Distributed Systems

- 48 - Communication Basics
- 49 - NFS
- 50 - AFS
- GFS

Communication Basics

- Building distributed systems
  - all components fail
  - communication fails
  - how to build systems that *rarely* fail from components that do?
- Issues:
  - performance
    - especially with interconnects much slower than buses
  - security
    - systems span users, domains
    - the Internet is scary
  - communication
    - what are the right primitives?
    - what are the right types of applications?
Communication

“progress and correctness of distributed consensus algorithms is impossible to prove in asynchronous environments” - FLP theorem

- communication is fundamentally unreliable
  - packet loss
  - packet corruption
  - packet delays
- maybe don’t rely on reliability
  - maybe add encryption to the link!
  - but….

End-to-End Argument

crypto is always good, right?

- example of end-to-end argument says:
  - provided encryption might not be good enough
    - 3DES is ancient, maybe want to use AES, blowfish
  - provided encryption might be too expensive
    - might not need encryption at all, just adds overhead
  - app semantics might be needed
    - different app messages might have different needs
  - but strong semantics in underlying layers do help
Distributed Systems \textit{reliable communication layers}

- Need to be able to detect and recover from packet loss:
  - \textit{acknowledge} ("ack") receipt of a message

![Message Plus Acknowledgment](image)

Figure 48.3: Message Plus Acknowledgment

- What if we don’t get the ack? How do we even know we don’t get the ack?

![Message Plus Acknowledgment: Dropped Request](image)

Figure 48.4: Message Plus Acknowledgment: Dropped Request
Distributed Systems *reliable communication layers*

- Need to be able to detect and recover from packet loss:
  - *acknowledge* (“ack”) receipt of a message
- What if we don’t get the ack? How do we even know we don’t get the ack?

![Message Plus Acknowledgment: Dropped Reply](image)

- 48.4 and 48.5 appear the same to the server…
  - but the msg was received in 48.4, and not in 48.5
  - this is bad, as server’s default is to repeat the message, not good if messages are not idempotent
- fix is to include *sequence numbers* in messages
  - receiver could track every number ever seen, but expensive.
- *monotonically increasing* sequence numbers better
  - receiver tracks highest received sequence number
    - acks, but does not execute duplicate messages
    - dealing with out-of-order messages (42, 44, 43, 45…)
    - app-dependent
- Seq numbers important for UDP (unreliable), but TCP uses much more sophisticated approaches under the hood.
Remote Procedure Calls

- turn remote requests into procedure calls to local functions
- need interface definition:

```
interface {
  int func1(int arg1);
  int func2(int arg1, int arg2);
};
```

- client stub generator uses interface def to:
  - create a msg buffer
  - pack (marshal) request into buffer
  - send to destination
  - synchronously wait for reply
  - unpack (unmarshal) return values
  - return return values to caller

- server stub generator uses interface def to:
  - unpack (unmarshal) the message
  - call local func w/ arguments
  - pack the return values into a reply buffer
  - send the reply

Remote Procedure Calls

- What about pointers, or other complex data types?
  - architecture- and language-independent encodings
    - JSON
    - protocol buffers
    - etc.

- What about concurrency in server?
  - want the server to be multi-threaded
  - need to ensure no data races between server stubs and the functions they call

- RPC generally doesn’t need reliable communication (TCP)
  - “ack” is not needed, as RPC (“the app”) generally returns a response
Remote Procedure Calls

interface {
    int func1(int arg1);
    int func2(int arg1, int arg2);
};

• server stub generator uses interface def to:
  • unpack (unmarshal) the message
  • call local func w/ arguments
  • pack the return values into a reply buffer
  • send the reply

Distributed Systems

• 48 - Communication Basics
• 49 - NFS
• 50 - AFS
• GFS
NFS  
*Sun Microsystems*

- first widely used distributed file system
  - clients diskless
    - easy sharing
    - centralized admin
    - security

---

**NFS**

- distributed file system should be *transparent*
  - except possibly in performance
  - client issues same file-system calls as standalone system
a distributed system is one where a machine I've never heard of goes down and I can't read my email"

- Leslie Lamport: Turing Award Winner for his work on distributed systems

- NFS goals:
  - simple and fast file recovery
  - *stateless protocol*: server keeps no client state
    - server scales well
    - client crashes transparent
    - server crashes transparent
    - client must maintain all state the server needs for any communication

- NFS actually NFSv2

- file handle: uniquely describe file or directory
  - volume ID
  - inode number
  - generation number (inumbers get re-used)

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NFSPROC.GETATTR</td>
<td>file handle</td>
</tr>
<tr>
<td></td>
<td>returns: attributes</td>
</tr>
<tr>
<td>NFSPROC.SETATTR</td>
<td>file handle, attributes</td>
</tr>
<tr>
<td></td>
<td>returns: –</td>
</tr>
<tr>
<td>NFSPROC.LOOKUP</td>
<td>directory file handle, name of file/dir to look up</td>
</tr>
<tr>
<td></td>
<td>returns: file handle</td>
</tr>
<tr>
<td>NFSPROC.READ</td>
<td>file handle, offset, count</td>
</tr>
<tr>
<td></td>
<td>data, attributes</td>
</tr>
<tr>
<td>NFSPROC.WRITE</td>
<td>file handle, offset, count, data</td>
</tr>
<tr>
<td></td>
<td>attributes</td>
</tr>
<tr>
<td>NFSPROC.CREATE</td>
<td>directory file handle, name of file, attributes</td>
</tr>
<tr>
<td>NFSPROC.REMOVE</td>
<td>directory file handle, name of file to be removed</td>
</tr>
<tr>
<td>NFSPROC.MKDIR</td>
<td>directory file handle, name of directory, attributes</td>
</tr>
<tr>
<td>NFSPROC.RMDIR</td>
<td>directory file handle, name of directory to be removed</td>
</tr>
<tr>
<td>NFSPROC.REaddir</td>
<td>directory handle, count of bytes to read, cookie</td>
</tr>
<tr>
<td></td>
<td>returns: directory entries, cookie (to get more entries)</td>
</tr>
</tbody>
</table>
NFS reading a file:

```
Client: fd = open("/foo", ...);
        Send LOOKUP (rootdir FH, "foo")

Server: Receive LOOKUP request
           look for "foo" in root dir
           return foo's FH + attributes

Receive LOOKUP reply
allocate file desc in open file table
store foo's FH in table
store current file position (0)
return file descriptor to application

read(fd, buffer, MAX);
Index into open file table with fd
get NFS file handle (FH)
use current file position as offset
Send READ (FH, offset=0, count=MAX)

Server: Receive READ request
        use FH to get volume/inode num
        read inode from disk (or cache)
        compute block location (using offset)
        read data from disk (or cache)
        return data to client

Receive READ reply
update file position (+bytes read)
set current file position = MAX
return data/error code to app

read(fd, buffer, MAX);
Same except offset=MAX and set current file position = 2*MAX

read(fd, buffer, MAX);
Same except offset=2*MAX and set current file position = 3*MAX

close(fd);
Just need to clean up local structures
Free descriptor "fd" in open file table
(No need to talk to server)
```
NFS *performance*

- client-side caching
  - read file data (and metadata) cached by client
  - all good unless the file changes on the server
- client-side write buffers
  - coalescing
  - aggregating disparate messages

- However: cache consistency!

---

NFS *cache consistency*

Problems:
- update visibility
  - $C_1$ writes `foo.c`, but does not immediately push to server
  - $C_2$ reads, sees old version
  - $C_1$ flushes to server
- stale cache
  - $C_2$ reads again, still sees old version (`foo.c` locally cached)

Fixes:
- close-to-open consistency
  - every open guaranteed to see every prior write to the server
    - must validate cache (GETATTR)
    - but maybe not all the time

*NFS consistency is weak… (so are most other FSs)*
NFS server caching

- tons of memory
  - wants to use it for disk cache (satisfy reads)
  - wants to use it for write buffer (quickly ack writes)
  - what could go wrong?
- server could ack a write before writing to disk!
  - say file initially has three 4k blocks of data:
    - xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
    - yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
    - zzzzzzzzzzzzzzzzzzzzzzzzzzzzz
  - client overwrites with:
    - write(aaa..., 0)
    - write(bbb..., 4k)
    - write(ccc..., 8k):
  - server crashes after acking second block, before writing:
    - aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
    - yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
    - ccccccbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
  - client never even knows that the server crashed