\text{s}\)
  – Nanosecond: \(1\text{ns} \Rightarrow 10^{-9}\text{s}\)
  – Picosecond: \(1\text{ps} \Rightarrow 10^{-12}\text{s}\)

• Integer Sizes: “b” \(\Rightarrow \)”bit”, “B” \(\Rightarrow \) “byte” \(\Rightarrow \) 8 bits, “W” \(\Rightarrow \)”word” \(\Rightarrow \)? (depends. Could be 16b, 32b, 64b)

• Units of Space (memory), sometimes called the “binary system”
  – Kilo: \(1\text{KB} \equiv 1\text{KiB} \Rightarrow 1024\text{ bytes} \Rightarrow 2^{10}\text{ bytes} \Rightarrow 1024 \approx 1.0 \times 10^3\)
  – Mega: \(1\text{MB} \equiv 1\text{MiB} \Rightarrow (1024)^2\text{ bytes} \Rightarrow 2^{20}\text{ bytes} \Rightarrow 1,048,576 \approx 1.0 \times 10^6\)
  – Giga: \(1\text{GB} \equiv 1\text{GiB} \Rightarrow (1024)^3\text{ bytes} \Rightarrow 2^{30}\text{ bytes} \Rightarrow 1,073,741,824 \approx 1.1 \times 10^9\)
  – Tera: \(1\text{TB} \equiv 1\text{TiB} \Rightarrow (1024)^4\text{ bytes} \Rightarrow 2^{40}\text{ bytes} \Rightarrow 1,099,511,627,776 \approx 1.1 \times 10^{12}\)
  – Peta: \(1\text{PB} \equiv 1\text{PiB} \Rightarrow (1024)^5\text{ bytes} \Rightarrow 2^{50}\text{ bytes} \Rightarrow 1,125,899,906,842,624 \approx 1.1 \times 10^{15}\)
  – Exa: \(1\text{EB} \equiv 1\text{EiB} \Rightarrow (1024)^6\text{ bytes} \Rightarrow 2^{60}\text{ bytes} \Rightarrow 1,152,921,504,606,846,976 \approx 1.2 \times 10^{18}\)

• Units of Bandwidth, Space on disk/etc, Everything else…, sometimes called the “decimal system”
  – Kilo: \(1\text{KB/s} \Rightarrow 10^3\text{ bytes/s}, \quad 1\text{KB} \Rightarrow 10^3\text{ bytes}\)
  – Mega: \(1\text{MB/s} \Rightarrow 10^6\text{ bytes/s}, \quad 1\text{MB} \Rightarrow 10^6\text{ bytes}\)
  – Giga: \(1\text{GB/s} \Rightarrow 10^9\text{ bytes/s}, \quad 1\text{GB} \Rightarrow 10^9\text{ bytes}\)
  – Tera: \(1\text{TB/s} \Rightarrow 10^{12}\text{ bytes/s}, \quad 1\text{TB} \Rightarrow 10^{12}\text{ bytes}\)
  – Peta: \(1\text{PB/s} \Rightarrow 10^{15}\text{ bytes/s}, \quad 1\text{PB} \Rightarrow 10^{15}\text{ bytes}\)
  – Exa: \(1\text{EB/s} \Rightarrow 10^{18}\text{ bytes/s}, \quad 1\text{EB} \Rightarrow 10^{18}\text{ bytes}\)
Internet Architecture: Five Layers

- Lower three layers implemented everywhere
- Top two layers implemented only at hosts
Internet Architecture: Five Layers

- Communication goes down to physical network
- Then from network peer to peer
- Then up to relevant layer
Layering Analogy: Packets in Envelopes

101010100110101110

Layering Analogy: Packets in Envelopes

101010100110101110
Internet Transport Protocols

• Datagram service (UDP): IP Protocol 17
  – No-frills extension of “best-effort” IP
  – Multiplexing/Demultiplexing among processes

• Reliable, in-order delivery (TCP): IP Protocol 6
  – Connection set-up & tear-down
  – Discarding corrupted packets (segments)
  – Retransmission of lost packets (segments)
  – Flow control/Congestion control

• Other examples:
  – DCCP (33), Datagram Congestion Control Protocol
  – RDP (26), Reliable Data Protocol
  – SCTP (132), Stream Control Transmission Protocol
Network Address Translation: Transport-Level IP Sharing

- Network Address Translation (NAT): Allow multiple clients to share Public IP
  - Translate connections with Private IP addresses to Public IP Address (of firewall)
- Allocate unique (client) port at firewall to distinguish different connections

TCP Connection #1: [192.168.1.4, Port: 6543, 142.251.42.36, Port: 80]
TCP Connection #1: [128.32.5.3, Port: 4340, 142.251.42.36, Port: 80]
TCP Connection #2: [192.168.1.4, Port: 8977, 11.33.40.5, Port: 80]
TCP Connection #2: [128.32.5.3, Port: 4341, 11.33.40.5, Port: 80]
Recall: Sockets in concept

**Client**
- Create Client Socket
- Connect it to server (host:port)
- Close Client Socket

**Server**
- Create Server Socket
- Bind it to an Address (host:port)
- Listen for Connection
- Accept syscall()
- Close Server Socket
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
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”
Problem: Dropped Packets

- All physical networks can garble or drop packets
  - Physical hardware problems (bad wire, bad signal)
- Therefore, IP can garble or drop packets
  - It doesn't repair this itself (end-to-end principle!)
- Building reliable message delivery
  - Confirm that packets aren't garbled
  - Confirm that packets arrive exactly once
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
Stop-and-Wait (No Packet Loss)

- Send; wait for ACK; repeat
- Round Trip Time (RTT): time it takes a packet to travel from sender to receiver and back
  - One-way latency \( d \): one way delay from sender and receiver
- For symmetric latency, \( RTT = 2d \)
Stop-and-Wait (No Packet Loss)

- How fast can you send data?
- Little’s Law applied to the network: \( n = B \cdot \text{RTT} \)
- For Stop-and-Wait, \( n = 1 \) packet
- So bandwidth is 1 packet per RTT
  - Depends only on latency, not network capacity (!)
Stop-and-Wait (No Packet Loss)

- So bandwidth is 1 packet per RTT
  - Depends only on latency, not network capacity (!)

- Suppose RTT = 100 ms and 1 packet is 1500 Bytes

- Throughput = \( \frac{1500 \text{ Bytes} \times 8 \text{ bits/Byte}}{100 \text{ ms} \times 10^{-3} \text{s/ms}} \)
  = 120 Kbps

- Very inefficient if we have a 100 Mbps link!
Stop-and-Wait with Packet Loss

- Loss recovery relies on timeouts
- How to choose a good timeout?
  - Too short – lots of duplication
  - Too long – packet loss is really disruptive!
- How to deal with duplication?
  - Retransmission certainly opens up the possibility for copies of packets
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 small number of 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:** doesn’t work if network can delay or duplicate messages arbitrarily
Advantages of Moving Away From Stop-and-Wait

• Larger space of acknowledgements
  – Pipelining: don’t wait for ACK before sending next packet

• ACKs serve dual purpose:
  – Reliability: Confirming packet received
  – Ordering: Packets can be reordered at destination

• How much data is in flight now?
  – Bytes in-flight: \( W_{send} = RTT \times B \)
  – Here \( B \) is in “bytes/second”
  – \( W_{send} \equiv \) Sender’s “Window Size”
  – Packets in flight = \( (W_{send} / \text{packet size}) \)

• How long does the sender have to keep the packets around?
• How long does the receiver have to keep the packets’ data?
• What if sender is sending packets faster than the receiver can process the data?
Recall: Communication Between Processes

- Data written by A is held in memory until B reads it
- Queue has a fixed capacity
  - Writing to the queue blocks if the queue is full
  - Reading from the queue blocks if the queue is empty
- POSIX provides this abstraction in the form of pipes

```c
write(wfd, wbuf, wlen);

n = read(rfd, rbuf, rmax);
```
Buffering in a TCP Connection

- A single TCP connection needs **four** in-memory queues:
  - Send buffer: add data on write syscall, remove data when ACK received
  - Receive buffer: add data when packets received, remove data on read syscall
A host’s window size for a TCP connection is how much remaining space it has in its receive queue.

A host advertises its window size in every TCP packet it sends!

Sender never sends more than receiver’s advertised window size.
Sliding Window Protocol

- TCP sender knows receiver’s window size, and aims never to exceed it.
- But packets that it previously send may arrive, filling the window size!

**Rule:** TCP sender ensures that:

Number of Sent but UnACKed Bytes < Receiver’s Advertised Window Size

- Can send new packets as long as sent-but-unacked packets haven’t already filled the advertised window size.
Sliding Window (No Packet Loss)

- Example: Window size ($w$) = 3 packets
- Window size to fill link is given by:
  \[ w = B_{pkt} \cdot RTT \]
- $B_{pkt} \equiv \text{Packets/sec}$
- Little’s Law once again!
- For TCP, window is in bytes, not packets

<table>
<thead>
<tr>
<th>Time</th>
<th>Sender</th>
<th>Receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{1}</td>
<td>{}</td>
</tr>
<tr>
<td>2</td>
<td>{1, 2}</td>
<td>{}</td>
</tr>
<tr>
<td>3</td>
<td>{1, 2, 3}</td>
<td>{}</td>
</tr>
<tr>
<td>4</td>
<td>{2, 3, 4}</td>
<td>{}</td>
</tr>
<tr>
<td>5</td>
<td>{3, 4, 5}</td>
<td>{}</td>
</tr>
<tr>
<td>6</td>
<td>{4, 5, 6}</td>
<td>{}</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
</tbody>
</table>
TCP Windows and Sequence Numbers: PER BYTE!

- **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)
Congestion

• Too much data trying to flow through some part of the network

• IP’s solution: Drop packets

• What happens to TCP connection?
  – Lots of retransmission – wasted work and wasted bandwidth (when bandwidth is scarce)
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”
Congestion Management

- TCP artificially restricts the window size if it sees packet loss
- Careful control loop to make sure:
  1. We don’t send too fast and overwhelm the network
  2. We utilize most of the bandwidth the network has available
  - In general, these are conflicting goals!

Recall: Connection Setup over TCP/IP

• 5-Tuple identifies each connection:
  1. Source IP Address
  2. Destination IP Address
  3. Source Port Number
  4. Destination Port Number
  5. Protocol (always TCP here)

• Often, Client Port “randomly” assigned
  – Done by OS during client socket setup

• Server Port often “well known”
  – 80 (web), 443 (secure web), 25 (sendmail), etc
  – Well-known ports from 0—1023
Establishing TCP Service

• Open connection: 3-way handshaking
  – Need to establish bidirectional communication, including sequence numbers

• Reliable byte stream transfer from (IPa, TCP_Port1) to (IPb, TCP_Port2)
  – Indication if connection fails: Reset

• Close (tear-down) connection
Sockets in concept

Client

- Create Client Socket
- Connect it to server (host:port)
- Read response
- Write request
- Close Client Socket

Server

- Create Server Socket
- Bind it to an Address (host:port)
- Listen for Connection
- Accept syscall()
- Read request
- Write response
- Close Connection Socket
- Close Server Socket
Open Connection: 3-Way Handshake

- Server calls `listen()` to wait for a new connection.
- Client calls `connect()` providing server’s IP address and port number.
- Each side sends SYN packet proposing an initial sequence number (one for each sender) and ACKs the other.
Sockets in concept

Client

Create Client Socket

Connect it to server (host:port)

Connection Socket

write request

read response

Close Client Socket

Server

Create Server Socket

Bind it to an Address (host:port)

Listen for Connection

Accept syscall()

Connection Socket

read request

write response

Close Server Socket

Close Connection Socket
Close Connection: 4-Way Teardown

• Connection is not closed until both sides agree

• If multiple FDs on Host 1 refer to this connection, all of them must be closed
• Same for close() call on Host 2

Host 1

close()

FIN

FIN ACK

Any calls to read() return 0

close()

OS deallocates connection state

Host 2

data

FIN

FIN ACK

OS deallocates connection state

Can retransmit FIN ACK if it is lost

OS discards data (no socket to give it to)

timeout
Recall: Distributed Applications Build With Messages

• 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
Question: Data Representation

• An object in memory has a machine-specific binary representation
  – Threads within a single process have the same view of what’s in memory
  – Easy to compute offsets into fields, follow pointers, etc.

• In the absence of shared memory, externalizing an object requires us to turn it into a sequential sequence of bytes
  – **Serialization/Marshalling:** Express an object as a sequence of bytes
  – **Deserialization/Unmarshalling:** Reconstructing the original object from its marshalled form at destination
Simple Data Types

uint32_t x;

• Suppose I want to write a x to a file

• First, open the file: FILE* f = fopen("foo.txt", "w");

• Then, I have two choices:
  1. fprintf(f, "%lu", x);
  2. fwrite(&x, sizeof(uint32_t), 1, f);
     » Or equivalently, write(fd, &x, sizeof(uint32_t)); (perhaps with a loop to be safe)

• Neither one is “wrong” but sender and receiver should be consistent!
Machine Representation

- Consider using the machine representation:
  - fwrite(&x, sizeof(uint32_t), 1, f);

- How do we know if the recipient represents $x$ in the same way?
  - For pipes, is this a problem?
  - What about for sockets?
Endianness

• For a byte-address machine, which end of a machine-recognized object (e.g., int) does its byte-address refer to?

• Big Endian: address is the most-significant bits

• Little Endian: address is the least-significant bits
What Endian is the Internet?

- Big Endian
  - Network byte order
  - Vs. “host byte order”

NAME

arpa/inet.h - definitions for internet operations

SYNOPSIS

#include <arpa/inet.h>

DESCRIPTION

The in_port_t and in_addr_t types shall be defined as described in <netinet/in.h>.
The in_addr structure shall be defined as described in <netinet/in.h>.
The INET_ADDRSTRLEN [8] and INET6_ADDRSTRLEN [8] macros shall be defined as described in <netinet/in.h>.
The following shall either be declared as functions, defined as macros, or both. If functions are declared, function prototypes:

```c
uint32_t htonl(uint32_t);
uint16_t htons(uint16_t);
uint32_t ntohl(uint32_t);
uint16_t ntohs(uint16_t);
```
The uint32_t and uint16_t types shall be defined as described in <inttypes.h>.
The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided:

```c
in_addr_t inet_addr(const char *);
char *inet_ntoa(struct in_addr);
const char *inet_ntop(int, const void *restrict, char *restrict, socklen_t);
int inet_pton(int, const char *restrict, void *restrict);
```

Inclusion of the <arpa/inet.h> header may also make visible all symbols from <netinet/in.h> and <inttypes.h>.
Dealing with Endianness

- Decide on an “on-wire” endianness
- Convert from native endianness to “on-wire” endianness before sending out data (**serialization/marshalling**)
  - `uint32_t htonl(uint32_t)` and `uint16_t htons(uint16_t)` convert from native endianness to network endianness (big endian)

- Convert from “on-wire” endianness to native endianness when receiving data (**deserialization/unmarshalling**)
  - `uint32_t ntohl(uint32_t)` and `uint16_t ntohs(uint16_t)` convert from network endianness to native endianness (big endian)
What About Richer Objects?

- Consider `word_count_t` of Homework 0 and 1 ...
- Each element contains:
  - An int
  - A `pointer` to a string (of some length)
  - A `pointer` to the next element
- `fprintf_words` writes these as a sequence of lines (character strings with `\n`) to a file stream
- What if you wanted to write the whole list as a binary object (and read it back as one)?
  - How do you represent the string?
  - Does it make any sense to write the pointer?

```c
typedef struct word_count {
    char *word;
    int count;
    struct word_count *next;
} word_count_t;
```
Data Serialization Formats

- JSON and XML are commonly used in web applications
- Lots of ad-hoc formats
# Data Serialization Formats

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache Avro</td>
<td>Apache Software Foundation</td>
<td>N/A</td>
<td>Yes</td>
<td>Apache Avro™ 1.8.x Specification</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes (book)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>Zero-stage support for binary and human-readable data.</td>
</tr>
<tr>
<td>Apache Parquet</td>
<td>Apache Software Foundation</td>
<td>N/A</td>
<td>Yes</td>
<td>Apache Parquet™ 1.8 Specification</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>N/A</td>
</tr>
<tr>
<td>ASH</td>
<td>ISG, ERC, IT&amp;I</td>
<td>N/A</td>
<td>Yes</td>
<td>EICM/WIP 9, 98-96 series of ITU-T Recommendations</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>Yes (ISO)</td>
</tr>
<tr>
<td>Bonita</td>
<td>Start-Citizen Software</td>
<td>N/A</td>
<td>Yes</td>
<td>Enhance protocol specification</td>
<td>Partial</td>
<td>Yes</td>
<td>Yes (book)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>N/A</td>
</tr>
<tr>
<td>Box</td>
<td>Server Platform</td>
<td>N/A</td>
<td>No</td>
<td>Box Specification</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>N/A</td>
</tr>
<tr>
<td>JSON</td>
<td>MongoDB</td>
<td>JSON</td>
<td>No</td>
<td>JSON Specification</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>N/A</td>
</tr>
<tr>
<td>CBIX</td>
<td>Corentis</td>
<td>JSON</td>
<td>Yes</td>
<td>CBIX Specification</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>N/A</td>
</tr>
<tr>
<td>Common Object Request</td>
<td>Java object serialization</td>
<td>N/A</td>
<td>Partial</td>
<td>Java object serialization</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes (book)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>N/A</td>
</tr>
<tr>
<td>Common Data Representation</td>
<td>CDR Management Group</td>
<td>N/A</td>
<td>Yes</td>
<td>Generic Inter-COINS Protocol</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>N/A</td>
</tr>
<tr>
<td>D-Bus Message Protocol</td>
<td>Message Protocol</td>
<td>N/A</td>
<td>Yes</td>
<td>D-Bus Protocol</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>N/A</td>
</tr>
<tr>
<td>Elastic XML Metadata</td>
<td>XSD</td>
<td>XML</td>
<td>Yes</td>
<td>Elastic XML interoperability profiles</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes (book)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>N/A</td>
</tr>
<tr>
<td>FileMeta</td>
<td>Google</td>
<td>N/A</td>
<td>No</td>
<td>FileMeta structure</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>N/A</td>
</tr>
<tr>
<td>Fast (Simple)</td>
<td>ISG, ERC, IT&amp;I</td>
<td>N/A</td>
<td>Yes</td>
<td>Fast (Simple)</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>N/A</td>
</tr>
<tr>
<td>FHIR</td>
<td>Health Level 7</td>
<td>FHIR</td>
<td>Yes</td>
<td>FHIR interoperability profiles</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes (book)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>N/A</td>
</tr>
<tr>
<td>IBM</td>
<td>AWS</td>
<td>JSON</td>
<td>Yes</td>
<td>The Amazon JSON Specification</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>N/A</td>
</tr>
<tr>
<td>Java serialization</td>
<td>Oracle Corporation</td>
<td>N/A</td>
<td>Yes</td>
<td>Java Object serialization</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes (book)</td>
<td>NA</td>
<td>NA</td>
<td>N/A</td>
</tr>
<tr>
<td>JSON</td>
<td>Doug Lea</td>
<td>JSON</td>
<td>Yes</td>
<td>JSON Specification</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes (book)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>N/A</td>
</tr>
<tr>
<td>MongoDB</td>
<td>MongoDB</td>
<td>JSON (book)</td>
<td>Yes</td>
<td>MongoDB protocol specification</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>N/A</td>
</tr>
<tr>
<td>NSDL</td>
<td>Not applicable</td>
<td>N/A</td>
<td>No</td>
<td>NSDL specification</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>N/A</td>
</tr>
<tr>
<td>OPC DA Browser</td>
<td>OPC Foundation</td>
<td>OPC</td>
<td>No</td>
<td>OPC foundation</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>N/A</td>
</tr>
<tr>
<td>OpenSSL</td>
<td>OpenSSL</td>
<td>C, PHP</td>
<td>No</td>
<td>OpenSSL</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes (book)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>N/A</td>
</tr>
<tr>
<td>Paho (Python)</td>
<td>Guido van Rossum</td>
<td>Python</td>
<td>Yes</td>
<td>Paho protocol specification</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>N/A</td>
</tr>
<tr>
<td>Property ID</td>
<td>Not applicable</td>
<td>Not (book)</td>
<td>Yes</td>
<td>Property ID</td>
<td>Yes</td>
<td>No</td>
<td>?</td>
<td>Yes (book)</td>
<td>NA</td>
<td>NA</td>
<td>N/A</td>
</tr>
<tr>
<td>Protocol Buffers (protobuf)</td>
<td>Google</td>
<td>N/A</td>
<td>No</td>
<td>Protocol Buffers specification</td>
<td>Yes</td>
<td>Yes (book)</td>
<td>Yes</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>N/A</td>
</tr>
</tbody>
</table>

4/25/23

Kubiatowicz CS162 © UCB Spring 2023
Remote Procedure Call (RPC)

• 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
  – And must deal with machine representation by hand

• Another option: Remote Procedure Call (RPC)
  – Calls a procedure on a remote machine
  – Idea: Make communication look like an ordinary function call
  – Automate all of the complexity of translating between representations
  – Client calls:
    ```
    remoteFileSystem→Read("rutabaga");
    ```
  – Translated automatically into call on server:
    ```
    fileSys→Read("rutabaga");
    ```
RPC Concept

Client (caller)

\[ r = f(v_1, v_2); \]

Server (callee)

\[ \text{res}_t f(a_1, a_2) \]
Client (caller)
\[ r = f(v1, v2); \]

Server (callee)
\[ res_t f(a1, a2) \]
RPC 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 Details (1/3)

• 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
RPC Details (2/3)

• Cross-platform issues:
  – What if client/server machines are different architectures/ languages?
    » Convert everything to/from some canonical form
    » Tag every item with an indication of how it is encoded (avoids unnecessary conversions)

• How does client know which mbox (destination queue) 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
RPC Details (3/3)

- **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 dist. 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
Problems with RPC: Performance

• RPC is \textit{not} performance transparent:
  – Cost of Procedure call « same-machine RPC « network RPC
  – Overheads: Marshalling, Stubs, Kernel-Crossing, Communication

• 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 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

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-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
- *Mount* remote files into your local file system
  - Directory in local file system refers to remote files
  - e.g., `/users/jane/prog/foo.c` on laptop actually refers to `/prog/foo.c` on `adj.cs.berkeley.edu`

- **Naming Choices:**
  - `[Hostname,localname]`: Filename includes server
    - No location or migration transparency, except through DNS remapping
  - A global name space: Filename unique in “world”
    - Can be served by any server
Enabling Design: VFS

- The System Call Interface
- Process Management
- Memory Management
- Filesystems
- Device Control
- Networking
- Concurrency, multitasking
- Memory Manager
- Files and dirs: the VFS
- Connectivity
- Architecture Dependent Code
- File System Types
- Block Devices
- Device Control
- TTYs and device access
- Network Subsystem
- IF drivers
Recall: Layers of I/O...

length = read(input_fd, buffer, BUFFER_SIZE);

ssize_t read(int, void *, size_t) {
    marshal args into registers
    issue syscall
    register result of syscall to rtn value
}

Exception U→K, interrupt processing
void syscall_handler (struct intr_frame *f) {
    unmarshall call#, args from regs
    dispatch : handlers[call#](args)
    marshal results fo syscall ret
}

ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos) {
    User Process/File System relationship
call device driver to do the work
}
Virtual Filesystem Switch

- **VFS**: Virtual abstraction similar to local file system
  - Provides virtual superblocks, inodes, files, etc
  - Compatible with a variety of local and remote file systems
    » provides object-oriented way of implementing file systems
- **VFS** allows the same system call interface (the API) to be used for different types of file systems
  - The API is to the VFS interface, rather than any specific type of file system

```c
inf = open("/floppy/TEST", O_RDONLY, 0);
outf = open("/tmp/test",
           O_WRONLY|O_CREAT|O_TRUNC, 0600);

   do {
       i = read(inf, buf, 4096);
       write(outf, buf, i);
   } while (i);

close(outf);
close(inf);
```
VFS Common File Model in Linux

- Four primary object types for VFS:
  - superblock object: represents a specific mounted filesystem
  - inode object: represents a specific file
  - dentry object: represents a directory entry
  - file object: represents open file associated with process
- There is no specific directory object (VFS treats directories as files)
- May need to fit the model by faking it
  - Example: make it look like directories are files
  - Example: make it look like have inodes, superblocks, etc.
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, but can be cache at server-side
- Advantage: Server provides 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
Dealing with Failures

• What if server crashes? Can client wait until it comes back and just continue making requests?
  – Changes in server's cache but not in disk are lost

• What if there is shared state across RPC's?
  – Client opens file, then does a seek
  – Server crashes
  – What if client wants to do another read?

• Similar problem: What if client removes a file but server crashes before acknowledgement?
Stateless Protocol

- **Stateless Protocol**: A protocol in which all information required to service a request is included with the request.

  Even better: Idempotent Operations – repeating an operation multiple times is same as executing it just once (e.g., storing to a mem addr.)

- Client: timeout expires without reply, just run the operation again (safe regardless of first attempt)

- Recall HTTP: Also a stateless protocol
  - Include cookies with request to simulate a session
Case Study: Network File System (NFS)

- Three Layers for NFS system
  - **UNIX file-system interface**: 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
  - XDR Serialization standard for data format independence
  - 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 required 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 the same effect as performing them 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 other side effects
  - Example: What about “remove”? NFS does operation twice and second time returns an advisory error
- **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):
    » Hang until server comes back up (next week?)
    » Return an error. (Of course, most applications don’t know they are talking over network)
NFS Architecture

- System calls interface
- VFS interface
- Other types of file systems
- UNIX file system
- NFS client
- RPC/XDR
- Disk
- Network
- VFS interface
- NFS server
- UNIX file system
- RPC/XDR
- Disk
NFS Cache consistency

- NFS protocol: weak consistency
  - Client polls server periodically to check for changes
    » Polls server if data hasn’t been checked in last 3-30 seconds (exact timeout is 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?
  - i.e. what if 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

- 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 # clients
    » Must keep checking to see if caches out of date
    » Server becomes bottleneck due to polling traffic
Andrew File System

- Andrew File System (AFS, late 80’s) → DCE DFS (commercial product)
- **Callbacks:** Server records who has copy of file
  - On changes, server immediately tells all with old copy
  - No polling bandwidth (continuous checking) needed
- **Write through on close**
  - Changes not propagated to server until close()
  - Session semantics: updates visible to other clients only after the file is closed
    » As a result, do not get partial writes: all or nothing!
    » Although, for processes on local machine, updates visible immediately to other programs who have file open
- In AFS, everyone who has file open sees old version
  - Don’t get newer versions until reopen file
Andrew File System (con’t)

- Data cached on local disk of client as well as memory
  - On open with a cache miss (file not on local disk):
    » Get file from server, set up callback with server
  - On write followed by close:
    » Send copy to server; tells all clients with copies to fetch new version from server on next open (using callbacks)

- What if server crashes? Lose all callback state!
  - Reconstruct callback information from client: go ask everyone “who has which files cached?”

- AFS Pro: Relative to NFS, less server load:
  - Disk as cache ⇒ more files can be cached locally
  - Callbacks ⇒ server not involved if file is read-only

- For both AFS and NFS: central server is bottleneck!
  - Performance: all writes⇒server, cache misses⇒server
  - Availability: Server is single point of failure
  - Cost: server machine’s high cost relative to workstation
Summary (1/2)

• **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

• **Remote Procedure Call (RPC)**: Call procedure on remote machine or in remote domain
  – Provides same interface as procedure
  – Automatic packing and unpacking of arguments without user programming (in stub)
  – Adapts automatically to different hardware and software architectures at remote end
Summary (2/2)

- **Distributed File System:**
  - Transparent access to files stored on a remote disk
  - Caching for performance

- **VFS:** Virtual File System layer (Or Virtual Filesystem Switch)
  - Provides mechanism which gives same system call interface for different types of file systems

- **Cache Consistency:** Keeping client caches consistent with one another
  - If multiple clients, some reading and some writing, how do stale cached copies get updated?
    - NFS: check periodically for changes
    - AFS: clients register callbacks to be notified by server of changes