Introduction to Distributed Systems

Arvind Krishnamurthy
Today’s Lecture

- Introduction
- Course details
- RPCs
- Primary-backup systems (start discussion)
Distributed Systems are everywhere!

- Some of the most powerful services are powered using distributed systems
  - systems that span the world,
  - serve millions of users,
  - and are always up!
- ... but also pose some of the hardest CS problems
- Incredibly relevant today
What is a distributed system?

- multiple interconnected computers that cooperate to provide some service

- what are some examples of distributed systems?
Why distributed systems?

- Higher capacity and performance
- Geographical distribution
- Build reliable, always-on systems
What are the challenges in building distributed systems?
(Partial) List of Challenges

- Fault tolerance
  - different failure models, different types of failures
- Consistency/correctness of distributed state
- System design and architecture
- Performance
- Scaling
- Security
- Testing
We want to build distributed systems to be more scalable, and more reliable.

But it’s easy to make a distributed system that’s less scalable and less reliable than a centralized one!
Challenge: failure

- Want to keep the system doing useful work in the presence of partial failures
Consider a datacenter

- E.g., Facebook, Prineville OR
- 10x size of this building, $1B cost, 30 MW power
  - 200K+ servers
  - 500K+ disks
  - 10K network switches
  - 300K+ network cables
- What is the likelihood that all of them are functioning correctly at any given moment?
Typical first year for a cluster

• ~0.5 overheating (power down most machines in <5 mins, ~1-2 days to recover)
• ~1 PDU failure (~500-1000 machines suddenly disappear, ~6 hours to come back)
• ~1 rack-move (plenty of warning, ~500-1000 machines powered down, ~6 hours)
• ~1 network rewiring (rolling ~5% of machines down over 2-day span)
• ~20 rack failures (40-80 machines instantly disappear, 1-6 hours to get back)
• ~5 racks go wonky (40-80 machines see 50% packetloss)
• ~8 network maintenances (4 might cause ~30-minute random connectivity losses)
• ~12 router reloads (takes out DNS and external vips for a couple minutes)
• ~3 router failures (have to immediately pull traffic for an hour)
• ~dozens of minor 30-second blips for dns
• ~1000 individual machine failures
• ~thousands of hard drive failures
• slow disks, bad memory, misconfigured machines, flaky machines, etc

[Jeff Dean, Google, 2008]
At any given point in time, there are many failed components!

Leslie Lamport (c. 1990): “A distributed system is one where the failure of a computer you didn’t know existed renders your own computer useless”
Challenge: Managing State

- Question: what are the issues in managing state?
State Management

- Keep data available despite failures:
  - make multiple copies in different places

- Make popular data fast for everyone:
  - make multiple copies in different places

- Store a huge amount of data:
  - split it into multiple partitions on different machines

- How do we make sure that all these copies of data are consistent with each other?

- How do we do all of this efficiently?
Lot of subtleties

● Simple idea: make two copies of data so you can tolerate one failure

● We will spend a non-trivial amount of time this quarter learning how to do this correctly!
  ● What if one replica fails?
  ● What if one replica just thinks the other has failed?
  ● What if each replica thinks the other has failed?
The Two Generals Problem

- Two armies are encamped on two hills surrounding a city in a valley.

- The generals must agree on the same time to attack the city.

- Their only way to communicate is by sending a messenger through the valley, but that messenger could be captured (and the message lost).
The Two-Generals Problem

- No solution is possible!
- If a solution were possible:
  - it must have involved sending some messages
  - but the last message could have been lost, so we must not have really needed it
  - so we can remove that message entirely
- We can apply this logic to any protocol, and remove all the messages — contradiction
What does this have to do with distributed systems?
Distributed Systems are Hard

- Distributed systems are hard because many things we want to do are provably impossible.
  - **consensus**: get a group of nodes to agree on a value (say, which request to execute next)
  - be certain about which machines are alive and which ones are just slow
  - build a storage system that is always consistent and always available (the “CAP theorem”)

- We need to make the right assumptions and also resort to “best effort” guarantees
This Course

- Introduction to the major challenges in building distributed systems
- Will cover key ideas, algorithms, and abstractions in building distributed systems
- Will also cover some well-known systems that embody such as ideas
Topics

- Implementing distributed systems: system and protocol design
- Understanding the global state of a distributed system
- Building reliable systems from unreliable components
- Building scalable systems
- Managing concurrent accesses to data with transactions
- Abstractions for big data analytics
- Building secure systems from untrusted components
- Latest research in distributed systems
Course Components

- Readings and discussions of research papers (20%)
  - no textbook
  - online response to discussion questions — one or two paras
  - we will pick the best 7 out of 8 scores

- Programming assignments (80%)
  - building a scalable, consistent key-value store
  - three parts (if done as individuals) or four parts (if done as groups of two)
  - total of 5 slack days with no penalty
Course Staff

- Instructor: Arvind
- TAs:
  - Kaiyuan Zhang
  - Paul Yau
- Contact information on the class page
Canvas

- Link on class webpage
- Post responses to weekly readings
- Please use Canvas “discussions” to discuss/clarify the assignment details
- Upload assignment submissions
Remote Procedure Call

- How should we communicate between nodes in a distributed system?
- Could communicate with explicit message patterns
  - But that could be too low-level
- RPC is a communication abstraction to make programming distributed systems easier
Common Pattern: Client/server

- Client requires an operation to be performed on a server and desires the result
- RPC fits this design pattern:
  - hides most details of client/server communication
  - client call is much like ordinary procedure call
  - server handlers are much like ordinary procedures
Local Execution

```c
float balance(int accountID) {
    return balance[accountID];
}

void deposit(int accountID, float amount) {
    balance[accountID] += amount
    return OK;
}

client() {
    deposit(42, $50.00);
    print balance(42);
}
```
Hard-coded Distributed Protocol

```plaintext
request "balance" = 1 {
  arguments {
    int accountID (4 bytes)
  }
  response {
    float balance (8 bytes);
  }
}

request "deposit" = 2 {
  arguments {
    int accountID (4 bytes)
    float amount (8 bytes)
  }
  response {
  }
}
```
Hard-coding Client/Server

```java
client() {
    s = socket(UDP)
    msg = {2, 42, 50.00} // marshalling
    send(s, server_address, msg)
    response = receive(s)
    check response == "OK"

    msg = {1, 42}
    send(s -> server_address, msg)
    response = receive(s)
    print "balance is" + response
}

server() {
    s = socket(UDP)
    bind s to port 1024
    while (1) {
        msg, client_addr = receive(s)
        type = byte 0 of msg
        if (type == 1) {
            account = bytes 1-4 of msg // unmarshalling
            result = balance(account)
            send(s -> client_addr, result)
        } else if (type == 2) {
            account = bytes 1-4 of msg
            amount = bytes 5-12 of msg
            deposit(account, amount)
            send(s -> client_addr, "OK")
        }
    }
}
```

Question: Why is this a bad approach to developing systems?
RPC Approach

- Compile high level protocol specs into stubs that do marshalling/unmarshalling

- Make a remote call look like a normal function call
RPC Approach
RPC hides complexity

```c
float balance(int accountID) {
    return balance[accountID];
}

void deposit(int accountID, float amount) {
    balance[accountID] += amount
    return OK;
}

client() {
    RPC_deposit(server, 42, $50.00);
    print RPC_balance(server, 42);
}
```
Question: is the complexity all gone?

what are the issues that we still would have to deal with?
Dealing with Failures

- Client failures
- Server failures
- Communication failures

Client might not know when failure happened

E.g., client never sees a response from the server — server could have failed before or after handling the message
At-least-once RPC

- Client retries request until it gets a response
- Implications:
  - requests might be executed twice
  - might be okay if requests are idempotent
Alternative: at-most-once

- Include a unique ID in every request
- Server keeps a history of requests it has already answered, their IDs, and the results
- If duplicate, server resends result

- Question: how do you guarantee uniqueness of IDs?
- Question: how can we garbage collect the history?
First Assignment

- Implement RPCs for a key-value store
- Simple assignment — goal is to get you familiar with the framework
- Due on 1/16 at 5pm
Primary-Backup Replication

- Widely used
- Reasonably simple to implement
- Hard to get desired consistency and performance
- Will revisit this and consider other approaches later in the class
Fault Tolerance

- we'd like a service that continues despite failures!
- available: still useable despite *some class of* failures
- strong consistency: act just like a single server to clients
- very useful!
- very hard!
Core Idea: replication

- Two servers (or more)
- Each replica keeps state needed for the service
- If one replica fails, others can continue
Key Questions

- What state to replicate?
- How does replica get state?
- When to cut over to backup?
- Are anomalies visible at cut-over?
- How to repair/re-integrate?
Two Main Approaches

- State transfer
  - "Primary" replica executes the service
  - Primary sends [new] state to backups

- Replicated state machine
  - All replicas execute all operations
  - If same start state, same operations, same order, deterministic → then same end state

- There are tradeoffs: complexity, costs, consistency
VMware’s FT Virtual Machines

- Whole-system replication
- Completely transparent to applications and clients
- High availability for any existing software

Failure model:
- Independent hardware faults
- Site-wide power failure

Limited to uniprocessor VMs
Overview

- two machines, primary and backup
- shared-disk for persistent storage
- back-up in "lock step" with primary
  - primary sends all inputs to backup
  - outputs of backup are dropped
- heart beats between primary and backup
  - if primary fails, start backup executing!
Challenges

- Making it look like a single reliable server
- How to avoid two primaries? ("split-brain syndrome")
- How to make backup an exact replica of primary
- What inputs must send to backup?
- How to deal with non-determinism?
Technique 1: Deterministic Replay

- Goal: make x86 platform deterministic
  - idea: use hypervisor to make virtual x86 platform deterministic

- Log all hardware events into a log
  - clock interrupts, network interrupts, i/o interrupts, etc.
  - for non-deterministic instructions, record additional info
    - e.g., log the value of the time stamp register
  - on replay: return the value from the log instead of the actual register
Deterministic Replay

- **Replay:** deliver inputs in the same order, at the same instructions
  - if during recording delivered clock interrupt at nth instr.
  - during replay also deliver the interrupt at the nth instr.

- **Given an event log, deterministic replay recreates VM**
  - hypervisor delivers first event
  - lets the machine execute to the next event
  - using special hardware registers to stop the processor at the right instruction
  - OS runs identical, applications runs identical

- **Limitation:** cannot handle multicore processors and interleaving
Applying Deterministic Replay to VM-FT

- Hypervisor at primary records
  - Sends log entries to backup over logging channel
- Hypervisor at backup replays log entries
  - We need to stop virtual x86 at instruction of next event
  - We need to know what is the next event
  - backup lags behind one event
Example

- Primary receives network interrupt
  - hypervisor forwards interrupt plus data to backup
  - hypervisor delivers network interrupt to OS kernel
  - OS kernel runs, kernel delivers packet to server
  - server/kernel write response to network card
  - hypervisor gets control and puts response on the wire

- Backup receives log entries
  - backup delivers network interrupt
  - ...
  - hypervisor does *not* put response on the wire
  - hypervisor ignores local clock interrupts
Technique 2: FT Protocol

- Primary delays any output until the backup acks
  - Log entry for each output operation
  - Primary sends output after backup acked receiving output operation

- Performance optimization:
  - primary keeps executing past output operations
  - buffers output until backup acknowledges
Questions

● Why send output events to backup and delay output until backup has acked?

● What happens when primary fails after receiving network input but before sending a corresponding log entry to backup?

● Can the same output be produced twice?
Design Space

- Active or passive replicas
- Symmetric replicas or primary-backup
- Replicate commands or low-level inputs
Lab Framework

- Designed with the following requirements in mind:
  - single machine, centralized orchestration
  - simulate arbitrary network behavior
  - allow for model checking, visualization

- First lab:
  - introduce the framework, understand “client” and “timeout”

- Second and subsequent labs:
  - all interactions through messages
  - you have complete control over everything