Distributed Distributed Systems

Lazy Replication and Transactions
Today!

• Lazy Replication

• CAP Theorem

• ACID Semantics

• Distributed Transactions
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 must wait for condition to hold

• Update: only update data if local timestamp is higher than client's times stamp
  • Server return OK but waits internally for condition to hold

• Server May need to wait for replication before responding to an FE
What is Gossip?

Diagram:
- Server A
- Server B
- Server C
- Server D
- Server E

Connections:
- Server A to Server B
- Server A to Server C
- Server A to Server E
- Server C to Server E
- Server D to Server E

Diagram:
- Server A
- Server B
- Server D
- Server C
- Server E

Connections:
- Server A to Server B
- Server A to Server D
- Server C to Server E
- Server D to Server E

Diagram:
- Server A
- Server B
- Server D
- Server C
- Server E

Connections:
- Server A to Server B
- Server A to Server D
- Server C to Server E
- Server D to Server E

What is Gossip?

• Each node selects a random subset
  • Replicates information to this subset

• Eventually data is replicated across all nodes
  • The same data may be sent to a node multiple times
  • Replication can take a while

Replication = (Frequency of Gossiping (freq. of data replication) | Policy for node selection (how to select nodes to gossip to) | Network Partitions)
Lazy Replication: Sending Gossip

- Gossip: sending update log to nodes
  - Randomly select gossip group
    - Gossip group == nodes to gossip with
  - For each node, $N_i$, in gossip group
    - Gossip subset of update-log with $VC_{req} \geq VC_i$
Lazy Replication: Receiving Gossip

- On receive gossip message
  - Merge with ``update-log''
  - Only keep messages newer than node’s timestamp

- Process ``update-log''
  - Order ``update-log'' according to timestamps
  - Use requests to update K,V store

![Diagram of update log and relevant nodes](image)
Lazy Replication: Receiving Gossip

• On receive gossip message
  • Merge with ``update-log’’
  • Only keep messages newer than node’s timestamp

• Process ``update-log’’
  • Order ``update-log’’ according to timestamps
  • Use requests to update K,V store
Lazy Replication: All Put Together

- Updates
- Gossip messages
- Timestamp table
- Update log
- Executed log
- FE
- Other replica managers
- VC
- K, V
- log
- ID
- K, V
- VC
- ID
- K, V
- VC
Lazy Replication: Open Challenges

• Log message can grow to an infinite size

• Messages exchanged can be quite large

• May be unable to order clocks (due to conflicts)
Lazy Replication: Sending Gossip

- **Gossip**: sending update log to nodes
  - Randomly select gossip group
    - Gossip group == nodes to gossip with
  - For each node, \( N_i \), in gossip group
    - Gossip subset of update-log with \( VC_{req} \geq VC_i \)

- **Update log**
  - \( VC_A, VC_B, VC_C \)
  - \( K, V, VC_K \) local K/V store

- **Executed log**
  - \( VC_A, VC_B, VC_C \)
  - \( K, V, VC_K \) local K/V store

- **Gossip group**
  - Nodes to gossip with

- **Server A**
  - \( VC_A, VC_B, VC_C \)
  - \( K, V, VC_K \) local K/V store
  - Update log
  - Executed log

- **Server B**
  - \( VC_A, VC_B, VC_C \)
  - \( K, V, VC_K \) local K/V store
  - Update log
  - Executed log

- **Server C**
  - \( VC_A, VC_B, VC_C \)
  - \( K, V, VC_K \) local K/V store
  - Update log
  - Executed log
Lazy Replication: Open Challenges

• Log message can grow to an infinite size
  • Use VC and Client-ACKs to limit size
  • Client-ACKs $\rightarrow$ limits executed log

• Messages exchanged can be quite large
  • Use VC to limit size
  • VC limits size of log in gossip messages

• May be unable to order clocks (due to conflicts)
  • Discuss in the future
Email Services

• Sending/receiving emails

• Add/remove users

• Should both use causal consistency? i.e., lazy replication?
• Services want a combination of Passive and Lazy replication
  • Some operations require passive replications (e.g., sending money)
  • Others work with lazy replication (e.g., posting)

• How do you merge passive and lazy replication?
  • Approach 1: two different systems
  • Approach 2: same system which can switch modes
CAP Theorem

• Consistency Model: How does your system react during partition
  
  • C: Consistency (linearizable)
    • All access is linearizable
    • Requires consensus
  
  • A: Availability
    • All clients can make progress
  
  • P: Partition tolerance
    • C and A can happen during a partition
**CAP Theorem**

- **Partition tolerance**: Return data during network failures
- **Consistency**: Return the most recent write else and error. Consistency == (linearizability)
- **Availability**: Return the data ASAP.

[https://medium.com/system-design-blog/cap-theorem-1455ce5fc0a0](https://medium.com/system-design-blog/cap-theorem-1455ce5fc0a0)
Approaches to Replication

Active Replication

Passive Replication

Lazy Replication

Server A

Server B (follower)

Server C (Follower)

Server A

Server B

Server C
Where Does Raft lie?
Where Does Raft lie?

Can’t commit:
- Not the leader
- Can’t reach leader

No availability
Where Does Lazy Replication lie?

Set(k, v, ts)

FE

Server A

k_A v_A ts_A
k_B v_B ts_B

Server B

k_A v_A ts_A
k_B v_B ts_B

Server C
Where Does Lazy Replication lie?

- **Query:** return value only if local timestamp is higher than client’s time stamp
  - Client must wait for condition to hold

- **Update:** only update data if local timestamp is higher than client's times stamp
  - Server return OK but waits internally for condition to hold
Raft(linearizable): Passive Replication:
- Strong consistency
- During partition ---
  - some clients will make no progress
  - Because leader is unavailable

Eventual Consistency:
- During partition ---
  - Some clients will make progress
  - Since clients can change same data
    - No consistency guarantees
• 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