Distributed systems
[Fall 2009]

G22.3033-001

Lec 1: Course Introduction & Lab Intro
Know your staff

• Instructor: Prof. Jinyang Li (me)
  – jinyang@cs.nyu.edu
  – Office Hour: Tue 5-6pm (715 Bway Rm 708)

• TA: Bonan Min
  – min@cs.nyu.edu
  – Office Hour: TBA (715 Bway Rm 705)
Important addresses

• Class webpage:
  http://www.news.cs.nyu.edu/~jinyang/fa09
  – Check regularly for announcements, reading questions

• Sign up for class mailing list
  g22_3033_001_fa09@cs.nyu.edu
  – We will email announcements using this list
  – You can also email the entire class for questions, share information, find project member etc.

• Staff mailing list includes just me and Bonan
  dss-staff@cs.nyu.edu
This class will teach you …

• Basic tools of distributed systems
  – Abstractions, algorithms, implementation techniques
  – System designs that worked

• Build a real system!
  – Synthesize ideas from many areas to build working systems

• Your (and my) goal: address new (unsolved) system challenges
Who should take this class?

• Pre-requisite:
  – Undergrad OS
  – Programming experience in C or C++

• If you are not sure, do Lab1 asap.
Course readings

• No official textbook
• Lectures are based on (mostly) research papers
  – Check webpage for schedules
• Useful reference books
  – Principles of Computer System Design. (Saltzer and Kaashoek)
  – Distributed Systems (Tanenbaum and Steen)
  – Advanced Programming in the UNIX environment (Stevens)
  – UNIX Network Programming (Stevens)
Course structure

• Lectures
  – Read assigned papers before class
  – Answer reading questions, hand-in answers in class
  – Participate in class discussion

• 8 programming Labs
  – Build a networked file system with detailed guidance!

• Project (workload is equivalent to 2 labs)
  – Extend the lab file system in any way you like!
How are you evaluated?

- Class participation 10%
  - Participate in discussions, hand in answers
- Labs 45%
- Project 15%
  - In teams of 2 people
  - Demo in last class
  - Short paper (<=4 pages)
- Quizzes 30%
  - mid-term and final (90 minutes each)
Questions?
What are distributed systems?

- Examples?
  - Multiple hosts
  - A network cloud
  - Hosts cooperate to provide a unified service
Why distributed systems? for ease-of-use

• Handle geographic separation
• Provide users (or applications) with location transparency:
  – **Web**: access information with a few “clicks”
  – **Network file system**: access files on remote servers as if they are on a local disk, share files among multiple computers
Why distributed systems? for availability

• Build a reliable system out of unreliable parts
  – Hardware can fail: power outage, disk failures, memory corruption, network switch failures…
  – Software can fail: bugs, mis-configuration, upgrade …
  – To achieve 0.999999 availability, replicate data/computation on many hosts with automatic failover
Why distributed systems? for scalable capacity

- Aggregate resources of many computers
  - CPU: Dryad, MapReduce, Grid computing
  - Bandwidth: Akamai CDN, BitTorrent
  - Disk: Frangipani, Google file system
Why distributed systems?
for modular functionality

• Only need to build a service to accomplish a single task well.
  – Authentication server
  – Backup server.
Challenges

• System design
  – What is the right interface or abstraction?
  – How to partition functions for scalability?

• Consistency
  – How to share data consistently among multiple readers/writers?

• Fault Tolerance
  – How to keep system available despite node or network failures?
Challenges (continued)

- Different deployment scenarios
  - Clusters
  - Wide area distribution
  - Sensor networks

- Security
  - How to authenticate clients or servers?
  - How to defend against or audit misbehaving servers?

- Implementation
  - How to maximize concurrency?
  - What’s the bottleneck?
  - How to reduce load on the bottleneck resource?
A word of warning

A distributed system is a system in which I can’t do my work because some computer that I’ve never even heard of has failed.”

-- Leslie Lamport
Topics in this course
Case Study: Distributed file system

A distributed file system provides:
• location transparent file accesses
• sharing among multiple clients

```bash
$ ls /dfs
f1 f2
$ cat f2
$ echo "test" > f2
$ ls /dfs
f1 f2
```
A simple distributed FS design

- A single server stores all data and handles clients' FS requests.
Topic: System Design

• What is the right interface?
  – possible interfaces of a storage system
    • Disk
    • File system
    • Database

• Can the system handle a large user population?
  – E.g. all NYU students and faculty

• How to store peta-bytes of data?
  – Has to use more than 1 server
  – Idea: partition users across different servers
Topic: Consistency

• When C1 moves file f1 from /d1 to /d2, do other clients see intermediate results?
• What if both C1 and C2 want to move f1 to different places?
• To reduce network load, cache data at C1
  – If C1 updates f1 to f1’, how to ensure C2 reads f1’ instead of f1?
Topic: Fault Tolerance

• How to keep the system running when some file server is down?
  – Replicate data at multiple servers
• How to update replicated data?
• How to fail-over among replicas?
• How to maintain consistency across reboots?
Topic: Security

• Adversary can manipulate messages
  – How to authenticate?

• Adversary may compromise machines
  – Can the FS remain correct despite a few compromised nodes?
  – How to audit for past compromises?

• Which parts of the system to trust?
  – System admins? Physical hardware? OS? Your software?
Topic: Implementation

- The file server should serve multiple clients concurrently
  - Keep (multiple) CPU(s) and network busy while waiting for disk
- Concurrency challenge in software:
  - Server threads need to modify shared state:
    - Avoid race conditions
    - One thread’s request may dependent on another
      - Avoid deadlock and livelock
Intro to programming Lab:
Yet Another File System (yfs)
YFS is inspired by Frangipani

- Frangipani goals:
  - Aggregate many disks from many servers
  - Incrementally scalable
  - Automatic load balancing
  - Tolerates and recovers from node, network, disk failures
Frangipani Design

- serve file system requests
- use Petal to store data
- incrementally scalable with more servers

- aggregate disks into one big virtual disk
- interface: put(addr, data), get(addr)
- replicated for fault tolerance
- incrementally scalable with more servers
- ensure consistent updates by multiple servers
- replicated for fault tolerance
Frangipani security

- Simple security model:
  - Runs as a cluster file system
  - All machines and software are trusted!
Frangipani server implements FS logic

- Application program:
  creat(“/d1/f1”, 0777)
- Frangipani server:
  1. GET root directory’s data from Petal
  2. Find inode # or Petal address for dir “/d1”
  3. GET “/d1”s data from Petal
  4. Find inode # or Petal address of “f1” in “/d1”
  5. If not exists
     alloc a new block for “f1” from Petal
     add “f1” to “/d1”s data, PUT modified “/d1” to Petal
Concurrent accesses cause inconsistency

Server S1: Server S2:
... ...
GET "/d1" GET "/d1"
Find file “f1” in “/d1” Find file “f2” in “/d1”
If not exists If not exists
  ... ...
  PUT modified “/d1” PUT modified “/d1”

What is the final result of “/d1”? What should it be?
Solution: use a lock service to synchronize access

Server S1:  Server S2:

..  
LOCK("/d1")
GET "/d1"
Find file "f1" in "/d1"
If not exists

...  ...
LOCK("/d1")

PUT modified "/d1"
UNLOCK("/d1")

GET "/d1"
Putting it together

create ("/d1/f1")

1. LOCK "/d1"
2. GET "/d1"
3. PUT "/d1"
4. UNLOCK "/d1"
NFS (or AFS) architecture

- Simple clients
  - Relay FS calls to the server
  - LOOKUP, CREATE, REMOVE, READ, WRITE …
- NFS server implements FS functions
NFS messages for reading a file

Client

OPEN ("f", 0);

LOOKUP (dirfh, "f")

fh and file attributes

READ (fh, buf, n)

READ (fh, 0, n)

data and file attributes

Server

lookup "f" in directory dirfh

read from fh
Why use file handles in NSF msg, not file names?

Program 1 on client 1

CHDIR ("dir1");

\( fd = \text{OPEN} ("f", \text{READONLY}); \)

\( \text{READ} (fd, buf, n); \)

Program 2 on client 2

\( \text{RENAME} ("dir1", "dir2"); \)

\( \text{RENAME} ("dir3", "dir1"); \)

• What file does client 1 read?
  – Local UNIX fs: client 1 reads dir2/f
  – **NFS using filenames:** client 1 reads dir1/f
  – NFS using file handles: client 1 reads dir2/f

• File handles refer to actual file object, not names
## Frangipani vs. NFS

<table>
<thead>
<tr>
<th>Feature</th>
<th>Frangipani</th>
<th>NFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scale storage</td>
<td>Add Petal nodes</td>
<td>Buy more disks</td>
</tr>
<tr>
<td>Scale serving capacity</td>
<td>Add Frangipani servers</td>
<td>Manually partition FS namespace among multiple servers</td>
</tr>
<tr>
<td>Fault tolerance</td>
<td>Data is replicated On multiple Petal Nodes</td>
<td>Use RAID</td>
</tr>
</tbody>
</table>
YFS: simplified Frangipani

- Single extent server to store data
- Communication using remote procedure calls (RPC)
Lab schedule

- L1: lock server
  - Programming w/ threads
  - RPC semantics
- L2: basic file server
- L3: Sharing of files
- L4: Locking
- L5: Caching locks
- L6: Caching extend server+consistency
- L7: Paxos
- L8: Replicated lock server
- L9: Project: extend yfs

Due every week:

Due every other week:
L1: lock server

• Lock service consists of:
  – Lock server: grant a lock to clients, one at a time
  – Lock client: talk to server to acquire/release locks

• Correctness:
  – At most one lock is granted to any client

• Additional Requirement:
  – acquire() at client does not return until lock is granted