Review

SWE 622, Spring 2017
Distributed Software Engineering
GMU SWE 622 HW 5 Scores (max possible = 80)
What do we want from Distributed Systems?

- Scalability
- Performance
- Latency
- Availability
- Fault Tolerance

“Distributed Systems for Fun and Profit”, Takada
Distributed Systems Goals

- **Scalability**
- Performance
- Latency
- Availability
- Fault Tolerance

“the ability of a system, network, or process, to handle a growing amount of work in a capable manner or its ability to be enlarged to accommodate that growth.”

“Distributed Systems for Fun and Profit”, Takada
Distributed Systems Goals

• Scalability
• Performance
• Latency
• Availability
• Fault Tolerance

“is characterized by the amount of useful work accomplished by a computer system compared to the time and resources used.”
Distributed Systems Goals

- Scalability
- Performance
- **Latency**
- Availability
- Fault Tolerance

“The state of being latent; delay, a period between the initiation of something and the it becoming visible.”
Distributed Systems Goals

- Scalability
- Performance
- Latency
- **Availability**
- Fault Tolerance

“the proportion of time a system is in a functioning condition. If a user cannot access the system, it is said to be unavailable.”

Availability = \( \frac{\text{uptime}}{\text{uptime} + \text{downtime}} \).

Often measured in “nines”

<table>
<thead>
<tr>
<th>Availability %</th>
<th>Downtime/year</th>
</tr>
</thead>
<tbody>
<tr>
<td>90%</td>
<td>&gt;1 month</td>
</tr>
<tr>
<td>99%</td>
<td>&lt; 4 days</td>
</tr>
<tr>
<td>99.9%</td>
<td>&lt; 9 hours</td>
</tr>
<tr>
<td>99.99%</td>
<td>&lt;1 hour</td>
</tr>
<tr>
<td>99.999%</td>
<td>5 minutes</td>
</tr>
<tr>
<td>99.9999%</td>
<td>31 seconds</td>
</tr>
</tbody>
</table>
Distributed Systems Goals

- Scalability
- Performance
- Latency
- Availability
- Fault Tolerance

“What kind of faults?”

Disks fail
Power supplies fail
Power goes out
Networking fails
Security breached
Datacenter goes offline
More machines, more problems

• Say there’s a 1% chance of having some hardware failure occur to a machine (power supply burns out, hard disk crashes, etc)

• Now I have 10 machines
  • Probability(at least one fails) = 1 - Probability(no machine fails) = 1-(1-.01)^{10} = 10%
  • 100 machines -> 63%
  • 200 machines -> 87%
  • So obviously just adding more machines doesn’t solve fault tolerance
Designing and Building Distributed Systems

To help design our algorithms and systems, we tend to leverage abstractions and models to make assumptions

Generally: Stronger assumptions -> worse performance
Weaker assumptions -> more complicated

System model
- Asynchronous
- Synchronous

Failure Model
- Crash-fail
- Byzantine

Consistency Model
- Eventual
- Sequential

Generally: Stronger assumptions -> worse performance
Weaker assumptions -> more complicated
Redis Replication

1: Write key “foo” = “bar”

2: Acknowledge write

3: Send update

Asynchronous updates, weak consistency model, highly available
CAP Theorem

• Pick two of three:
  • Consistency: All nodes see the same data at the same time (strong consistency)
  • Availability: Individual node failures do not prevent survivors from continuing to operate
  • Partition tolerance: The system continues to operate despite message loss (from network and/or node failure)

• You can not have all three, ever*
  • If you relax your consistency guarantee (from strong to weak), then you can probably guarantee that
Choosing a consistency model

- Sequential consistency
  - All over - it’s the most intuitive
- Causal consistency
  - Increasingly useful
- Eventual consistency
  - Very popular in industry and academia
  - File synchronizers, Amazon’s Bayou and more
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount){
    if(from.balance >= amount)
    {
        from.balance = from.balance - amount;
        to.balance = to.balance + amount;
        return true;
    }
    return false;
}
```

What can go wrong here?
Agreement

• In distributed systems, we have multiple nodes that need to all agree that some object has some state

• Examples:
  • Who owns a lock
  • Whether or not to commit a transaction
  • The value of a clock
Properties of Agreement

• Safety (correctness)
  • All nodes agree on the same value (which was proposed by some node)

• Liveness (fault tolerance, availability)
  • If less than N nodes crash, the rest should still be OK
2PC Event Sequence

Coordinator

Transaction state:
- prepared
- committed
- done

Can you commit?

Yes

OK, commit

OK I committed

Participant

Local state:
- prepared
- uncertain
- committed
Timeouts in 2PC

- Example:
  - Coordinator times out waiting for Goliath National Bank’s response
  - Bank times out waiting for coordinator’s outcome message

- Causes?
  - Network
  - Overloaded hosts
  - Both are very realistic…
Coordinator Timeouts

• If coordinator times out waiting to hear from a bank
  • Coordinator hasn’t sent any commit messages yet
  • Can safely abort - send abort message
  • Preserves correctness, sacrifices performance (maybe didn’t need to abort!)
• If either bank decided to commit, it’s fine - they will eventually abort
Handling Bank Timeouts

- What if the bank doesn’t hear back from coordinator?
- If bank voted “no”, it’s OK to abort
- If bank voted “yes”
  - It can’t decide to abort (maybe both banks voted “yes” and coordinator heard this)
  - It can’t decide to commit (maybe other bank voted yes)
- Does bank just wait for ever?
Handling Bank Timeouts

• Can resolve SOME timeout problems with guaranteed correctness in event bank voted “yes” to commit

• Bank asks other bank for status (if it heard from coordinator)

• If other bank heard “commit” or “abort” then do that

• If other bank didn’t hear
  • but other voted “no”: both banks abort
  • but other voted “yes”: no decision possible!
2PC Timeouts

• We can solve a lot (but not all of the cases) by having the participants talk to each other

• But, if coordinator fails, there are cases where everyone stalls until it recovers

• Can the coordinator fail?… yes

• Hence, 2PC does not guarantee liveness: a single node failing can cause the entire set to fail
3 Phase Commit

- Goal: Eliminate this specific failure from blocking liveness
3 Phase Commit

• Goal: Avoid blocking on node failure

• How?
  • Think about how 2PC is better than 1PC
    • 1PC means you can never change your mind or have a failure after committing
    • 2PC still means that you can’t have a failure after committing (committing is irreversible)
  • 3PC idea:
    • Split commit/abort into 2 sub-phases
      • 1: Tell everyone the outcome
      • 2: Agree on outcome
3PC Example

Coordinator

Participants (A,B,C,D)

Soliciting votes

Status: Uncertain

Commit authorized (if all yes)

Status: Prepared to commit

Done

Status: Committed
3PC Crash Handling

- Can B/C/D reach a safe decision...
  - If any one of them has received preCommit?
    - YES! Assume A is dead. When A comes back online, it will recover, and talk to B/C/D to catch up.
    - Consider equivalent to in 2PC where B/C/D received the “commit” message and all voted yes
3PC Crash Handling

• Can B/C/D reach a safe decision…
  • If NONE of them has received preCommit?
    • YES! It is safe to abort, because A can not have committed (because it couldn’t commit until B/C/D receive and acknowledge the pre-commit)
  • This is the big strength of the extra phase over 2PC
• Summary: Any node can crash at any time, and we can always safely abort or commit.
3PC Timeout Handling

Coordinator      Participants (A,B,C,D)

Soliciting votes
Timeout causes abort
Commit authorized (if all yes)
Timeout causes abort
Done

Status: Uncertain
Timeout causes abort

Status: Prepared to commit
Timeout causes commit

Status: Committed
Does 3PC guarantee consensus?

• Reminder, that means:
  • Liveness (availability)
    • Yes! Always terminates based on timeouts
  • Safety (correctness)
    • Hmm…
Partitions

Coordinator

Soliciting
Authorized

Prepared to commit

Network Partition!!!

Timeout behavior: commit!

Participant A: Yes, Committed
Participant B: Yes, Aborted
Participant C: Yes, Aborted
Participant D: Yes, Aborted

Timeout behavior: abort

Participant B: Yes, Uncertain
Participant C: Yes, Uncertain
Participant D: Yes, Uncertain

J. Bell

GMU SWE 622 Spring 2017
FLP - Intuition

- Why can’t we make a protocol for consensus/agreement that can tolerate both partitions and node failures?
- To tolerate a partition, you need to assume that **eventually** the partition will heal, and the network will deliver the delayed packages.
- But the messages might be delayed **forever**.
- Hence, your protocol would not come to a result, until **forever** (it would not have the **liveness** property).
Partition Tolerance

• Key idea: if you always have an odd number of nodes…
• There will always be a minority partition and a majority partition
• Give up processing in the minority until partition heals and network resumes
• Majority can continue processing
Partition Tolerant Consensus Algorithms

• Decisions made by **majority**

• Typically a fixed coordinator (**leader**) during a time period (**epoch**)

• How does the leader change?
  
  • Assume it starts out as an arbitrary node
  
  • The leader sends a heartbeat
  
  • If you haven’t heard from the leader, then you **challenge** it by advancing to the next epoch and try to elect a new one
  
  • If you don’t get a **majority** of votes, you don’t get to be leader
  
  • …hence no leader in a minority partition
ZooKeeper - Data Model

• Provides a hierarchical namespace
• Each node is called a znode
• ZooKeeper provides an API to manipulate these nodes
ZooKeeper - Guarantees

• **Liveness guarantees**: if a majority of ZooKeeper servers are active and communicating the service will be available

• **Durability guarantees**: if the ZooKeeper service responds successfully to a change request, that change persists across any number of failures as long as a quorum of servers is eventually able to recover
ZooKeeper - Applications

- Distributed locks
- Group membership
- Leader election
- Shared counters
Failure Handling in ZK

• Just using ZooKeeper does not solve failures
• Apps using ZooKeeper need to be aware of the potential failures that can occur, and act appropriately
• ZK client will guarantee consistency if it is connected to the server cluster
Failure Handling in ZK

1. Client creates an event.
2. ZK2 has a network problem.
3. Client reconnects to ZK3.
4. Client reissues the create event to ZK3.
Failure Handling in ZK

• If in the middle of an operation, client receives a `ConnectionLossException`
• Also, client receives a `disconnected message`
• Clients can’t tell whether or not the operation was completed though - perhaps it was completed before the failure
• Clients that are disconnected can not receive any notifications from ZK
Dangers of ignorance

Client 1 creates /leader and then gets disconnected.

ZK receives a notification that /leader is dead.

Client 2 creates /leader and becomes the leader.

Client 2 reconnects and discovers it is no longer the leader.
Dangers of ignorance

• Each client needs to be aware of whether or not its connected: when disconnected, can not assume that it is still included in any way in operations

• By default, ZK client will NOT close the client session because it's disconnected!
  • Assumption that eventually things will reconnect
  • Up to you to decide to close it or not
ZK: Handling Reconnection

• What should we do when we reconnect?
• Re-issue outstanding requests?
  • Can't assume that outstanding requests didn't succeed
  • Example: create /leader (succeed but disconnect), re-issue create /leader and fail to create it because you already did it!
GFS

- Hundreds of thousands of regular servers
- Millions of regular disks
- Failures are normal
  - App bugs, OS bugs
  - Human Error
  - Disk failure, memory failure, network failure, etc
- Huge number of concurrent reads, writes
GFS Architecture
GFS Summary

• Limitations:
  • Master is a huge bottleneck
  • Recovery of master is slow
• Lots of success at Google
• Performance isn't great for all apps
• Consistency needs to be managed by apps
• Replaced in 2010 by Google's Colossus system - eliminates master
Distributing Computation

• Can't I just add 100 nodes and sort my file 100 times faster?

• Not so easy:
  • Sending data to/from nodes
  • Coordinating among nodes
  • Recovering when one node fails
  • Optimizing for locality
  • Debugging
Distributing Computation

• Lots of these challenges re-appear, regardless of our specific problem
  • How to split up the task
  • How to put the results back together
  • How to store the data
• Enter, MapReduce
MapReduce

• A programming model for large-scale computations
  • Takes large inputs, produces output
  • No side-effects or persistent state other than that input and output
• Runtime library
  • Automatic parallelization
  • Load balancing
  • Locality optimization
  • Fault tolerance
MapReduce: Divide & Conquer

Big Data (lots of work)

Partition

w1
- worker
  - r1

w2
- worker
  - r2

w3
- worker
  - r3

w4
- worker
  - r4

w5
- worker
  - r5

Combine

Result
MapReduce: Fault Tolerance

• Ideally, fine granularity tasks (more tasks than machines)

• On worker-failure:
  • Re-execute completed and in-progress map tasks
  • Re-executes in-progress reduce tasks
  • Commit completion to master

• On master-failure:
  • Recover state (master checkpoints in a primary-backup mechanism)
Hadoop + ZooKeeper

Note - this is why ZK is helpful here: we can have the ZK servers partitioned *too* and still tolerate it the same way.
Where do we find data?

• What's bad with the single master picture?
• HDFS/GFS leverage the fact that there is relatively little metadata, lots of data (e.g. few files, each file is large)
• What if there is really only metadata?
• How can we build a system with high performance without having this single server that knows where data is stored?
Hashing

- The last one mapped every input to a different hash
- Doesn't have to, could be collisions
## Hashing for Partitioning

<table>
<thead>
<tr>
<th>Input</th>
<th>Hash Result</th>
<th>Server ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Some big long piece of text or database key</td>
<td>hash() = 900405 % 20 = 5</td>
<td>5</td>
</tr>
</tbody>
</table>
Consistent Hashing

It is relatively smooth: adding a new bucket doesn't change that much

Add new bucket: only changes location of keys 7,8,9,10

Delete bucket: only changes location of keys 1,2,3
Hi, I signed on, I have files f1, f2, f3

Who has f1?

client 1 does

Can I have f1?

Here is f1

Doesn’t everything just look like GFS, even things that predated it? :)

Napster
Gnutella 1.0

Can I have f1?

where is f1?

client 1

client 2

c2 has f1

where is f1?

client 3

client 4

client 5
BitTorrent
Byzantine Generals Problem

Attack!

Don’t attack!

Attack!

Attack!
Oral BFT Example (n=4, m=1)

Commander

Lieutenant 1

Lieutenant 2

Lieutenant 3

x

y

z

x

z

y
Blockchains

• Solution: make it hard for participants to take over the network; provide rewards for participants so they will still participate

• Each participant stores the entire record of transactions as blocks

• Each block contains some number of transactions and the hash of the previous block

• All participants follow a set of rules to determine if a new block is valid
Blockchain's view of consensus

Miner 1:  
Miner 2:  
Miner 3:  

"Longest chain rule"
When is a block truly safe?
CFS

- What is the system model?
  - Synchronous replication (thanks to WAIT), but it has no rollback in event replication fails
  - We solve this by destroying the entire server if the replication fails
- Partition tolerant
CFS Fault Tolerance

Goal 1: Promote new master when master crashes
CFS Fault Tolerance

Disconnected

Disconnected

Disconnected

Slave

Master

Slave

Client

Client

OK
Goal 2: Promote new master when partitioned + have quorum, cease functioning in minority
CFS + Failures

Would a partition like this cause inconsistency?

No, if we WAIT for all replicas; Yes if we only wait for a quorum!