Questions answered in this lecture:

What is challenging about distributed systems?
How can a reliable messaging protocol be built on unreliable layers?
What is RPC?
What is the NFS stateless protocol?
What are idempotent operations and why are they useful?
What state is tracked on NFS clients?
A distributed system is one where a machine I’ve never heard of can cause my program to fail.

— Leslie Lamport

Definition:
More than 1 machine working together to solve a problem

Examples:
• client/server: web server and web client
• cluster: page rank computation

Other courses:
• **CS 542**: Computer Networks
• **CS 550**: Advanced Operating Systems
WHY GO DISTRIBUTED?

More computing power

More storage capacity

Fault tolerance

Data sharing
NEW CHALLENGES

System failure: need to worry about partial failure

Communication failure: links unreliable
- bit errors
- packet loss
- node/link failure

Motivation example:
Why are network sockets less reliable than pipes?
PIPE

Writer Process

Reader Process

user

kernel
PIPE

Writer Process

Reader Process
PIPE

Writer Process

Reader Process

user

kernel
PIPE

Writer Process

Reader Process

user

kernel
write waits for space
write waits for space
NETWORK SOCKET

Machine A

writer

Router

Machine B

Reader Process

Network Socket

user

kernel

user

kernel

Writer Process

user

kernel

Reader Process

user

kernel
what if router’s buffer is full?
what if B’s buffer is full?
From A’s view, network and B are largely a black box.
Raw messages: UDP

Reliable messages: TCP

Remote procedure call: RPC
UDP: User Datagram Protocol

API:
- reads and writes over socket file descriptors
- messages sent from/to ports to target a process on machine

Provide minimal reliability features:
- messages may be lost
- messages may be reordered
- messages may be duplicated
- only protection: checksums to ensure data not corrupted
Advantages

- Lightweight
- Some applications make better reliability decisions themselves (e.g., video conferencing programs)

Disadvantages

- More difficult to write applications correctly
TCP: Transmission Control Protocol

Using software, build reliable, logical connections over unreliable connections

Techniques:

- acknowledgment (ACK)
**TECHNIQUE #1: ACK**

**Sender**
- [send message]
- [recv ack]

**Receiver**
- [recv message]
- [send ack]

Sender knows message was received
Sender doesn’t receive ACK…
What to do?
**Technique #2: Timeout**

Sender
- [send message]
- [start timer]

... waiting for ack ...

- [timer goes off]
- [send message]

Receiver
- [recv message]
- [send ack]

[recv ack]
LOST ACK: ISSUE 1

How long to wait?

Too long?

• System feels unresponsive

Too short?

• Messages needlessly re-sent
• Messages may have been dropped due to overloaded server. Resending makes overload worse!
How long to wait?

One strategy: be adaptive

Adjust time based on how long acks usually take

For each missing ack, wait longer between retries
What does a lost ack really mean?
Lost ACK: How can sender tell between these two cases?

Case 1

Sender: [send message]  □  Receiver

[timeout]

Case 2

Sender: [send message]  □  Receiver

[recv message]  □  [send ack]

[timeout]

ACK: message received exactly once

No ACK: message may or may not have been received

What if message is command to increment counter?
Proposal:
Sender could send an **AckAck** so receiver knows whether to retry sending an Ack

Sound good?
Suppose generals agree after N messages
Did the arrival of the N’th message change decision?
- if yes: then what if the N’th message had been lost?
- if no: then why bother sending N messages?
RELIABLE MESSAGES: LAYERING STRATEGY

Using software, build reliable, logical connections over unreliable connections

Techniques:
- acknowledgment
- timeout
- remember sent messages
**TECHNIQUE #3: RECEIVER REMEMBERS MESSAGES**

**Sender**
- [send message]
- [timeout]
- [send message]
- [recv ack]

**Receiver**
- [recv message]
- [send ack]
- [ignore message]
- [send ack]

**Questions:**
- how does receiver know to ignore?
SOLUTIONS

Solution 1: remember every message ever received

Solution 2: sequence numbers
- senders gives each message an increasing unique seq number
- receiver knows it has seen all messages before N
- receiver remembers messages received after N

Suppose message K is received. Suppress message if:
- K < N
- Msg K is already buffered
TCP

TCP: Transmission Control Protocol

Most popular protocol based on seq nums

Buffers messages so arrive in order

Timeouts are adaptive
Raw messages: UDP

Reliable messages: TCP

Remote procedure call: RPC
Remote Procedure Call

What could be easier than calling a function?

**Strategy**: create wrappers so calling a function on another machine feels just like calling a local function

Very common abstraction
int main(…) {
    int x = foo(”hello”);
}

int foo(char *msg) {
    send msg to B
    recv msg from B
}

void foo_listener() {
    while(1) {
        recv, call foo
    }
}

Machine A

Machine B

What it feels like for programmer
RPC

Machine A

```c
int main(...) {
    int x = foo("hello");
}

int foo(char *msg) {
    send msg to B
    recv msg from B
}
```

Machine B

```c
int foo(char *msg) {
    ...
}

void foo_listener() {
    while(1) {
        recv, call foo
    }
}
```

Actual calls
int main(...) {
    int x = foo("hello");
}

int foo(char *msg) {
    send msg to B
    recv msg from B
}

Machine A

Machine B

int foo(char *msg) {
    ...
}

void foo_listener() {
    while(1) {
        recv, call foo
    }
}

Wrappers

RPC
RPC TOOLS

RPC packages help with two components

(1) Runtime library
   • Thread pool
   • Socket listeners call functions on server

(2) Stub generation
   • Create wrappers automatically
   • Many tools available (rpcgen, thrift, protobufs)
Wrappers must do conversions:

- client arguments to message
- message to server arguments
- convert server return value to message
- convert message to client return value

Need uniform endianness (wrappers do this)

Conversion is called marshaling/unmarshaling, or serializing/deserializing
Why are pointers problematic?

Address passed from client not valid on server

Solutions?

- smart RPC package: follow pointers and copy data
RPC OVER TCP?

Why wasteful?
Strategy: use function return as implicit ACK

Piggybacking technique

What if function takes a long time?
- then send a separate ACK
DISTRIBUTED FILE SYSTEMS

File systems are great use case for distributed systems

Local FS:
processes on same machine access shared files

Network FS:
processes on different machines access shared files in same way
GOALS FOR DISTRIBUTED FILE SYSTEMS

Fast + simple crash recovery
- both clients and file server may crash

Transparent access
- can’t tell accesses are over the network
- normal UNIX semantics

Reasonable performance
Think of NFS as more of a protocol than a particular file system.

Many companies have implemented NFS: Oracle/Sun, NetApp, EMC, IBM

We’re looking at NFSv2
  • NFSv4 has many changes

Why look at an older protocol?
  • Simpler, focused goals
OVERVIEW

Architecture

Network API

Write Buffering

Cache
NFS ARCHITECTURE
GENERAL STRATEGY: EXPORT FS
GENERAL STRATEGY: EXPORT FS

Client

Server

Local FS

Local FS

read
GENERAL STRATEGY: EXPORT FS

Client

Server

Local FS

read
GENERAL STRATEGY: EXPORT FS

Client

Server

Local FS

Local FS
GENERAL STRATEGY: EXPORT FS

Client

Server

Local FS  NFS

mount

Local FS
• /dev/sda1 on /
• /dev/sdb1 on /backups
• NFS on /home/kyle
GENERAL STRATEGY: EXPORT FS

Client

Local FS  NFS

Server

Local FS

read
GENERAL STRATEGY: EXPORT FS

Client

Server

Local FS
NFS

read

Local FS
GOALS FOR NFS

Fast + simple crash recovery
- both clients and file server may crash

Transparent access
- can’t tell accesses are over the network
- normal UNIX semantics

Reasonable performance
Overview

Architecture

Network API

Write Buffering

Cache
Attempt: Wrap regular UNIX system calls using RPC

open() on client calls open() on server
open() on server returns fd back to client

read(fd) on client calls read(fd) on server
read(fd) on server returns data back to client
FILE DESCRIPTORS

Client

open() = 2

Local FS  NFS

Server

client fds

Local FS
FILE DESCRIPTORS

Client

Local FS  NFS

read(2)

Server

client fds

Local FS
FILE DESCRIPTORS

Client

Local FS  NFS

Server

client fds

Local FS

read(2)
STRATEGY 1 PROBLEMS

What about crashes?

```c
int fd = open("foo", O_RDONLY);
read(fd, buf, MAX);
read(fd, buf, MAX);
...
read(fd, buf, MAX);
```

Imagine server crashes and reboots during reads…

Server crash!

nice if acts like a slow read
1. Run some crash recovery protocol upon reboot
   • Complex

2. Persist fds on server disk.
   • Slow
   • What if client crashes? When can fds be garbage collected?
STRATEGY 2:
PUT ALL INFO IN REQUESTS

Use “stateless” protocol!

• server maintains no state about clients
• server still keeps other state, of course
Eliminate File Descriptors

Client

Local FS  NFS

Server

Local FS
STRATEGY 2: PUT ALL INFO IN REQUESTS

Use “stateless” protocol!

- server maintains no state about clients

Need API change. One possibility:

```c
pread(char *path, buf, size, offset);
pwrite(char *path, buf, size, offset);
```

Specify path and offset each time. Server need not remember anything from clients.

Pros? Server can crash and reboot transparently to clients.

Cons? Too many path lookups.
STRATEGY 3: INODE REQUESTS

inode = open(char *path);
pread(inode, buf, size, offset);
pwrite(inode, buf, size, offset);

This is pretty good! Any correctness problems?

If file is deleted, the inode could be reused
- Inode not guaranteed to be unique over time
Strategy 4: File Handles

```c
fh = open(char *path);
pread(fh, buf, size, offset);
pwrite(fh, buf, size, offset);
```

File Handle = `<volume ID, inode #, generation #>`

Opaque to client (client should not interpret internals)
CAN NFS PROTOCOL INCLUDE APPEND?

fh = open(char *path);
pread(fh, buf, size, offset);
pwrite(fh, buf, size, offset);
append(fh, buf, size);

Problem with append()?

If RPC library retries, what happens when append() is retried?

Problem: Why is it difficult to replay append()?
Replica Suppression is Stateful

Sender
(send message)

Receiver
(recv message)
(send ack)

[timeout]
(send message)

[ignore message]
(send ack)

[recv ack]

TCP suppresses repeated message

Problem: TCP is stateful
If server crashes, forgets which RPC’s have been executed!
Solution:
Design API so no harm to executing function more than once

If \( f() \) is idempotent, then:
\[
\text{\( f() \) has the same effect as \( f(); f(); \ldots f(); f() \)}
\]
PWRITE IS IDEMPOTENT

AAAA
AAAA

AAAA
AAAA

AAAA
AAAA

AAAA
AAAA

AAAA
AAAA
APPEND IS NOT IDEMPOTENT

A → append → AB → append → ABB → append → ABBB
Idempotent

- any sort of read that doesn’t change anything
- pwrite

Not idempotent

- append

What about these?

- mkdir
- creat
Strategy 4: File Handles

fh = open(char *path);

pread(fh, buf, size, offset);
pwrite(fh, buf, size, offset);
append(fh, buf, size);

File Handle = <volume ID, inode #, generation #>
STRATEGY 5: CLIENT LOGIC

Build normal UNIX API on client side on top of idempotent, RPC-based API

Client open() creates a local fd object

It contains:

- file handle
- offset
FILE DESCRIPTORS

Client

Local

FS

NFS

client fds

Server

Local

FS

fd 5

fh=<…>

off=123

read(5, 1024)

local

off=123

pread(fh, 123, 1024)

RPC

local

local FS
OVERVIEW

Architecture

Network API

Write Buffering

Cache
server acknowledges write before write is pushed to disk; what happens if server crashes?
SERVER WRITE BUFFER LOST

client:
write A to 0
write B to 1
write C to 2

server mem: A B C
server disk:

server acknowledges write before write is pushed to disk
client:

write A to 0
write B to 1
write C to 2

server mem:

server disk:

server acknowledges write before write is pushed to disk
SERVER WRITE BUFFER

LOST

client:

write A to 0
write B to 1
write C to 2
write X to 0

server mem:
X B C

server disk:
A B C

server acknowledges write before write is pushed to disk
client:

write A to 0
write B to 1
write C to 2
write X to 0

server acknowledges write before write is pushed to disk
Server write buffer:

Client:

- write A to 0
- write B to 1
- write C to 2
- write X to 0
- write Y to 1

Server memory:

- X
- Y
- C

Server disk:

- X
- B
- C

Server acknowledges write before write is pushed to disk.
SERVER WRITE BUFFER

LOST

client:

write A to 0
write B to 1
write C to 2
write X to 0
write Y to 1

server mem: 

server disk: X B C 

server acknowledges write before write is pushed to disk

crash!
SERVER WRITE BUFFER

LOST

client:

write A to 0
write B to 1
write C to 2
write X to 0
write Y to 1

server mem:

server disk: X B C

server acknowledges write before write is pushed to disk
client:

write A to 0
write B to 1
write C to 2
write X to 0
write Y to 1
write Z to 2

server mem: Z

server disk: X B C

server acknowledges write before write is pushed to disk
SERVER WRITE BUFFER

LOST

client:

write A to 0
write B to 1
write C to 2
write X to 0
write Y to 1
write Z to 2

server mem:  

server disk:  

Problem:
No write failed, but disk state doesn’t match any point in time

Solutions?????
1. Don’t use server write buffer (persist data to disk before acknowledging write)
   Problem: Slow!
2. use persistent write buffer (more expensive)
OVERVIEW

Architecture

Network API

Write Buffering

Cache
NFS can cache data in three places:

- server memory
- client disk
- client memory

How to make sure all versions are in sync?
DISTRIBUTED CACHE

Client 1

NFS cache:

Server

Local FS cache: A

Client 2

NFS cache:
CACHE

Client 1

NFS cache: A

read

Server

Local FS cache: A

Client 2

NFS cache:
CACHE

Client 1

NFS

cache: A

Server

Local FS

cache: A

Client 2

NFS

cache: A

read
“Update Visibility” problem: server doesn’t have latest version

What happens if Client 2 (or any other client) reads data? Sees old version (different semantics than local FS)
“Stale Cache” problem:
client 2 doesn’t have latest version

What happens if Client 2 reads data?
Sees old version (different semantics than local FS)
CACHE

Client 1

NFS

cache: B

Server

Local FS

cache: B

Client 2

NFS

cache: B

read
When client buffers a write, how can server (and other clients) see update?

- Client flushes cache entry to server

**When** should client perform flush?????? (3 reasonable options??)

**NFS solution:** flush on fd close
Problem 2: Stale Cache

Problem: Client 2 has stale copy of data; how can it get the latest?

One possible solution:
- If NFS had state, server could push out update to relevant clients

NFS solution:
- Clients recheck if cached copy is current before using data
Client cache records time when data block was fetched (t1)

Before using data block, client does a STAT request to server
- get’s last modified timestamp for this file (t2) (not block…)
- compare to cache timestamp
- refetch data block if changed since timestamp (t2 > t1)
NFS developers found `stat` accounted for 90% of server requests

Why?

Because clients frequently recheck cache
Solution: cache results of `stat` calls

What is the result? Never see updates on server!

Partial Solution: Make stat cache entries expire after a given time (e.g., 3 seconds) (discard t2 at client 2)

What is the result? Could read data that is up to 3 seconds old
NFS handles client and server crashes very well; robust APIs are often:

- **stateless**: servers don’t remember clients

- **idempotent**: doing things twice never hurts

Caching and write buffering is harder in distributed systems, especially with crashes

Problems:

- Consistency model is odd (client may not see updates until 3 seconds after file is closed)
- Scalability limitations as more clients call stat() on server