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: Two-Phase Commit Protocol (2PC)

- **Prepare Phase:**
  - The global coordinator requests that all participants will promise to commit or rollback the transaction.
  - Participants record promise in log, then acknowledge.
  - If anyone votes to abort, coordinator writes "Abort" in its log and tells everyone to abort; each records "Abort" in log.

- **Commit Phase:**
  - After all participants respond that they are prepared, then the coordinator writes "Commit" to its log.
  - Then asks all nodes to commit; they respond with ACK.
  - After receive ACKs, coordinator writes "Got Commit" to log.

- **Persistent stable log on each machine:** keep track of whether commit has happened.
  - Required for good semantics.
  - If a machine crashes, when it wakes up it first checks its log to recover state of world at time of crash.
Alternatives to 2PC

• Three-Phase Commit: One more phase, allows nodes to fail or block and still make progress.
• PAXOS: An alternative used by Google and others that does not have 2PC blocking problem
  – Develop by Leslie Lamport (Turing Award Winner)
  – No fixed leader, can choose new leader on fly, deal with failure
  – Some think this is extremely complex!
• RAFT: PAXOS alternative from John Osterhout (Stanford)
  – Simpler to describe complete protocol

• What happens if one or more of the nodes is malicious?
  – Malicious: attempting to compromise the decision making
  – Use a more hardened decision making process: Byzantine Agreement and Block Chains
Byzantine General’s Problem

- Byzantine General’s Problem (n players):
  - One General and n-1 Lieutenants
  - Some number of these (f) can be insane or malicious

- The commanding general must send an order to his n-1 lieutenants such that the following Integrity Constraints apply:
  - IC1: All loyal lieutenants obey the same order
  - IC2: If the commanding general is loyal, then all loyal lieutenants obey the order he sends
Byzantine General’s Problem (con’t)

• Impossibility Results:
  – Cannot solve Byzantine General’s Problem with n=3 because one malicious player can mess up things
  – With f faults, need n > 3f to solve problem

• Various algorithms exist to solve problem
  – Original algorithm has #messages exponential in n
  – Newer algorithms have message complexity O(n^2)
    » One from MIT, for instance (Castro and Liskov, 1999)

• Use of BFT (Byzantine Fault Tolerance) algorithm
  – Allow multiple machines to make a coordinated decision even if some subset of them (< n/3 ) are malicious
Is a BlockChain a Distributed Decision Making Algorithm?

• BlockChain: a chain of blocks connected by hashes to root block
  – The Hash Pointers are unforgeable (assumption)
  – The Chain has no branches except perhaps for heads
  – Blocks are considered “authentic” part of chain when they have authenticity info in them

• How is the head chosen?
  – Some consensus algorithm
  – In many BlockChain algorithms (e.g. BitCoin, Ethereum), the head is chosen by solving hard problem
    » This is the job of “miners” who try to find “nonce” info that makes hash over block have specified number of zero bits in it
    » The result is a “Proof of Work” (POW)
    » Selected blocks above (green) have POW in them and can be included in chains
  – Longest chain wins
Is a Blockchain a Distributed Decision Making Algorithm? (Con’t)

• Decision means: Proposal is locked into BlockChain
  – Could be Commit/Abort decision
  – Could be Choice of Value, State Transition, ….
• NAK: Didn’t make it into the block chain (must retry!)
• Anyone in world can verify the result of decision making!
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
    » Over Internet, destination specified by IP address and Port (Recall Web server example!)
  - 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
How do we know that both sides speak same language?

- 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 points to most-significant byte
  • Little Endian: address points to least-significant byte

Experiment:

```c
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

Result:
What Endian is the Internet?

**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 [IP6] and INET6_ADDRSTRLEN [3] 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>`.

- **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 (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: Many Options

<table>
<thead>
<tr>
<th>Name</th>
<th>Creator/ maintainer</th>
<th>Based on</th>
<th>Standardized?</th>
<th>Specification</th>
<th>Binary?</th>
<th>Human-readable?</th>
<th>Supports references?</th>
<th>Schema/DSL?</th>
<th>Standard APIs</th>
<th>Supports (These are generally operations)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache Avro</td>
<td>Apache Software Foundation</td>
<td>N/A</td>
<td>No</td>
<td>Apache Avro™ &amp; 1.1 Specfication</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Java, Python</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Apache Parquet</td>
<td>Apache Software Foundation</td>
<td>N/A</td>
<td>No</td>
<td>Apache Parquet® 1</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Java, Python</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>ASN.1</td>
<td>ISO, IEC, ITU-T</td>
<td>N/A</td>
<td>Yes</td>
<td>ISO/IEC 8859, X.680 series of ITU-T Recommendations</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Bencode</td>
<td>Bram Cohen (creator), <a href="https://">license</a></td>
<td>N/A</td>
<td>No</td>
<td>Bencode protocol specification</td>
<td>Partial</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Bin</td>
<td>Bernard Ronders</td>
<td>N/A</td>
<td>No</td>
<td>Bin Specification</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>BSON</td>
<td>MongoDB</td>
<td>JSON</td>
<td>No</td>
<td>BSON Specification</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>CBOR</td>
<td>Christian Bornemann, P. Hofmann</td>
<td>JSON (binary)</td>
<td>Yes</td>
<td>RFC 7492</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes (CBOR)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Comma-separated values (CSV)</td>
<td>EDC author: YellowBrick</td>
<td>N/A</td>
<td>Partial (practically all variants used)</td>
<td>RFC 4180 (among others)</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>N/A</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Common Data Representation (CDR)</td>
<td>Open Management Group</td>
<td>N/A</td>
<td>Yes</td>
<td>General Inter-OSI Protocol</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>N/A</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>D-Bus Message Protocol</td>
<td>freedesktop.org</td>
<td>N/A</td>
<td>Yes</td>
<td>D-Bus Specification</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Partial (Signature schema)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Efficient XML, Interchange (EEX)</td>
<td>W3C</td>
<td>XML, Efficient XML</td>
<td>Yes</td>
<td>EEX Syntax, Interchange CDS Format 1.0</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>(EEX)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Flatbuffers</td>
<td>Google</td>
<td>N/A</td>
<td>No</td>
<td>Flatbuffers protocol specification</td>
<td>Yes</td>
<td>No</td>
<td>Yes (Apache Avro)</td>
<td>Partial (Internal to the buffer)</td>
<td>Yes (C, C++, Java, C#, Go, Python, Rust, JavaScript, PHP, C#, C, C++, Llvm, Tpp)</td>
<td>No</td>
</tr>
<tr>
<td>FAST</td>
<td>ISO, IEC, ITU-T</td>
<td>XML</td>
<td>Yes</td>
<td>FAST T K.841 and ISO/IEC 11178-1:2007</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>(HTML, XHTML)</td>
<td>Yes</td>
<td>(XML, schema)</td>
</tr>
<tr>
<td>FastR</td>
<td>Health Level 7</td>
<td>REST-based</td>
<td>Yes</td>
<td>FastRedaction</td>
<td>Interoperability Resources</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>N/A</td>
<td>No</td>
</tr>
<tr>
<td>Ion</td>
<td>Amazon</td>
<td>JSON</td>
<td>No</td>
<td>Ion specification</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>JSON serialization</td>
<td>Christie Conradon</td>
<td>N/A</td>
<td>Yes</td>
<td>JSON Object Serialization</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes (N/A)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>JSON</td>
<td>Douglas Crockford</td>
<td>JavaScript syntax</td>
<td>Yes</td>
<td>JSON Schema, RFC 4627, JSON Pointer</td>
<td>No, but see BSON, Smi, UDS</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>JSON Schema (Proposal), JSON (DRAFT), JSON-LD</td>
<td>Partial (Connect, JSON-LD, JSON-LD)</td>
</tr>
<tr>
<td>MessagePack</td>
<td>Stephen Punekal</td>
<td>JSON (binary)</td>
<td>No</td>
<td>MessagePack format specification</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Netlify</td>
<td>Dan Bernstein</td>
<td>N/A</td>
<td>No</td>
<td>netlink</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>OASIS</td>
<td>Roll Team</td>
<td>1</td>
<td>No</td>
<td>Specfication</td>
<td>(Binary Specification)</td>
<td>Yes</td>
<td>Yes</td>
<td>(Path Specification)</td>
<td>(Schema/DSL)</td>
<td>N/A</td>
</tr>
<tr>
<td>OPC UA Binary</td>
<td>OPC Foundation</td>
<td>N/A</td>
<td>No</td>
<td>OPC Foundation</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>OpenCDL</td>
<td>Eric Largett</td>
<td>C, PHP</td>
<td>No</td>
<td>OpenCDL.org</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No (OpenCDL library)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Protocol (Python)</td>
<td>Guide to Resources</td>
<td>Python</td>
<td>De facto standard via Python Encapsulation Formats (PEF)</td>
<td></td>
<td>Yes (PEF 0.14 - Protocol Encapsulation Formats)</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
<td>No</td>
</tr>
<tr>
<td>Property list</td>
<td>NIST (inserst)</td>
<td>Apple</td>
<td>1</td>
<td>Public DTD for XML, formati</td>
<td>Yes (1)</td>
<td>Yes</td>
<td>No</td>
<td>1</td>
<td>C, C++, C#, Java, Python, JavaScript, Go</td>
<td>No</td>
</tr>
<tr>
<td>Protocol Buffers: gRPC</td>
<td>Google</td>
<td>N/A</td>
<td>No</td>
<td>Protocol Buffers: Encoding</td>
<td>Yes</td>
<td>Partial</td>
<td>No</td>
<td>Yes (Built-in)</td>
<td>C, C++, C#, Java, Python, Javascript, Go</td>
<td>No</td>
</tr>
</tbody>
</table>

4/25/24  Kubiatowicz CS162 © UCB Spring 2024  Lec 25.18
Administrivia

• 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 hours – 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” ⇒ ”bit”, “B” ⇒ “byte” = 8 bits, “W”⇒”word”=? (depends. Could be 16b, 32b, 64b)

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

• Units of Bandwidth, Space on disk/etc, Everything else…, sometimes called the “decimal system”
  – Kilo: $1\text{KB/s} \Rightarrow 10^3\text{ bytes/s}$, $1\text{KB} \Rightarrow 10^3\text{ bytes}$
  – Mega: $1\text{MB/s} \Rightarrow 10^6\text{ bytes/s}$, $1\text{MB} \Rightarrow 10^6\text{ bytes}$
  – Giga: $1\text{GB/s} \Rightarrow 10^9\text{ bytes/s}$, $1\text{GB} \Rightarrow 10^9\text{ bytes}$
  – Tera: $1\text{TB/s} \Rightarrow 10^{12}\text{ bytes/s}$, $1\text{TB} \Rightarrow 10^{12}\text{ bytes}$
  – Peta: $1\text{PB/s} \Rightarrow 10^{15}\text{ bytes/s}$, $1\text{PB} \Rightarrow 10^{15}\text{ bytes}$
  – Exa: $1\text{EB/s} \Rightarrow 10^{18}\text{ bytes/s}$, $1\text{EB} \Rightarrow 10^{18}\text{ bytes}$
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)
\[ \text{res}_t f(a_1, a_2) \]
RPC Information Flow

Client (caller)

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

Server (callee)

\[ \text{res} = f(a_1, a_2) \]

Machine A

Machine B

Packet Handler

Network
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.
  – Use of standardized *serialization* protocol
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”
**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
- TTYs and device access
- Connectivity
- Architecture Dependent Code
- Memory Manager
- File System Types
- Block Devices
- Device Control
- Network Subsystem
- IF drivers
Recall: Layers of I/O…

User App:

User library:

Application / Service
- High Level I/O
- Low Level I/O
- Syscall
- File System
- I/O Driver

User App:

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);

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

\[
\begin{align*}
\text{read}(f1) & \rightarrow V1 \\
\text{read}(f1) & \rightarrow V1 \\
\text{read}(f1) & \rightarrow V1 \\
\text{write}(f1) & \rightarrow \text{OK} \\
\text{read}(f1) & \rightarrow V2 \\
\end{align*}
\]
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

Diagram:

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

<table>
<thead>
<tr>
<th>Client 1:</th>
<th>Read: gets A</th>
<th>Write B</th>
<th>Read: parts of B or C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client 2:</td>
<td>Read: gets A or B</td>
<td>Write C</td>
<td></td>
</tr>
<tr>
<td>Client 3:</td>
<td></td>
<td></td>
<td>Read: parts of B or C</td>
</tr>
</tbody>
</table>

- Time

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

• NFS Pros:
  – Simple, Highly portable

• NFS Cons:
  – Sometimes inconsistent!
  – Doesn’t scale to large # 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)

• Byzantine General’s Problem: distributed decision making with malicious failures
  – One general, n-1 lieutenants: some number of them may be malicious (often “f” of them)
  – All non-malicious lieutenants must come to same decision
  – If general not malicious, lieutenants must follow general
  – Only solvable if $n \geq 3f+1$

• BlockChain protocols:
  – Cryptographically-driven ordering protocol
  – Could be used for distributed decision making

• 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