CS 677: Big Data

Fault Tolerance and Consensus

Lecture 6

Fault Tolerance + Consensus?

- It may seem a bit odd to see these two topics presented at the same time
  - "handling failures and agreeing on stuff"
- The reason is simple: the way we handle failures in distributed systems is with some type of **replication**
  - There are many ways to achieve replication
- Once we have replicas of data stored at different nodes, we can have **diverging views** of that data
  - Maybe a node goes down, misses some updates, and comes back up
  - We need to find ways to get them to all agree, even when failures occur

Today's Schedule

- Failures and Replication
- Reaching Consensus
- Consensus as a Service

Today's Schedule

- **Failures and Replication**
- Reaching Consensus
- Consensus as a Service

The Great Unknown

- It's hard to be sure about anything
  - True in general, but even more true with distributed systems
- Is a node down, or is the network slow?
- Did we shut the service down, or did it crash?
- Is the system in a steady state?
- If a network breaks into several partitions and nobody is around to hear it, does it make a sound?

Being a Total Failure

- Failures are very common when dealing with large datasets
  - Very common? More like unavoidable
- If tornadoes, space pirates, and earthquakes are all hitting your datacenter at the same time, what to do?
- ...
- What if we just make lots of copies of everything?

Replication

- Maintaining replicas is a great way to make our systems resilient to failures
- The general rule: create **three** replicas for each object you're storing
  - At least one of those replicas should be in a different physical location, if possible
- We can also leverage replicas as a cache to improve performance
  - If a node is closer, has less load, etc. then we can use it instead of the original copy

But isn't that Wasteful?

- Yep, now if you have a 1 TB dataset, it takes 3 TB
- Still, you do benefit from having more locations for each file
  - Increases **data locality** for computations
- The original GFS design stored multiple copies of each file, but later versions took advantage of parity files to decrease the space wasted
  - Used for error correction on media such as CD-ROMs, but also allows reconstruction of missing file chunks

Parity Algorithm

- Let's imagine we have a 100 MB file broken into 5 chunks (20 MB per chunk):

1	0	1	1	1	0	0	1	.	.	.	0
0	1	1	0	0	0	1	1	.	.	.	1
0	1	0	1	0	0	0	0	.	.	.	0
0	1	1	1	1	0	1	0	.	.	.	0
1	0	0	1	1	0	1	1	.	.	.	0

- If we want to be able to recover a missing chunk, we can perform an `XOR` over each bit to produce a *parity stripe*

Parity Chunk

	1	0	1	1	1	0	0	1	.	.	.	0
	0	1	1	0	0	0	1	1	.	.	.	1
	0	1	0	1	0	0	0	0	.	.	.	0
	0	1	1	1	1	0	1	0	.	.	.	0
	1	0	0	1	1	0	1	1	.	.	.	0
XOR	0	1	1	0	1	0	1	1	.	.	.	1

- Now if we lose any of the original chunks, we can use the parity chunk to rebuild it
  - (as long as all the other chunks are available)

Faulty Hardware

- Entire servers aren't the only thing that can fail
- Hard disk drives have many awful failure cases
  - Bad sectors can develop over time
- SSD cells can wear out
- RAM can have silent bit flips
  - Your amazing file gets corrupted in memory, then you save it to the disk...

File Integrity

- Most file systems don't actually have safeguards against bit rot and silent corruption
  - Seriously.
- ZFS and a few other file systems can maintain checksums for your files
  - And use parity data to reconstruct them in the case of a failure
- We need to be able to detect and repair corrupted files

Thought Experiment: Replication

We had one problem: fault tolerance. If we solve it with replication, what problems do we have now?

Managing Replicas

- Any time we start replicating data across multiple machines, things start to get complicated
- What happens when the replicas get modified at the same time?
  - Approach #1: Figure out the latest modification and use that as the "real" state of the replica
    - But how do we synchronize time between machines?
  - Approach #2: Distributed transaction support
    - Like what you might see in relational databases
    - Downside: latency from coordination and locking

Reaching Consensus

- Solving this problem with replicas is just one example of coming to a **consensus** in distributed systems
- Some other examples:
  - Clock synchronization, broadcasting, leader election
- Reaching a consensus can be difficult due to:
  - Heterogeneity
  - Geography (...latency)
  - Hardware **and** software failures

CAP Theorem [1/3]

- Deals with the guarantees that can be provided by distributed systems, especially during failures
- Observed by Eric Brewer
  - Co-founder of Inktomi
    - Search engine tech, ISP software
  - Professor at UC Berkeley
- Formalized in 2002 with a proof by Gilbert and Lynch
  - *Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services.* SIGACT. 2002.

CAP Theorem [2/3]

- **C**onsistency:
  All nodes see the same data.

- **A**vailability:
  A partial failure does not stop the system.

- **P**artition Tolerance:
  The system can handle network partitions.

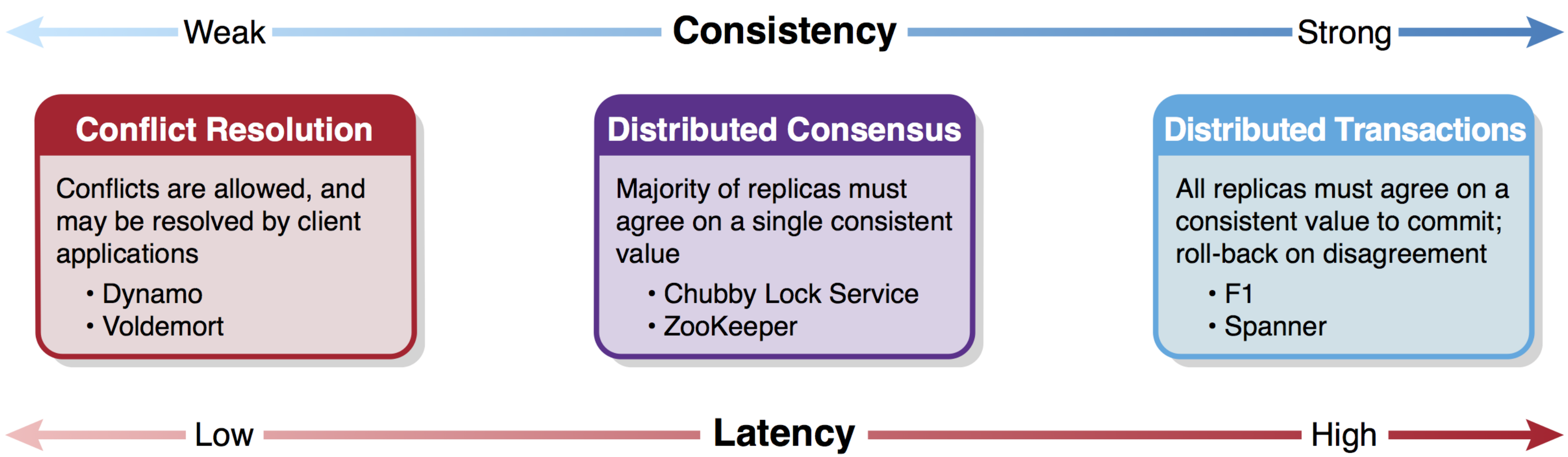
CAP Theorem [3/3]

- **Important**: this isn't a "pick two of the three" kind of situation
  - A mistake that is made frequently
- Rather, the CAP theorem describes what a system does when it encounters a network failure (partition)
- If everything is operating normally, the system can provide both high availability **and** consistency

CAP Classifications

- AP systems: highly available
  - Can result in inconsistent views of the dataset
  - Shopping cart
- CP systems: highly consistent
  - **Can** experience downtime if a partition occurs
    - That's okay, because we're assuming it's better to be offline than cause inconsistencies!
  - Billing system

Consistency-Latency Tradeoff



Today's Schedule

- Failures and Replication
- **Reaching Consensus**
- Consensus as a Service

Reaching Consensus

- There are two popular ways to get nodes to agree on something:
  - Paxos
  - Raft
- We're not a distributed systems class, so we won't go into these in depth (or build them)
- Advice for 99% of situations: don't invent your own algorithm, ignore Paxos, and use a Raft library

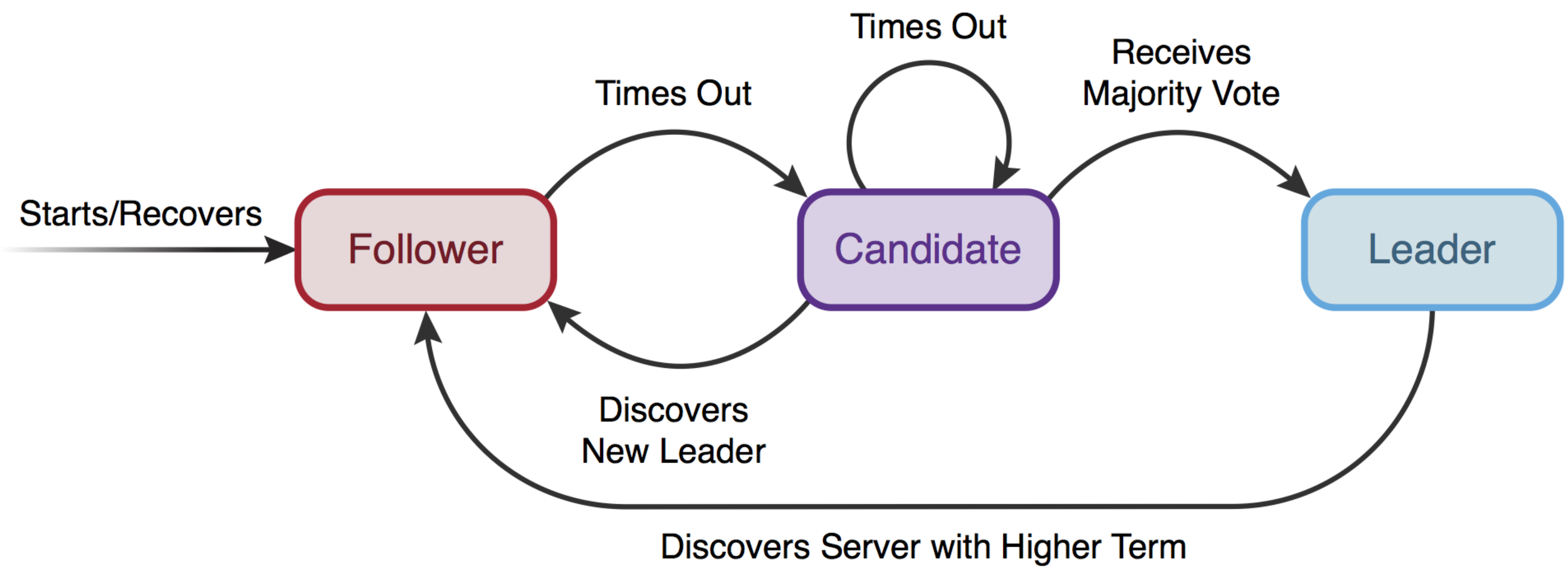
Paxos

- Described in The Part Time Parliament by Leslie Lamport
- Describes a fictional parliamentary consensus protocol used by legislators in Paxos, Greece
  - Took around 10 years to get published… it was a bit unconventional
- Used frequently to achieve distributed consensus
- Really, really hard to get right

Raft

- Raft is an attempt to build a more understandable consensus algorithm
- Each component can be explained in isolation
  - Leader, candidate, follower
- Uses **strong leaders**
  - One leader per term
  - When a failed node comes back up, it assumes that it is a follower and waits for a timeout rather than trying to become a leader immediately
- Each leader election increments the term number

Raft: Components and Flow



Today's Schedule

- Failures and Replication
- Reaching Consensus
- **Consensus as a Service**

Zookeeper Atomic Broadcast

- Zookeeper is often used to coordinate between components and detect failures
- Supports **atomic broadcast**, where not only consensus must be reached but event ordering matters
  - ZAB
- Three phases: discovery, synchronization, broadcast

Chubby

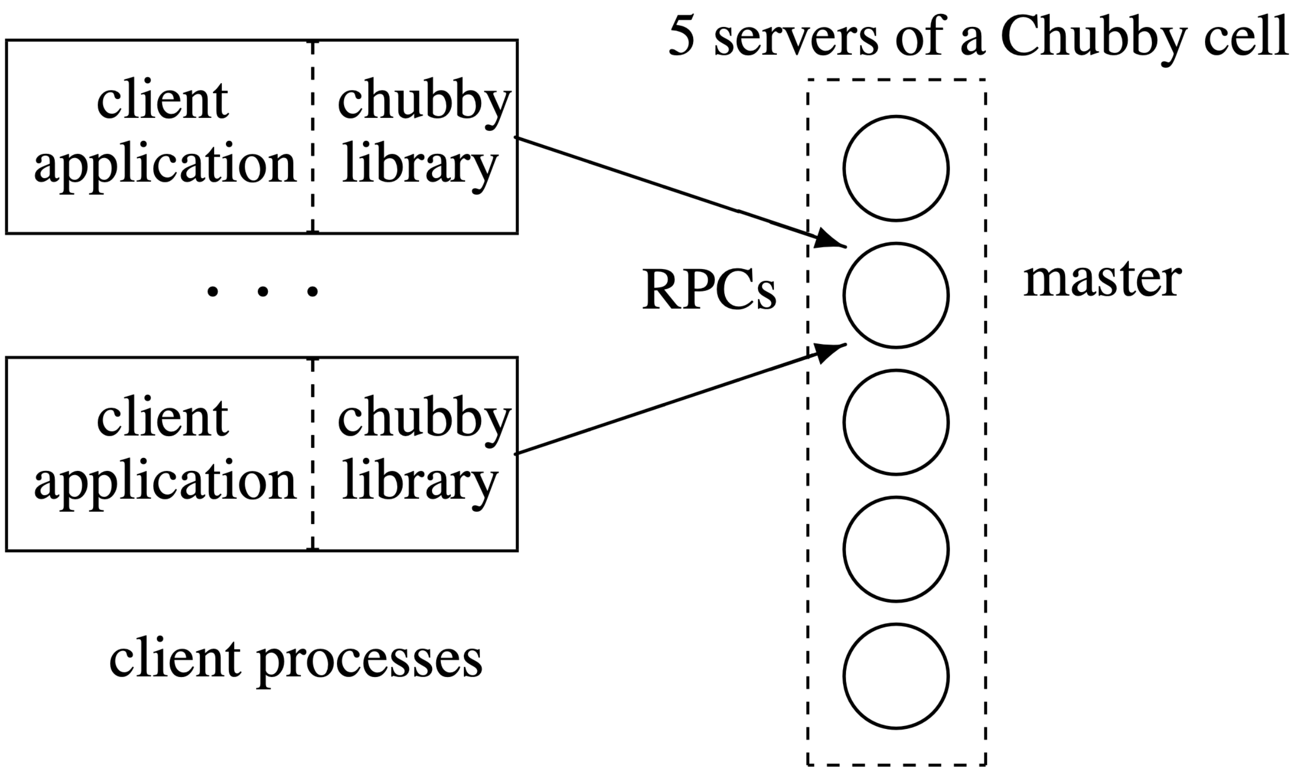
- Chubby is used to coordinate between components at Google
  - Locking, name services, config store
- Partially inspired by the VMS operating system
  - General purpose, global lock service
- Provides coarse-grained locking capabilities and simple storage facilities
  - Based on a file system model

Chubby

"Chubby is intended to operate within a single company, and so malicious denial-of-service attacks against it are rare. However, mistakes, misunderstandings, and the differing expectations of our developers lead to effects that are similar to attacks."

– Mike Burrows,
Google, Inc.,
The Chubby lock service for loosely-coupled distributed systems

Overview


5 servers of a Chubby cell
client application
chubby library
. . .
client application
chubby library
client processes
RPCs
master

File System Interface

- An example: `/ls/foo/wombat/pouch`
- `ls` – 'lock service'
- `foo` – the chubby **cell**, or instance of the system
  - Found via DNS lookup
- `wombat/pouch` – directory and file name
  - Files are just arrays of bytes

Abusive Clients

- As mentioned, incorrectly using Chubby is similar to an attack
- Initially, the system had no storage quotas
  - Not intended for a data store
  - Used for one anyway… 1.5 MB file rewritten for **every** client action
- Publish/subscribe
  - Can be used to publish changes, but not the intended use case

Lessons Learned

- Developers rarely consider availability
  - Chubby outages have caused cascading effects!
- Be careful with API design expectations
  - The system provides an event notification when a master failover occurs
    - Should help developers know that they need to verify the most recent actions
    - Instead, most applications decided to just crash
- Developers want to use their own favorite language

Call Me Maybe: Jepsen

- For a long time, storage systems made all kinds of outlandish claims
- Check out *Jepsen* by Kyle Kingsbury:
  - https://aphyr.com/tags/jepsen
  - https://jepsen.io
- Breaks down systems' consistency claims
  - Even includes illustrations!