CS 138: Review session

Theophilus Benson
Final Exam Details

• Exam Details
  – Cumulative exam
  – Two hours
  – Closed book

• Exam Venue
  – 2pm on Monday May 14th
  – Location: Metcalf AUD
Purpose of this Review Session

• Cover highlights of the last three months
  – We can’t cover the whole class in 1.5 hours
  – Specify concepts not covered in the Final
Closing Remarks

• Distributed systems: art of providing consensus while tackling failures while providing high performance

• Failure V. Performance V. Correctness/consensus
  – Different type of failures → different implications (different detectors)
    • Mostly heart beats
  
  – Depends on the application: sometimes you don’t need linearizable (total ordering) on all events
    • Zookeeper: reads are not linearizable
    • Dynamo/Cassandra: reads/writes are causally consistent

  – Performance: Shard and distribute
    • Consistent hashing to find shards

• Final on Monday 5/14 at 2pm
Failure: Taxonomy and Classifications

- Failure types
  - Synchronous V. Asynch.

- Failure Types
  - Byzantine failures
Synchronous vs. Asynchronous

• Execution speed
  – synchronous: bounded
  – asynchronous: unbounded

• Message transmission delays
  – synchronous: bounded
  – asynchronous: unbounded

• Local clock drift rate:
  – synchronous: bounded
  – asynchronous: unbounded
Failures

• Omission failures
  – something doesn’t happen
    • process crashes
    • data lost in transmission
    • etc.

• Byzantine (arbitrary) failures
  – something bad happens
    • message is modified
    • message received twice
    • any kind of behavior, including malicious

• Timing failures
  – something takes too long
Detecting Crashes

• **Synchronous systems**
  – Timeouts

• **Asynchronous systems**
  – ?

• **Fail-stop**
  – an oracle lets us know
Partition and Scaling
(L2-L4)
Partitioning and Scaling

• Historical information
  – Napster, Gnutella

• P2P systems
  – Consistent Hashing
    • Replication/load-balancing
  – Chord
    • Adding/removing nodes
  – Tapestry
    • Adding/removing nodes

• Practical Use-cases
  – Facebook’s Cassandra
  – Amazon’s Dynamo
Chord - Overview

Each node maintains:
- Finger table
- Success
- Predecessor

Identifier ring over hash space $2^m$

Finger table for node at id $i$

<table>
<thead>
<tr>
<th>finger</th>
<th>node id</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\text{succ}(i)$</td>
</tr>
<tr>
<td>2</td>
<td>$\text{succ}(i + 2)$</td>
</tr>
<tr>
<td>$j$</td>
<td>$\text{succ}(i + 2^{j-1})$</td>
</tr>
</tbody>
</table>

node id = hash( node )

key id = hash( key )

chord images from Nirvan Tyagi
Publishing

Publishing == putting content into Tapestry

Content needs to be routed to the Root (node with closest ID)

Each object can have multiple root: use salting

- Step 1: pick random node
- Step 2: send publish request to random node
- Step 3: random node routes request to root (objectID)
- Step 4: root stores map of ObjectID to Client-IP
A Problem: Re-rooting and Race Conditions

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxxx</td>
<td>0121 128.148.158.13</td>
<td>1001 128.18.11.192</td>
<td>2130 128.118.225.127</td>
<td>3311 128.12.248.192.76</td>
</tr>
<tr>
<td>3xx</td>
<td>—</td>
<td>3120 128.162.253.247</td>
<td>—</td>
<td>3320 128.248.192.76</td>
</tr>
<tr>
<td>33x</td>
<td>—</td>
<td>3311 128.12.236.81</td>
<td>3320 128.248.192.76</td>
<td>—</td>
</tr>
<tr>
<td>332</td>
<td>3320 128.248.192.76</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

- Node 3322 is added
- Object 3321’s surrogate was 3320
  - now it’s 3322
  - what about all the location info that was stored assuming 3320?
Tapestry Issues

• How do you add/delete nodes?
  – How do you build the forwarding table

• What is surrogate routing?

• What are backpointers?
Challenges with Partition scheme

• Partition doesn’t account for
  – Load imbalance (some items are more popular)
  – Key imbalance (items aren’t evenly distributed to keys)

• Replication

• How to solve this?
  – Have a node appear in multiple locations (dynamoDB)
  – Salt the keys (Puddlestore)
  – Analyze the keyspace and Move lightly loaded nodes in the ring (Cassandra)
Redundancy/Replication: ReSalt V. Multiple DHTs
Applications of Consistent Hashing

• Dynamo and Cassandra

• Key differences
  – Chord: Finger Table
  – Tapestry: Routing Table
  – Dynamo/Cassandra: Each node has a list of all other nodes
    • Replication/load balancing strategy
Consensus

• Raft election
  – Candidate: request vote
  – Follower: at most one vote

• Distribute transactions
  – Coordinator: request transaction
  – Nodes: Commit/Abort

• General Current assumptions
  – Follower/Nodes don’t lie

• What happens if nodes can lie?
  – Byzantine Failures
Consensus Background

- Nodes start in *undecided state*
- Nodes propose states to each other
- Nodes converge on the same value
Consensus

• Setting: a group of processes, some correct, some faulty

• Two primitives, propose(v) and decide(v)

• If each correct process proposes a value:
  – **Termination**: every correct process eventually decides on a value
  – **Agreement**: if a correct process decides v, then all processes eventually decide v
  – **Integrity**: if a correct process decides v, then v was previously proposed by some process
Consensus: Snapshots
Snapshot Terminology

- A process’s state -> memory region
  - E.g., Variables used and defined

- Channel -> connection between two processes
  - Each channel is unidirectional
  - One channel from P1 to P2 and a different one from P2 to P1
  - Channel deliver messages in FIFO order
  - Channel does not lose messages
  - Channel does not fail

- Snapshot are done one at a time
  - Initiated by a global and external entity
Snapshot Rules

• Marker receiving rule for process $p_i$
  On $p_i$’s receipt of a marker message over channel $c$:
  \[if\] ($p_i$ has not yet recorded its state)
  it records its state
  it records the state of $c$ as the empty sequence
  it turns on recording of messages arriving over other channels
  \[else\]
  $p_i$ records the state of $c$ as the set of messages it has received over $c$ since it saved its state and before it received the marker over $c$

• Marker sending rule for process $p_i$
  After $p_i$ has recorded its state, for each outgoing channel $c$:
  $p_i$ sends one marker message over $c$ (before it sends any other messages over $c$)
Termination

• Process P has completed its part of the algorithm when it has processed markers on all input channels

• It sends its saved local state and channel histories to the initiator
  – the intent is that collection of local states form consistent cut
    • channel histories are the messages in transit at time of cut
Consensus: Replication
Passive Replication
• Total ordering
• Requires Raft like protocol

Active Replication
• Causal ordering
• Requires total ordered multicast

Lazy replication
• Causal ordering
• A special form of multicast
  – Each node periodically sends updates to a subset of nodes
  – The subset is chosen at random each time
Passive Replication: Raft

- Proposed by Ongaro and Ousterhout in 2014

- Five components
  - Leader election
  - Log replication
  - Safety
  - Client protocol
  - Membership changes

- Assumes crash failures (so no byzantine failures)
- No dependency on time for safety
  - But depends on time for availability
- Tolerates \((N-1)/2\) failures
Raft Summary

- Raft consensus protocol
  - Leader election
  - Log replication: append entries to end of file
  - Safety+ consistency: followers eventually have identical logs to the leader
    - Change to raft protocol: election and commitment
    - Neutralizing old leaders: using the term
  - Client protocol
    - Dealing with network partitions (and server crashes)
      - Leader step downs: if no majority heart beat
      - Maintain request IDs to ensure “exactly once”
  - Membership changes
    - Switch using two-phase commit
Server States

- At any given time, each server is either:
  - **Leader**: handles all client interactions, log replication
    - At most 1 viable leader at a time
  - **Follower**: completely passive (issues no RPCs, responds to incoming RPCs)
  - **Candidate**: used to elect a new leader

- Normal operation: 1 leader, N-1 followers
Heartbeats and Timeouts

- Servers start up as followers

- Followers expect to receive RPCs from leaders or candidates
  - Leaders must send **heartbeats** (**empty AppendEntries RPCs**) to maintain authority

- If **electionTimeout** elapses with no RPCs:
  - Follower assumes leader has crashed
  - Follower starts new election → follower becomes candidate
  - **Timeouts typically 100-500ms**
Gossip Summary

• Different consistency models
  – Linearizability, sequential

• Gossip Protocol
  – Models: Immediate, forced, causal
  – Challenges: bounding logs, trimming CIDs
Gossip Paper: Provides Different Models of Consistency

• Causal ordering

• Forced ordering
  – both causal and total order

• Immediate ordering
  – forced ordering with minimal delay
Gossip: lazy Replication

- A special form of multicast
  - Each node periodically sends updates to a subset of nodes
  - The subset is chosen at random each time

- How do we get forced updates? Or immediate updates?
  - Normal Gossip only provides casual ordering
  - Recall we want total ordering
  - Combination of Passive and Lazy: first passive and in background use lazy
Rough Outline
(Causal Ordering)

• **Query (getting a value)**
  – client sends request to one or more RMIs
  – respond when causally possible

• **Update (setting a value)**
  – client sends request to one or more RMIs
  – update and respond when causally possible
  – propagate changes to others via “gossip” messages
    • not specified how this is done
    • allows many possibilities
Active Replication

• Total ordered multicast
  – Mutual exclusion
Consensus: Transactions
Summary of Distributed Transaction

- Transactions
  - Atomic Commit of Transactions
  - Five desired properties

- Two Phase Commit
  - Analyzed state machines for different roles
  - Participant and Coordinator response to failures
  - Blocking issue
  - State machines

- Three Phase Commit
  - Attempt to eliminate Blocking issue
  - More complex protocol including more messages
  - Analyzed state machines for different roles
Transactions

- “ACID” property:
  - atomic
    - all or nothing
  - consistent
    - take system from one consistent state to another
  - isolated
    - have no effect on other transactions until committed
  - durable
    - persists
Atomic Commit Properties

• AC1: All participants that reach a (commit/abort) decision reach the same one

• AC2: A participant cannot reverse its decision

• AC3: The commit decision can be reached only if all participants agree

• AC4: If there are no failures and all participants vote yes, then decision will be commit

• AC5: For any execution, if all failures are repaired and no new failures occur for a sufficiently long interval, then all participants will reach a (commit/abort) decision
Two-Phase Commit

• **Phase 1**
  – coordinator prepares to commit:
    • asks participants to vote either “commit” or “abort”
    – participants respond appropriately

• **Phase 2**
  – coordinator decides outcome:
    • if all participants vote commit, outcome is commit, otherwise outcome is abort
    • outcome sent to all participants
  – participants do what they’re told
Two-Phase Commit

Coordinator

pariticipants

Decide to abort
Or commit based
On local state

Commit or Abort

Once a node responds. The node can’t change responses

Commit or Abort

Response

If all Commit
Then Commit
If any Abort
Then Abort

Make a change

Commit or Abort
Two-Phase Commit

Coordinator

pariticpants

If all Commit
Then Commit
If any Abort
Then Abort
(AC3, AC4)

Make a change

Commit or Abort

Response

Commit or Abort (AC1)

Commit or Abort (AC1)

Decide to abort
Or commit based
On local state

Once a node responds.
The node can’t change
responses (AC2)
Consensus: Byzantine Failures
Byzantine Lecture Recap

• Discussed consensus properties
  – Termination, integrity, agreement
  – Analyzed Raft, Distributed transactions, and total ordered multicast within this context

• Discussed message passing assumptions
  – Impact of hidden assumptions on byzantine problem

• Byzantine failures
  – Lack of consensus in 2f+1 scenarios
  – Demonstrate of consensus in 3f+1 scenarios
  – Impact of message passing assumptions on byzantine failures
Message-Passing Assumptions

• **Nodes only communicate through message-passing**
  – Every message sent is delivered correctly
  – The receiver of a message knows who sent it
  – The absence of a message can be detected

• **Hidden assumptions**
  – No message signing (encryption)
  – Nodes can only listen to messages to/from them
  – Nodes can only modify messages they send
Byzantine Generals Problem

• C1: All loyal lieutenant generals obey the same order

• C2: If the commanding general is loyal, then every loyal lieutenant general obeys the order she sends

• Note: if the commanding general is lying then Byzantine Generals doesn’t help reach consensus
<table>
<thead>
<tr>
<th></th>
<th>Termination</th>
<th>Integrity</th>
<th>Agreement</th>
</tr>
</thead>
<tbody>
<tr>
<td>consensus</td>
<td>Eventually set decision</td>
<td>All correct processes will have the same value</td>
<td>All correct process have the same decision</td>
</tr>
<tr>
<td>Byzantine general</td>
<td></td>
<td>If leading general is correct, then correct processes will have the same value</td>
<td></td>
</tr>
</tbody>
</table>
Byzantine V. Consensus

- **Consensus**
  - Survive $f$ failures
  - $N > 2f$

- **Byzantine general**
  - Survive $f$ failures
  - $N > 3f$
Distributed File System (L15)
Design Goals for Distributed File System

• Improve fault tolerance
  – No state at the server
  – Operations are idempotent

• Improve performance
  – Caching at client and server
    • Client: cache in client side to reduce overhead of using the network
    • Server: Cache in memory to reduce overhead of going to disk

• Interesting Design questions
  – What state to maintain at server or clients?
  – How to maintain cache coherence?
Distributed File System: Consistency vs. Performance

• Strict consistency is easy ...  
  – ... if all operations take place on server  
  – no client caching  

• Performance is good ...  
  – ... if all operations take place on client  
  – everything is cached on client  

• Put the two together ...  

∅  
  – or you can do opportunistic locking
Consistency Models

• Opportunistic Locking
  – Enables client to safely cache file and modify it
  – Improves performance

• Opportunistic V. Strict Locks

• NFS: weak or strict
  – Strick if locks are used

• CIFS: strict consistency
  – Uses opportunistic locks
  – Revoke + flush

• DCE: Strict consistency
  – Uses tokens
  – Similar revoke + flush policy as CIFS
Large Scale Production Systems
(L17-L18 L20-22)
Applications and Usecases
Production Systems

• DynamoDB
  • Write-always
• Cassandra [FB]
  – High reads, high writes

• Megastore [Google]
• Spanner [Google]
  – GFS
  – Google’-MapReduce
  – Chubby
  – ZooKeeper
General Data Center Characteristics

• Build on Commodity components
  – “in cluster first year, it’s typical that 1000 individual machines failures will occur”
  – “there’s about a 50 percent chacne that the cluster will overhead, taking down most of the servers in less than 5 minutes and taking 1 or 2 days to recover”

• BUGS in code

• Failures will happen a lot!!!!
  – But node will recover → Failures are transient.
  – Don’t want to react drastically because the node will come back up
ACID Properties (Cont’d)

- Experience at Amazon has shown that data stores that provide ACID guarantees tend to have poor availability.

- Dynamo targets applications that operate with weaker consistency (the “C” in ACID) if this results in high availability.
Dynamo: Always-write: Conflicts

- Add brush
- Add book
- Brush Book
- Add Water
- Add book
- Brush Book Water
- Delete Water

**D1 ([Sx,1])**
- write handled by Sx

**D2 ([Sx,2])**
- write handled by Sx

**D3 ([Sx,2],[Sy,1])**
- write handled by Sy

**D4 ([Sx,2],[Sz,1])**
- write handled by Sz

**D5 ([Sx,3],[Sy,1][Sz,1])**
- reconciled and written by Sx
## Summary of techniques used in *Dynamo* and their advantages

<table>
<thead>
<tr>
<th>Problem</th>
<th>Technique (Dynamo)</th>
<th>Cassandra</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partitioning</td>
<td>Consistent Hashing</td>
<td>Consistent hashing</td>
</tr>
<tr>
<td>Load Balancing</td>
<td>Virtual nodes</td>
<td>Reassigning Node IDs</td>
</tr>
<tr>
<td>High Availability for writes</td>
<td>Vector clocks with reconciliation during reads</td>
<td>Version size is decoupled from update rates.</td>
</tr>
<tr>
<td>Handling temporary failures</td>
<td>Sloppy Quorum and hinted handoff</td>
<td>wait</td>
</tr>
<tr>
<td>Recovering from permanent failures</td>
<td>Anti-entropy using Merkle trees</td>
<td>Manual intervention</td>
</tr>
<tr>
<td>Membership and failure detection</td>
<td>Gossip-based membership protocol and failure detection.</td>
<td>Gossip-based membership protocol and PHI failure detector</td>
</tr>
</tbody>
</table>
Why Zookeeper? Chubby?

• How to do primary election?
  – Ad hoc (no harm from duplicated work)
  – Operator intervention (correctness essential)

• Unprincipled
  • Disorganized, Costly, Low availability
  • Re-implement Raft-like protocol or use critical regions

• Design based on well-known ideas
  – distributed consensus, caching, notifications, file-system interface
Zookeeper V. Chubby

• Lessons from Chubby
  – Most requests are read/keep-alive
  – Few developers use locks
  – File system API is easy to use

• Zookeeper = Chubby with weaker read semantics and no locks
  – Weaker read semantics: client can read from any node
  – Writes must still go through leader
  – While no lock, you can implement locks
  – Enable Asynch requests: FIFO execution
Scaling Chubby

• Clients connect to a single instance of master in a cell
  – Much more client processes that number of machines
  – Note: Master and client have same CPU/Memory

• Existing mechanisms:
  – **Partition**: More Chubby cells (consistent hashing)
  – **Increase lease time**: from 12s to 60s to reduce KeepAlive messages
    (dominant requests in experiments)
  – **Client caches**: reduces reads/keepalive not writes
  – **Add new type of servers**: Add Proxy servers
Building Blocks
(L5-L10)
Building Blocks

• Communications
  – Networking layers,
  – TCP, UDP, BGP
  – RPCs

• Security
  – Cryptographic Frameworks
  – Attacks

• Not covered:
  – BGP, Network layers
Building Blocks Continued: Time Summary

• NTP
  – External Synchronized
    • Calculating time using external servers
    – More precision with specialized hardware (PTP)

• Inaccuracy/drifts

• Logical Clocks
  – Mutual exclusion with logical clocks
  – Total ordering V. Causal ordering

• Vector Clocks
Final Exam Details

• Exam Details
  – Cumulative exam
  – Two hours
  – Closed book

• Exam Venue
  – 2pm on Monday May 14th
  – Location: Metcalf AUD