CS162
Operating Systems and Systems Programming
Lecture 25

RPC, NFS and AFS

April 26th, 2022
Prof. Anthony Joseph and John Kubiatowicz
http://cs162.eecs.Berkeley.edu
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”
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 \(\rightarrow\) wastes time if message lost
    - Too short \(\rightarrow\) retransmit even though ACK will arrive shortly
  - Stability problem: more congestion \(\Rightarrow\) ACK is delayed \(\Rightarrow\) unnecessary timeout \(\Rightarrow\) more traffic
    - \(\Rightarrow\) 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 \(\Rightarrow\) 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

1. Open connection: 3-way handshaking

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

3. Close (tear-down) connection
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 Connection Socket

Note: 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

**Client (initiator)**

- `connect()`
- SYN, SeqNum = x
- SYN and ACK, SeqNum = y and Ack = x + 1
- ACK, Ack = y + 1

**Server**

- `listen()`
- allocate buffer space, connection enqueued
- `accept()` dequeues connection

4/2622

Joseph & Kubiatowicz CS162 © UCB Spring 2022

Lec 25.9
Sockets in concept

Client

Create Client Socket

Connect it to server (host:port)

Server

Create Server Socket

Bind it to an Address (host:port)

Listen for Connection

Accept syscall()

Connection Socket

Connection Socket

write request

read request

write response

read response

Close Client Socket

Close Connection Socket

Close Server 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

- OS deallocates connection state after close() on both sides
- Can retransmit FIN ACK if it is lost
- OS discards data (no socket to give it to)
- Any calls to read() return 0
- OS deallocates connection state upon close()
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

```c
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

```
int main(int argc, char *argv[]) {
    int val = 0x12345678;
    int i;
    printf("val = %x\n", val);
    for (i = 0; i < sizeof(val); i++) {
        printf("val[%d] = %x\n", i, ((uint8_t *) &val)[i]);
    }
}
(base) CullerMac19:code09 culler$ ./endian
val = 12345678
val[0] = 78
val[1] = 56
val[2] = 34
val[3] = 12
```
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 and INET6_ADDRSTRLEN 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

  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.

  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?
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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache Avro</td>
<td>Apache Software Foundation</td>
<td>1.x</td>
<td>No</td>
<td>No</td>
<td>Apache Avro</td>
<td>Java, C++</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes (Java)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Apache Thrift</td>
<td>Apache Software Foundation</td>
<td>3.x</td>
<td>No</td>
<td>No</td>
<td>Apache Avro</td>
<td>Java, C++</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes (Java)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>AWS SNS</td>
<td>AWS, AWS, PTPY</td>
<td>1.x</td>
<td>Yes</td>
<td>Yes</td>
<td>SQS, SNS, SQS, SNS</td>
<td>Java, Python</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Bencode</td>
<td>Bencode</td>
<td>1.x</td>
<td>No</td>
<td>No</td>
<td>Bencode</td>
<td>Java</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>BSON</td>
<td>MongoDB</td>
<td>1.x</td>
<td>No</td>
<td>No</td>
<td>BSON Specification</td>
<td>Java, Python</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>CBOR</td>
<td>IETF, Mozilla Foundation</td>
<td>1.x</td>
<td>No</td>
<td>No</td>
<td>CBOR</td>
<td>JavaScript</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Comma-separated values (CSV)</td>
<td>RFC author</td>
<td>1.x</td>
<td>No</td>
<td>No</td>
<td>CSV</td>
<td>Python</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Common Data Representation (CDR)</td>
<td>Open Management Group</td>
<td>1.x</td>
<td>Yes</td>
<td>Yes</td>
<td>CDR</td>
<td>Java, Python</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>D-Bus Message Protocol</td>
<td>FreetLib.org</td>
<td>1.x</td>
<td>No</td>
<td>No</td>
<td>D-Bus</td>
<td>C, Java</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Efficient Message Exchange (EMX)</td>
<td>Mozilla</td>
<td>1.x</td>
<td>Yes</td>
<td>Yes</td>
<td>EMX</td>
<td>Java, Python</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>FFI</td>
<td>Google</td>
<td>1.x</td>
<td>No</td>
<td>No</td>
<td>FFI</td>
<td>Java, Python</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>FastWeb</td>
<td>Mozilla</td>
<td>1.x</td>
<td>Yes</td>
<td>Yes</td>
<td>JSON</td>
<td>C, Java</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>FastJSON</td>
<td>Amazon</td>
<td>1.x</td>
<td>No</td>
<td>No</td>
<td>FastJSON</td>
<td>Java</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Fused Serialization</td>
<td>Oracle Corporation</td>
<td>1.x</td>
<td>Yes</td>
<td>Yes</td>
<td>Fused Serialization</td>
<td>Java</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>JSON</td>
<td>Google</td>
<td>1.x</td>
<td>Yes</td>
<td>Yes</td>
<td>JSON</td>
<td>Java, Python</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Logstash</td>
<td>Google</td>
<td>1.x</td>
<td>Yes</td>
<td>Yes</td>
<td>Logstash</td>
<td>Java, Python</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>MessagePack</td>
<td>Rapid7</td>
<td>1.x</td>
<td>No</td>
<td>No</td>
<td>MessagePack</td>
<td>Java, Python</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Protocol Buffers</td>
<td>Google</td>
<td>3.x</td>
<td>Yes</td>
<td>Yes</td>
<td>Protocol Buffers</td>
<td>Java, Python</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Property File</td>
<td>Apache Lucene</td>
<td>1.x</td>
<td>Yes</td>
<td>Yes</td>
<td>Property File</td>
<td>Java, Python</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Property Buffers (protobuf)</td>
<td>Google</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Property Buffers</td>
<td>Java, Python</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**
- Yes: Supported
- No: Not supported
- NA: Not applicable

*4/2622 Joseph & Kubiatowicz CS162 © UCB Spring 2022*
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:
    \[ \text{remoteFileSystem} \rightarrow \text{Read("rutabaga")}; \]
  – Translated automatically into call on server:
    \[ \text{fileSys} \rightarrow \text{Read("rutabaga")}; \]
**RPC Concept**

- **Client (caller)**
  - \( r = f(v_1, v_2); \)
  - Bundle args
  - Call
  - Return
  - Unbundle ret vals
  - Send
  - Receive

- **Server (callee)**
  - \( res_t f(a_1, a_2) \)
  - Bundle ret vals
  - Call
  - Return
  - Unbundle args
  - Send
  - Receive
Client (caller)

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

Server (callee)

\[ \text{res_t } f(a_1, a_2) \]
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 $\leftrightarrow$ Request Message
  - Result $\leftrightarrow$ 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 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”
Administrivia

• Midterm 3: Thursday 4/28: 7-9PM
  – All course material
  – Review session Monday 4/25 1-3PM
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
Virtual memory
Files and dirs: the VFS
File System Types
Block Devices
TTYs and device access
Device Control
Connectivity

Architecture
Dependent Code
Memory Manager
Device Control
Network Subsystem
IF drivers

Concurrency, multitasking
Virtual memory
Files and dirs: the VFS
File System Types
Block Devices
TTYs and device access
Device Control
Connectivity

Architecture
Dependent Code
Memory Manager
Device Control
Network Subsystem
IF drivers
Recall: Layers of I/O…

length = \texttt{read}(\texttt{input\_fd}, \texttt{buffer}, \texttt{BUFFER\_SIZE});

\texttt{ssize\_t read}(\texttt{int}, \texttt{void*}, \texttt{size\_t}) {
    \texttt{marshal args into registers}
    \texttt{issue syscall}
    \texttt{register result of syscall to rtn value}
};

\texttt{Exception U\to K, interrupt processing}
\texttt{void syscall\_handler (struct intr\_frame *f) { }
    \texttt{unmarshall call#, args from regs}
    \texttt{dispatch : handlers[call#](args)}
    \texttt{marshal results fo syscall ret}
};

\texttt{ssize\_t vfs\_read(struct file *file, char \_\_user *buf,}
\texttt{size\_t count, loff\_t \*pos) { }
    \texttt{User Process/File System relationship}
    \texttt{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);
doi
   i = read(inf, buf, 4096);
   write(outf, buf, i);
} while (i);
close(outf);
close(inf);
```
• 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 require for execution
  – E.g. reads include information for entire operation, such as `ReadAt(inumber,position)`, not `Read(openfile)`
  – No need to perform network open() or close() on file – each operation stands on its own
• **Idempotent**: Performing requests multiple times has same effect as performing 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
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”

- 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
Thank you!

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