```
6.824 2012 Lecture 1: Introduction and lab overview

6.824: Distributed Systems Engineering

What is a distributed system?
  multiple connected computers
  cooperate to provide some service
  Examples: Internet E-Mail, Athena file server, Google MapReduce

Why distribute?
  to connect physically separate entities
  to achieve security via physical isolation
  to tolerate faults via replication at separate sites
  to increase performance via parallel CPUs/mem/disk/net

But:
  complex, hard to debug
  new classes of problems, e.g. partial failure (did he accept my e-mail?)
  Lamport: A distributed system is one in which the failure of a
    computer you didn't even know existed can render your own computer
    unusable.
  don't distribute if a central system will work

Why take this course?
  interesting -- hard problems, non-obvious solutions
  active research area -- lots of progress + big unsolved problems
  used by real systems -- unlike 10 years ago
    internet, clusters, multi-site, failures
  hands-on -- you'll build a real system in the labs

COURSE STRUCTURE

http://pdos.csail.mit.edu/6.824

Meetings: 1/2 lecture, 1/2 paper discussion (or lab help)

Research papers -- case studies
  must read papers before class
    otherwise boring, and you can't pick it up by listening
  each paper has a question (see web site)
  submit answer before class, one or two paragraphs

Mid-term quiz in class, and final exam

Labs: build a real cluster file system, like Frangipani
  Labs are due on Fridays
  First lab is due next week

Project: extend lab in any way you like.
  teams of two
  one-page report
  demo in last class meeting

Yandong Mao is TA, office hours on Web.


Example:
  single shared file system, so users can cooperate
  lots of client computers
  [diagram: clients, network, vague set of servers]

Topic: architecture
  Choice of interfaces
```

```
    Monolithic file server?
    Block server(s) -> FS logic in clients?
    Separate naming + file servers?
    Separate FS + block servers?
  Single machine room or unified wide area system?
    Wide-area dramatically more difficult.
  Client/server or peer-to-peer?
  Interact w/ performance, security, fault behavior.

Topic: implementation
  How do clients/servers communicate?
    Direct network communication is pretty painful
    Want to hide network stuff from application logic
  Most systems organize distribution with some structuring framework(s)
    RPC, RMI, DSM, MapReduce, &c

Topic: performance
  Distribution can hurt: network b/w and latency bottlenecks
    Lots of tricks, e.g. caching, threaded servers
  Distribution can help: parallelism
  Idea: scalable design
    Nx servers -> Nx total performance
  Need a way to divide the load by N
    Split by user
    Split by file name
  Rarely perfect -> only scales so far
    Load imbalance
      One very active user
      One very popular file
      -> one server 100%, added servers mostly idle
      -> Nx servers -> 1x performance
    Global operations, e.g. search

Topic: fault tolerance
  Can I use my files if server / network fails?
  Maybe: replicate the data on multiple servers
    Perhaps client sends every operation to both
    Maybe only needs to wait for one reply
  Opportunity: operate from two "replicas" independently if partitioned?
  Opportunity: can 2 servers yield 2x availability AND 2x performance?

Topic: consistency
  == contract w/ apps/users about meaning of operations
    hard due to partial failure, replication/caching, concurrency
  Problem: keep replicas identical
    If one is down, it will miss operations
      Must be brought up to date after reboot
    If net is broken, *both* replicas maybe live, and see different ops
      Delete file, still visible via other replica
  Problem: operations may appear to clients in different orders
    Due to caching or replication
    I make grades.txt unreadable, then TA write grades to it
    What if the operations exec in different order to different replicas?
  Problem: atomicity of multi-server operations
    mv /users/x/f /users/y/f
    Might an observer see no file at all?
    Might a crash result in lost file?
  Consistency often hurts performance (communication, blocking)
    Many systems cut corners -- "relaxed consistency"

Topic: security
  Threats:
    corrupt employees
```

```
    targeted external attack
    spammers collecting botnets
  How does the server know that a request is from me?
  Do I have to trust the system administrators?
  What if server code has bugs? (one per 1000 lines...)

We want to understand the individual techniques, and how to combine them

LAB: YET ANOTHER FILE SYSTEM (YFS)

Lab is inspired by Frangipani, a scalable distributed file system (see
paper). Designed/built at a research lab, not a product. You will
build a simplified version of Frangipani.

Frangipani goals
  scalable storage -- add disk servers
  scalable file service -- add file servers
  consistent -- much like single file server
  adaptive -- shifts data to where it is being used
  fault-tolerant -- server, network, and disk failures

Frangipani design
  diagram:
    client workstations
    Frangipani servers
    Petal servers
    lock servers
  Petal looks like a single huge disk
    interface: put and get (pretty low-level)
    replicates to tolerate any single server/disk failure
    add petal servers to increase storage capacity and throughput
  Frangipani file server
    knows about file names, directories, inodes, &c
    uses Petal for all storage
    all Frangipani servers serve same file system
      communicate only via Petal
    Frangipani servers cache file/directory data
      cache is write-back
      can often operate w/o contacting Petal
    add Frangipani servers to handle more client workstations
  Lock servers
    Frangipani servers use lock service to provide consistency
      e.g., lock the directory when creating a file
      locks also drive cache write-back to Petal
    locks servers are replicated
  No security beyond traditional file system security
    intended for a cluster, not wide-area

This is a common architecture
  many clients
  many front ends to provide CPU to process client requests
  shared back-end that *only* provides storage
  front-ends independent, communicate indirectly via back-end storage
  only storage back-end needs to be fault-tolerant
    front-ends are "stateless", nothing lost if they crash
  easier to provide fault-tolerance for storage than for general computation

When does Frangipani scale well?
  lots of independent users
  each user's Frangipani caches that user's files
  Frangipani servers don't need to wait for each other for locks
  so each added server doesn't slow down other servers => scalable
  write-back caching means Petal isn't a bottleneck
```

```
When might Frangipani *not* scale well?
  i.e. when might we not be able to get more performance by adding servers?
  if all clients are modifying the same set of files
  if there's a single demanding user

YFS
  Same basic structure as Frangipani, but single extent server
    i.e. you don't have to implement Petal.
  Diagram: many clients, one extent_server, multiple lock_servers
  Client diagram:
    app, kernel fuse, fuse.cc, yfs_client
  Each program written in C++ and pthreads, our own RPC library
    Next two lectures cover infrastructure in detail

Labs: build YFS incrementally
  L1: simple lock server
    threads, mutexes, condition variables
    then at-most-once RPC
  L2: extent_server, yfs_client, fuse
    in-memory store for extent_server
    basic yfs_client: create, lookup, readdir, write, read
  L3: yfs_client + extent_server + lock_server
    mkdir, unlink, and sharing/locks
  L4: caching lock_server
    add revocation to lock server
  L5: caching extent_server
    consistency using lock_server
  L6: paxos library
    agreement protocol (who is the master lock server?)
  L7: fault tolerant lock server
    replicated state machine using paxos
  L8: your choice
    e.g. distributed extent server for performance or fault tolerance

Lab 1: simple lock server

what does a lock server do?
  handles acquire and release RPCs from clients
  supports multiple locks, each lock has a number
  acquire(num):
    if another client holds lock num, wait for release(num)
    mark lock num as held
    then server sends "OK" reply to client
  release(num):
    mark lock num as not held
    wake up any waiting acquires

we supply you with an RPC library and some demo client/server code
  you have to add locking RPCs -- acquire and release

RPC library simplifies client/server communication
  overall structure:
    client app
    client stubs
    rpcc (one for each server client talks to)
    RPC library
    ... network
    RPC library
    rpcs (just one)
    server handlers

lock_demo.cc
```

```
  new lock_client(dst) -- create client stubs, connect to server
  lc->stat(1) -- looks like an ordinary C++ method call

lock_client.cc
  new rpcc(dst) -- creates connection to server
  each stub method (stat, acquire, release):
    cl->call(...)
    call() is part of RPC library
    sends msg to server: procedure #, arguments
    waits for reply
  you need to fill in acquire(), release()
    call cl->call()
  (really lock_client.cc should be generated automatically)

lock_smain.cc
  create rpcs (creates thread to listen for client msgs)
  register lock_server's handler functions
  you will have to register two new handlers
  (really lock_smain.cc should be generated automatically)

lock_server.cc
  handlers
  you will have to add two new handlers
  a table of locks (use C++ STL map class)
    map<lockID> -> held or not held
  AND (this is tomorrow's topic)
    acquire() must *wait* for a held lock
    release() must wake up waiting acquire()
    both must avoid using table at the same time

Why this arrangement?
  lock_demo and lock_server are "application logic"
    they don't know much about RPC
    almost as if lock_demo were directly calling lock_server
  lock_client and lock_smain know about RPC glue
  this arrangement keeps the real app logic from being polluted by RPC details
```