CS138 Midterm Review
Communication
(Naming, RPC, TCP/UDP)

Load Balancing: DHT
(Sharding, Replication, Partitioning)

Ordering: Time
(Global Snapshots)

Consensus: Agreement, Consistency, Coordination
(Active/Passive/Lazy Replication, Transactions....)

(Google, Amazon, Facebook, MongoDB)

Midterm Covers
L1-L13
RPC
Communication between Nodes

• Abstraction: all nodes running as one process
  • Ideal: everything is a function call.

Who is NodeX
pack data into a packet
Send packet to X
Wait()
Get result from X
Unpack result from packet
Return result

........
Y = sendTo(NodeX, data)
........
Remote Procedure Call

Who is NodeX
pack data into a packet
Send packet to X
Wait()
Get result from X
Unpack result from packet
Return result

• Framework for automating this process
  • No need to re-write this code over (and over)
  • Hide complexity
  • Simplify coding
  • Provides similar abstraction (function calls)

• Developer defines:
  • Data types.
Key Challenges: Failures

• What can fail?

• How do you deal with these failures?

NodeA

Hey, here’s some data. Do something

NodeX

Working!

Done!
## Semantic Guarantees of RPCs

<table>
<thead>
<tr>
<th>Semantics</th>
<th>Request is lost</th>
<th>Response is lost</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Retransmit?</td>
<td>Filter duplicate?</td>
</tr>
<tr>
<td>At-least-once</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>(1 or more calls)</td>
<td></td>
<td>Re-execute</td>
</tr>
<tr>
<td></td>
<td></td>
<td>function</td>
</tr>
<tr>
<td>At-most-once</td>
<td>Yes</td>
<td>Yes,</td>
</tr>
<tr>
<td>(0 or 1 calls)</td>
<td></td>
<td>Use history to</td>
</tr>
<tr>
<td></td>
<td></td>
<td>filter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Use history to</td>
</tr>
<tr>
<td></td>
<td></td>
<td>retransmit</td>
</tr>
</tbody>
</table>
Replication and Partitioning
• Reasons to shard/partition
  • Data is too large to store in one location
  • Storing data in one location leads to high latency

• Distributed Hash table:
  • Sharding/partitioning a hash table

Partition data into shards, maps shards to server with consistent hashing

Clients send requests To all replicas

Hash table

| k0 | v0 |
| k1 | v1 |
| k2 | v2 |
| k3 | v3 |
| k4 | v4 |
| k5 | v5 |

All Facebook Data

Node A

Node D
• Distributed Hash Table
  • Distribute parts of the hash table across servers
  • Partition the hash table
• Use consistent hash to identify location of partitions
• Distributed Hash Table
  • Distribute parts of the hash table across servers
  • Partition the hash table

• Use consistent hash to identify location of partitions

• Replicate this shard across multiple servers
  • Use replication strategy to maintain consistency
Replication and Partitioning
Partition

• Cut a file into "chunks" (or sharding)

• Reduce impact of file growth
  • Limits amount of processing

• Definition (size) of chunk is app-specific
Replications

- Make many copies of a file
- Provides fault tolerance
  - Always one copy alive
- Provides more CPU/BW to the file
Consistent Hashing
Consistent Hashing

• Both Keys and Servers are hashed
  • Node A \rightarrow 7fc5...
  • “Ali” \rightarrow 32ff...

• Use ”Mod” to ensure ID space loops
  KeyID = Hash (Name) Mod N

• Insight: Both Key/ServerIDs go in same space
Failures: Adding or Removing Servers

Adding More Server Leads to more even distribution of the space
Still Need More Load Balancing? How do we Reduce Variance?

• Insight: need more to add more “nodes”
  • Not enough IDs to get statistical properties

• Virtualize nodes
  • Option 1: multiple Chash networks
    • Give keys/Nodes multiple IDs
  • Option 1: give a node multiple names.
    • Virtual copies: make virtual copies of servers
  • Option 2: give an object multiple keys.
    • Salting: make virtual keys
Replication: ReSalt V. Multiple Chash

Key1 = Hash \((\text{Salt0}+\text{Name})\) Mod N
Key2 = Hash \((\text{Salt1}+\text{Name})\) Mod N
Key3 = Hash \((\text{Salt2}+\text{Name})\) Mod N

Key1 = Hash1 \((\text{Name})\) Mod N
Key2 = Hash2 \((\text{Name})\) Mod N
Key3 = Hash3 \((\text{Name})\) Mod N
Tapestry
Tapestry

• How to build a Table?
  • What is stored in backpointers?

• How to Route to a key?
  • What is surrogate routing?

• What are ``need to know nodes’’?

• How to deal with failures?
What is routing table for 1332

<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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1XXX</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13XX</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>133X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What is routing table for 3122

<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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3XXX</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31XX</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>312X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
How to Route? Using Prefix Lookup

look up: 3122
Routing Algorithm

// executed at each node in route to destination
NextHop(targetHash, step) {
    nextDigit = digit(targetHash, step)
    return(table[step, nextDigit])
}

Route Table for Node: 2130

<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>0331</td>
<td>1332</td>
<td>2302</td>
<td>3312</td>
</tr>
<tr>
<td>2XXX</td>
<td>---</td>
<td>2130</td>
<td>---</td>
<td>2302</td>
</tr>
<tr>
<td>21XX</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>2130</td>
</tr>
<tr>
<td>213X</td>
<td>2130</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>
Routing Algorithm

// executed at each node in route to destination
NextHop(targetHash, step) {
    nextDigit = digit(targetHash, step)
    return(table[step, nextDigit])
}

Route Table for Node: 3312

<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>03</td>
<td>1332</td>
<td>2302</td>
<td>3312</td>
</tr>
<tr>
<td>3XXX</td>
<td>3311</td>
<td>3111</td>
<td>—</td>
<td>3312</td>
</tr>
<tr>
<td>33XX</td>
<td>—</td>
<td>3312</td>
<td>3320</td>
<td>—</td>
</tr>
<tr>
<td>331X</td>
<td>—</td>
<td>3311</td>
<td>3312</td>
<td>—</td>
</tr>
</tbody>
</table>
Routing Algorithm

// executed at each node in route to destination
NextHop(targetHash, step) {
    nextDigit = digit(targetHash, step)
    return(table[step, nextDigit])
}

look up: 3122
Table [3,2]
Next Digit = 2

3122.at(3)
Implications of Node Failure

- Problem: when a node crashes, all objects stored on node are lost
- Naïve solution: clients republish objects periodically
  - I.e., You (as a client) need to republish your facebook pictures.
  - Drawback 1: clients need to store and republish
  - Drawback 2: objects are unavailable until client republishes

- What are some alternative solutions?
  - “salt” the hash and publish several copies of the object
  - Recover from failure through redundancy
Distributed Hash Table Recap

• Consistent hash (Chash)
  • Benefit of consistent hashing over traditional key allocation
  • How to map keys to servers

• Chord (practical use of Chash)
  • Terms: Successor, routing table (finger table),
  • Building a routing table
  • Performing look-ups

• Tapestry
  • Terms: root node, surrogate node, backpointers, publishing
  • Building routing table
  • Performing lookup (regular routing, surrogate routing)
  • Adding/deleting a node (``need-to-know’’)
  • Optimizations (``salting’’, entry selection)
Time: Logistical Clocks and Vector Clocks
Time Overview

• Logical Clocks
  • Motivation for logical clocks/vector clocks
  • Calculating logical/vector clocks
  • Differences between logical and vector clocks

• Distributed Snapshots
  • Different approaches to capturing snapshots
  • Challenges/limitations of different approaches
  • Identifying consistent and inconsistent snapshots
Logical Clocks Versus Vector Clocks

• Vector clocks provide better ordering than logical clocks
  • Vector Clock = vector of logical clocks
  • HOWEVER!! Expensive – scales with # of processes – size of vector is # of processes
    • Network message: include vector clocks
    • Processes: must maintain vector clocks
Logical Clocks

• Each process maintains a ID – i.e., Clock
  • Initially set to 0

• Rules for updating Clock
  • Each event increments the ID
    • Event: Send msg, recv msg, or run function
  • Include ID in every message
  • On Receiving a message:
    • ID = max (my_ID, ID-in-msg) ++

• Only way to exchange information is through message exchange
  • IDs are only exchanged through message exchange

If \( x \rightarrow z \), then \( \text{clock}(z) > \text{clock}(x) \)
Vector Clocks

- Each process maintains an array of ID
  - Array is of size $N$ ($N = \# \text{ of processes}$)
  - All entries initialized to 0
  - Array is called a Vector clock (VC)

- Size ($VC_i$) == $N$
- $VC_i[i]$
  - Number of events at $P_i$
- $VC_i[h] = K$
  - Process $P_i$ is aware of the first $k$ events at $P_h$

If $VC_x \leq VC_z$ then $x \rightarrow z$ ($x$ happens before $z$)
Vector Clocks

- Each process maintains an array of ID
  - Array is of size $N$ ($N = \# \text{ of processes}$)
  - All entries initialized to 0

- Rules for updating Vector Clock for process $Z$
  - Each event increments $VC_z[z]$
    - So each event increments processes clock in vector
    - Event: Send msg, recv msg, or run function
  - Include ENTIRE vector in every message
  - On Receiving a message from process $X$ –
    - This message will have $VC_x[]$ –$X$'s vector clock
    - $Z$ updates its clock for each entry:
      - $VC_z[K] = \max (VC_z[K], VC_x[K])$
      - There's an exception: for index $Z$, you need to increment because of received event
        - $VC_z[Z] = \max (VC_z[Z], VC_x[Z]) + 1$

- Only way to exchange information is through message exchange
  - IDs are only exchanged through message exchange

If $VC_x \leq VC_z$ then $x \rightarrow z$ ($x$ happens before $z$)
Vector Clocks  ----- versus ----- Logical Clocks

Vector Clock Causality:
If X >= M, then M -> X
M causes X.
Vector Clocks  ----- versus ----- Logical Clocks

Vector Clock Causality:
If X >= M, then M -> X
M causes X.
Vector Clocks

Given the following logical clocks, what is a possible ordering of events? Note: this does not contain all events

- P0: A->[1,3], m->[2,3], x->[3,3], z->[4,8]
- P1: y->[2,4], b->[2,8]
Global Snapshot

- **Alternative 0:** Distributed Snapshots are easy with total ordering
  - However, total ordering is too expensive

- **Alternative 1:** Independent Snapshots
  - Each process independently creates snapshots
  - Select a snapshot across all processes that is globally consistent.
  - What are limitations of this approach?

- **Alternative 2:** ChandyLamport Consistent Snapshot Algorithm
  - Intuition: Coordinate snapshots by sending a snapshot message
  - What are limitations of this approach?
Identifying Inconsistent Snapshots With Vectors Clocks

A snapshot is inconsistent...
• If there exists two processes, Pi and Px,
  • Such that the VCs of their last events, Pi[x] > Px[x]
  • Restated, the last VCs at Pi and PX
    • demonstrates that Pi knows of an event at Px
    • But Px does not know of that event

Which of these are consistent:
• C1,2 + C2,2 + C3,2
• C1,1 + C2,2 + C3,0
• C1,0+C2,0 + C3,0
Is this a Consistent Snapshot?  
(Now with Vector Clocks)

Includes no events!

- P1: A
- P2: B, X
- P3: Z, M

- P1: A, G
- P2: B, X
- P3: Z

- P1: A, G
- P2: B, X, O, F
- P3:

\[ C_{1,0}, C_{2,0}, C_{3,0} \]
\[ C_{1,1}, C_{2,1}, C_{3,2} \]
\[ C_{1,2}, C_{2,1}, C_{3,1} \]
\[ C_{3,0}, C_{2,2}, C_{1,2} \]

P1: A
P2: B, X
P3: Z

P1: A, G
P2: B, X, O, F
P3:

\[ \text{checkpoint interval} \]

- \( C_{1,0} \)
  - 1,0,0
  - \( A \)
  - m1
- \( C_{1,1} \)
- \( C_{1,2} \)
  - 2,0,0
  - m3

- \( C_{2,0} \)
  - 1,2,0
  - \( B \)
  - m2
- \( C_{2,1} \)
- \( C_{2,2} \)
  - 1,3,0
  - \( O \)
  - m4

- \( C_{3,0} \)
  - 1,2,1
  - \( Z \)
- \( C_{3,1} \)
- \( C_{3,2} \)
  - 1,3,2
  - \( M \)
Identifying Inconsistent Snapshots With Vectors Clocks

A snapshot is inconsistent...

- If there exists two processes, $P_i$ and $P_x$,
  - Such that the VCs of their last events, $Pi[x] > Px[x]$

- Restated, the last VCs at $Pi$ and $Px$
  - demonstrates that $Pi$ knows of an event at $Px$
  - But $Px$ does not know of that event

- $C_{1,1}, C_{2,1}, C_{3,2}$ is not consistent
  - $P_3$’s last event -- $M = [1,3,2]$
  - $P_2$’s last event -- $X=[1,2,0]$
  - So $P_3$ knows of event 3 at $P_2$
  - But $P_2$ only knows of event 2
Formal Algorithm to Ensure Global Snapshot

- Any node can initiate a snapshot
  - By sending a marker to all nodes
  - Start watching all channels

- If \( N_X \) does not have a snapshot and receives a marker from node \( N_A \)
  - \( N_X \) creates a snapshot
  - \( N_X \) records channel \( C_A \) as empty
  - Start watching all channel except \( C_A \)
  - Send a marker on all channels (include \( C_A \))

- If \( N_X \) has a snapshot and receives marker from node \( N_A \)
  - \( N_X \) records channel \( C_A \) and adds to snapshot

System Assumptions
- No server (node) failure.
  - Server eventually processes message
- Network is reliable:
  - Messages are delivered eventually
  - Messages delivered in FIFO order
- Network is unreliable:
  - Messages are delivered eventually
  - Messages delivered in random order
Replication
Replication Overview

• Approaches to Replication: lazy, active, passive

• Passive: Raft \rightarrow Linearizability
  • Leader election, commit rules,
  • Ensuring Safety, liveness, log-safety

• Lazy Replication: Gossip \rightarrow Causal+
  • Impact of gossip on read latency
  • Rules for update/querying

• Consensus: Consistency Models
  • Definitions of different consistency models
  • Differences between the models

• CAP Theorem: Given ‘P’, you can only have “A” and “P”.
  • When designing a system that must tolerate partitions, you must pick between “A” and “P”.
Approaches to Replication

Active Replication
- FIFO ordering
- Tolerates byzantine failures

Passive Replication
- Total ordering (Linearizable)
- Protocols: Zookeeper, Paxos, Chubby

Lazy Replication
- Causal ordering (causal+)
- Protocols: Gossip, DynamoDB, CassandraDB, VoldemortDB, MongoDB
Assumptions!

- Each program is a state machine
  - Deterministic
  - Given initial state + sequence of events
    - Terminates at same state

- Replicated State Machine (RSM)

- Implications of RSM
  - Each server can independently process events
  - AND reach same conclusion
    - ONLY if events are total ordered
Raft Properties

- **Safety**: at most one leader
  - Each follower votes for at most one candidate
  - A candidate needs a majority to be leader

- **Liveness**: eventually there will be a leader
  - Challenge: if multiple try to call for election → split vote
  - Timeout + randomness: randomness helps to ensure that one server detects faster than the others

- **Log Safety**: if leader commits, then data is in all future leaders
  - Election Modifications: followers only vote for client with higher term/index
  - Commit Modifications: New Leader does not commit until entries in current term have been agreed on by followers
Overview of Lazy Replication

• Goal: give client data newer than time stamp
  • Not the most recent data just newer than FE timestamp

• Query: return value only if local timestamp is higher than client’s time stamp
  • Client may have to wait until replica gets a new value

• Update: only update data if local timestamp is higher than client's timestamp

• Replica Server May need to wait for gossip before responding to an FE
Lazy Replication: All Put Together

Diagram:
- FE (Front End) nodes send updates to other replica managers.
- Gossip messages are exchanged between replica managers.
- A Timestamp table keeps track of stable updates.
- Executed log stores K, V, VC pairs.
- Replica manager receives updates and logs them for stability.

Key Elements:
- VCx
- log
- K, V, VC
- ID
- K, VCreq

Diagram shows the flow of data and interactions in a lazy replication system.
Thinking About Consistency

• All replicas are one server

• If different clients write and read to this "one" server, what should we expect?

C1
  Get(c)  set(c=5)

C2
  Get(c)  set(c=7)

C3
  Get(c)  Get(c)

C4
  Get(c)  Get(c)
Consistency Spectrum

- **Strict Serializability**: total order + real time for transactions
- **Linearizable**: total order + real time (for individual operations)
- **Sequential**: total order + client order
- **Causal+**: causally ordered + eventually everyone agree
- **Eventual**: eventually everyone agrees

**SLOWER**
BUT EASY TO PROGRAM

**FAST and Parallel**
BUT HARDER TO PROGRAM: need conflict resolution

**STRONG CONSISTENCY**

**WEAK CONSISTENCY**
Comparing Different Models

Initial $c = 3$

- Get($c$) set($c=5$)
- Get($c$) set($c=7$)
- Get($c$) Get($c$)
- Get($c$) Get($c$)

- Is this Linearizable?

| C1   | 3  | 3  |
| C2   | 5  | 7  |
| C3   |    |    |
| C4   |    |    |
Comparing Different Models

Initial $c = 3$

- C1: Get(c) → set(c=5)
- C2: Get(c) → set(c=7)
- C3: Get(c) → Get(c)
- C4: Get(c) → Get(c)

• Is this Linearizable?
Comparing Different Models

Initial $c = 3$

- C1: $\text{Get}(c)$, $\text{set}(c=5)$
- C2: $\text{Get}(c)$, $\text{set}(c=7)$
- C3: $\text{Get}(c)$, $\text{Get}(c)$
- C4: $\text{Get}(c)$, $\text{Get}(c)$

$C1$: 5, 7
$C2$: 5
$C3$: 7
$C4$: 5

• Is this Serializable?
Comparing Different Models

Initial c = 3

• Is this Serializable?

C1: Get(c) → set(c=5)

C2: Get(c) → set(c=7)

C3: Get(c) → Get(c) → 7

C4: Get(c) → Get(c) → 5

C1: 5

C2: 7

C3: 7

C4: 5
Comparing Different Models

Initial c = 3

- Is this Serializable?

C1
- Get(c)
- set(c=5)

C2
- Get(c)
- set(c=7)

C3
- Get(c)
- Get(c)

C4
- Get(c)
- Get(c)

Initial c = 3

C1
- Get(c)
- set(c=5)

C2
- Get(c)
- set(c=7)

C3
- Get(c)
- Get(c)

C4
- Get(c)
- Get(c)
Comparing Different Models

Initial $c = 3$

- Get($c$)
- set($c=5$)

C1

- Get($c$)
- Get($c$)

C2

- Get($c$)
- set($c=7$)

C3

- Get($c$)

C4

- Get($c$)

- Get($c$)

- Get($c$)

• Is this Causal+?
Comparing Different Models

• Is this Causal+?

Initial $c = 3$

C1

Get(c) set(c=5)

C2

Get(c) set(c=7)

C3

Get(c) Get(c)

C4

Get(c) Get(c)

\[
\begin{align*}
C_1 & \rightarrow 5 \\
C_2 & \rightarrow 5 \\
C_3 & \rightarrow 5 \\
C_4 & \rightarrow 5
\end{align*}
\]
CAP Theorem

• Given a “Partition”, you must pick between “Availability” and “Consistency”
  • Pick Consistently: Some clients (not all) can change “data consistently”
  • Pick Availability: All clients can change data but “inconsistently”

• C: Consistency (Linearizable)
• A: Availability
• P: Partition tolerance
Distributed Transactions

• Background on Transactions
  • ACID Semantics

• Distribute Transactions
  • Terminology: Transaction manager, Coordinator, Participant
  • Two Phase Commit
    • Adding Isolation with Locks: optimistic V. pessimistic
    • Performance Issues
  • Consistency Models
    • Serializability Versus Linearizability