Recall: 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" router

Congestion Avoidance

- 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)

TCP solution: "slow start" (start sending slowly)

- If no timeout, slowly increase window size (throughput) by 1 for each ACK received
- If timeout, cut window size in half, wait for more ACKs

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
  - How does the sender decide window size?
  - sender chooses a random value for window size
  - sender sends an initial window size (W)
  - receiver sends an ACK with a maximum segment size (MSS)
  - sender should choose to be less than MSS
  - sender should choose a window size (W') that is less than MSS and
  - fills network between sender and receiver

- 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**

- A 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)

**Establishing TCP Service**

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

**Sockets in concept**

- Client
  - Connection socket
  - Create Client Socket
    - Connect it to server (host:port)
  - Accept
    - syscall()
  - read request
  - Closed Client Socket
- Server
  - Connection socket
  - Create Server Socket
    - Bind it to an Address (host:port)
  - Listen for Connection
    - Accept
    - Close Connection Socket
  - Close Server Socket

**Recall Connection Setup over TCP/IP**
Open Connection: 3-Way Handshake

- Client (initiator) sends SYN, SeqNum = x
- Server sends SYN and ACK, SeqNum = y and Ack = x + 1
- Client sends ACK, Ack = y + 1

Close Connection: 4-Way Teardown

- Connection is not closed until both sides agree
- FIN, FIN ACK, FIN, FIN ACK

Recall: Distributed Applications Build With Messages

- How do you actually program a distributed application?
- Need to synchronize multiple threads running on different machines
- One Abstraction: send/receive messages
- Already atomic: no receiver gets portion of a message and two receivers cannot get same

Network
Send
Receive
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

```c
uint32_t x;
```

• Suppose I want to write a to the file:
  • First open the file:
    ```c
    FILE* f = fopen("foo.txt", "w");
    ```
  • Then, I have two choices:
    1. `fprintf(f, "%lu", x);`
    2. `fwrite(&x, sizeof(uint32_t), 1, f);`

Machine Representation

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

How do we know if the recipient represents x in the same way?

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 byte(s) (not always the same for all processors)
• Little Endian: Address is the least-significant byte(s)

- What is the problem for sockets?
- For pipes is this a problem?

- Consider the machine representation:
  ```c
  fprintf(f, "%lu", x);
  ```

- In the absence of shared memory, externalizing an object requires us to turn it into a sequential sequence of bytes

- Easy to compute offsets in objects following pointers, etc.
- Threads within a single process have the same view of what's in memory
What Endian is the Internet?
- Big Endian
- Network byte order
- Vs. “host byte order”

Dealing with Endianness
- Decide on an “on-wire” endianness
- Convert from native endianness to “on-wire” endianness before sending out data
  - uint32_t htonl(uint32_t)
  - uint16_t htons(uint16_t)
- Convert from “on-wire” endianness to native endianness when receiving data
  - uint32_t ntohl(uint32_t)
  - uint16_t ntohs(uint16_t)

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)?
- A pointer to a string
- A pointer to another (possibly empty)
- Each element contains:
  - Consider word_count_t of Homework 0 and 1…
  - …

Data Serialization Formats
- JSON and XML are commonly used in web applications

Data Endianness
- Network endianness to native endianness (big endian)
  - htonl(uint32_t) and htons(uint16_t) convert from network to native endianness when receiving data
- Native endianness to network endianness (big endian)
  - ntohl(uint32_t) and ntohs(uint16_t) convert from native to network endianness when sending data
- Decode on “on-wire” endianness
  - Network byte order

Big Endian

What is the Internet?
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 (caller):
\[
\text{r} = f(v_1, v_2);
\]

Server (callee):
\[
\text{res}_t = f(a_1, a_2);
\]

Machine A

Machine B

Packet Handler

Packet Handler

Network

Network

Server Stub

unbundle args

send

return

receive

Server Stub

unbundle args

send

return

receive

Client Stub

bundle args

unbundle return vals

bundle return vals

send

receive

Client

Stub

Data Serialization Formats

Remote Procedure Call (RPC)
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 (depending on system)
  - Converting values to a canonical form, serializing objects, copying arguments passed by value
  - Providing efficient, memory-safe data structure

- Dynamic Binding
  - Used in client-server communication

- Dynamic Binding
  - Most RPC systems use dynamic binding via name service
    - Name service provides dynamic translation of service

- Dynamic Binding
  - Why dynamic binding?
    - Access control: check who is permitted to access service
    - Fail-over: If server fails, use a different one

- Dynamic Binding
  - What if there are multiple servers?
    - Could give flexibility at binding time
      - Choose loaded server for each new client
      - Could provide same return mailbox (router-level redirect)
Problems with RPC: Non-Atomic Failures

- Different failure modes in distributed systems 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 the 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 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

- Example: split kernel into application-level servers
- The system looks remote even though on the same machine
- Services can be run wherever it's most appropriate
  - Service need not be on same machine as client

Examples of RPC systems:
- CORBA (Common Object Request Broker Architecture)
- DCOM (Distributed COM)
- RMI (Java Remote Method Invocation)
- ...
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:

- \([\text{Hostname}, \text{localname}]\):
  - 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

AMS: The System Call Interface

- Concurrency, multitasking
- Virtual memory
- Files and dirs: the VFS
- TTYs and device access
- Connectivity

MIDTERM 3: Thursday 4/28: 7-9PM

- All course material...
- Review session Monday 4/25 1-3PM
Virtual Filesystem Switch

VFS Common File Model in Linux

Simple Distributed File System

Virtual Filesystem

Exception Handling

User Layers of I/O...
Use of 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 not committed in server cache, lose some of the advantages of caching

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 the same as executing it just once (e.g., storing to a memory address.)
- Client: timeout expires without reply, just run the operation again (safe regardless of first attempt)
- Recall HTTP: Also a stateless protocol

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
- 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 (inode, position)`
  - 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 a network)

NFS Architecture

- 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 the 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?
  - Assume we want distributed system to behave exactly the same as it does on a single CPU
  - 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; other clients use old version of file until timeout.

NFS Cache Consistency

- Example: Start with file contents = "A"
- What would we actually want?
  - Assume we want distributed system to behave exactly the same as if all processes are running on a single CPU
  - 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

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

Client 2:
- Read: parts of B or C

Client 3:
- Read: parts of B or C
NFS Pros and Cons

- 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 (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

- What if server crashes? Lose all callback state!
  - Reconstruct callback information from client and server
    - Can 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!
- Cost: server machine's high cost relative to workstation
- Performance: all writes go to server, cache misses go to server
- Availability: Server is single point of failure
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 a mechanism for different types of file systems to have the same system call interface

- Cache Consistency:
  - Keeping client caches consistent with one another
  - NFS: clients register callbacks to be notified by server of changes
  - AFS: clients register callbacks to be notified by server of changes

Thank you!

- Thanks for all your great questions!
- Good bye! You have all been great!