# 15-440 Recitation 4: Intro to DFS Lab

Vijay Vasudevan

# Announcements

- DFS Lab Part 1 out
  - Due next Thursday, October 8 at 11:59pm
  - You need to have partners!
- Final exam dates available, FYI.

# You build a DFS!

- 4 stages, each building on each other
  1. Lock server, at-most-once RPC semantics
  2. Implement extent server; create/lookup/readdir FUSE ops
  3. Implement read/write/open/setattr
  4. Implement mkdir/unlink, integrate locks!
- You have ~1 month to do it, you might need it all

# Today: Part 1

- Implementing the lock server
  - Provide mutual exclusion to 'lock resources'
  - Using pthread mutexes, condition variables!
- Implementing at-most-once RPC semantics

# Using Virtual Machines

- VirtualBox virtualizing software
- We will provide the base Ubuntu OS image
- Root password is “systems” if you want other packages
  - sudo apt-get install package
- Demo....

# Providing mutual exclusion over the net

- In the lab: lock is a 64-bit number
- Client might:
  - acquire(lock_a); do work; release(lock_a);
- You must create the lock if it doesn’t yet exist on the server

# Example tester

```
% ./lock_tester 3772
simple lock client
acquire a release a acquire a release a
acquire a acquire b release b release a
test2: client 0 acquire a release a
test2: client 2 acquire a release a
...
```

# What’s provided

- Simple RPC framework, skeleton for lockserver code
- What it does: handles setting up sockets, marshalling/unmarshalling basic data types, sending over TCP (whew)
- What it doesn’t: does not keep track of RPC request state; duplicate RPCs are invoked twice! :(

# Example RPC call

```
lock_protocol::status
lock_server::stat(int clt, lock_protocol::lockid_t lid, int &r)
{
        lock_protocol::status ret = lock_protocol::OK;
        printf("stat request from clt %d\n", clt);
        r = 0;
        return ret;
}
```

```
% make
% ./lock_server 3772
% ./lock_demo 3772
stat request from clt 1450783179
stat returned 0
```

In lock server main:

```
   lock_server ls;
   rpcs server(htons(atoi(argv[1])));
   server.reg(lock_protocol::stat, &ls, &lock_server::stat);
```

# Your job for lock server

```
lock_protocol::status lock_server::acquire(int clt, lock_protocol::lockid_t lid, ...)
{
        // Does lock exist?
        // Is it available?
        // If not...
        // when function returns, only the calling client should own the lock.
}
lock_protocol::status lock_server::release(int clt, lock_protocol::lockid_t lid, ...)
{
     // Can assume lock exists (no malicious clients)
     // Unlock resource
     // Notify appropriate parties
}
```

RPC server spawns a thread for every RPC call

# Does it work?

- RPC_LOSSY: drops, duplicates, delays.
- Run lock_tester with RPC_LOSSY=0
- Now try running it with RPC_LOSSY=5
  - hmm... it should hang or error.  FAIL!

# Why?

- No at-most-once RPC semantics implemented...yet.
- If reply dropped, duplicate sent:
  - acquire(a), acquire(a)............................

# RPC reliability

- If RPC request dropped, need to resend
  - (timeouts!)
- If RPC request delayed, might try to resend.
  - (timeouts will cause duplicate to be sent)
- If RPC framework duplicates request
  - need to ensure request applied once!

# Implementing at-most-once

- Your job
  - Start the timeout thread in rpc server constructor
  - On server: Manage state of RPCs sent, replies

# Strawman

- Remember every RPC call
  - Client sends a unique RPC identifier
- Remember every RPC reply
  - To avoid invoking the actual function

- What's the problem with this approach?

# Sliding window RPCs

**Assume the following sent by client:**

```
marshall m1 << clt nonce  // client id
            << svr nonce  // server id
            << proc  // procedure number (bind, acquire, etc).
            << myxid // unique request id for this RPC

            << req.str(); // Data
```

**But we need some additional info from the client**

# Client's Reply window

- Keeps track of "outstanding" replies.
- Example.

# Sliding window RPCs

**Sent by client:**

```
marshall m1 << clt nonce  // client id
            << svr nonce  // server id
            << proc  // procedure number (bind, acquire, etc).
            << myxid // unique request id for this RPC
            << xid_rep_window.front() // Last out-of-order RPC reply received
            << req.str(); // Data
```

**On server:**

You must use the client id, xid, and last out of order RPC

1) Check whether the request is new, in progress, done, or forgotten

2) Figure out which replies you can forget

3) Keep track of replies of the local RPC calls to ensure at-most-once.

# At the end

```
% export RPC_LOSSY=5
% ./lock_server 3772 &
% ./lock_tester 3772
...
all tests passed
% ./rpctest
simple_tests
...
all tests passed
```

# Things you'll need to know

- You will be required to know a little bit about C++ STL data structures for Proj 2
- pthread mutexes, condition variables
  - Read the man pages!
  - Make sure you initialize them