Raft Replica Manager

State for the State machine

S1
S2
S4

A
B
A
B
A

Log Entries!!!!
When should you use Raft-style Replication?
When should you use Raft-style Replication?

- Server A
- New replica server

Sequence of events

Server Replication?

Log replication

Metadata Management
Ideal Usage scenario

- Small data with infrequent updates within the same data center
  - Small data → all data is stored in memory, so must fit in memory
  - Infrequent updates → updates require coordination and replication which is expensive
  - Same data center --> coordination-speed is a function on network latencies, server processing. Going over the internet means HUGE network latencies
Lazy Replication

VECTORS CLOCKS

HAVE A SPECIFIC SET OF SKILLS ... VERY SPECIFIC.
How do emails and forums work?

https://www.name.com/blog/fun-stuff/2012/05/office-email-chain-dog-policy/
Lazy Replication

- Supported operations
  - Get: returns value
  - Set: alters a value
    - Must get before setting

- Modifications:
  - K/V store includes ‘last modified time’

\[
\text{get}(k, t_s) \quad \text{set}(k, v, t_s) \quad \text{OK}(v, t_s) \quad \text{OK}(t_s)
\]

Time is last accessed time not current time
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 time stamp
  - Server return OK but waits internally for condition to hold

- **Server**: May need to wait for replication before responding to an FE
Lazy Replication: Query

Set(k, v, ts)

k_4  v_4  ts_4
k_5  v_5  ts_5

Server A

FE
Lazy Replication: Update

- FE creates ID
  - ID allows for duplicate suppression

- Submits (k, v, VC$_{req}$, ID) to server

- Server returns new time (i.e., VC$_{U}$)
  - For future requests, FE will use FC$_{U}$ as VC$_{FE}$

- Create Req-Info and store in `update-log`

Steps for creating new ts:
- Step 1: Find max of index i
- Step 2: Increment i by 1

Req_info === (K, V, ID, VC$_{req}$, VC$_{U}$)

Old timestamp

New timestamp

Request

Request ID
Lazy Replication: All Put Together

Set(k, v, ID, VC_{req})

Steps for creating new ts:
Step 1: Find max of index i
Step 2: Increment i by 1

FE

VC_U

Request

Req_info == (K, V, ID, VC_{req}, VC_U)

Old timestamp

New timestamp

FE

VC_{A}[#, #, #', #', #']

VC_{FE}[#, #, #, #, #]

VC_U[#, #, max(#, #') + 1, #, #]
Lazy Replication: All Put Together

Steps for creating a new ts:
Step 1: Find max of index $i$
Step 2: Increment $i$ by 1

Set($k,v, ID, VC_{req}$)

FE

VC_U

req_info
req_info
req_info
req_info

VC_A[#, #', #', #', #']

VC_FE[#, #, #, #, #]

VC_U[#, #, max(#, #') + 1, #, #]

K, V, VC_K
K, V, VC_K
K, V, VC_K

Local K/V store

Update log

Executed log

Req_info === (K, V, ID, VC_{req}, VC_U)

Request

Old timestamp

New timestamp

Request ID
Lazy Replication: Update

- Compare key’s time (VC_{K}) with req’s time (VC_{req})
  - Only update value if VC_{K} >= VC_{req}

- If value is updated store req-info in “executed log”

- Else keep req-info in “updated-log”
Lazy Replication: All Put Together

Set(k,v, ID, VC_{req})

\[ VC_K[\#,\#,\#] \geq VC_{req}[\#',\#',\#',\#'] \]
Lazy Replication: All Put Together

\[
\text{Set}(k,v, \text{ID}, VC_{\text{req}}) \rightarrow \text{FE} \rightarrow \text{VC}_U
\]

- Update log
  - \(..., VC_X, .. VC_N \)
  - \(K, V, VC_K \)
  - \(K, V, VC_K \)
  - \(K, V, VC_K \)
  - \(\text{Local K/V store} \)
  - \(\text{req info} \)
  - \(\text{req info} \)
  - \(\text{req info} \)
  - \(\text{req info} \)
  - \(\text{Executed log} \)

\[
VC_K [\#,\#,\#,\#] \geq VC_{\text{req}}[\#',\#',\#',\#']
\]
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 $V_{C_{\text{req}}} \geq V_{C_i}$
What is Gossip?

Pass

Lazy
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) \quad Policy for node selection (how to select nodes to gossip to) \quad 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
Lazy Replication: All Put Together

- **Gossip messages**:
  - From Other replica managers
  - To **Timestamp table**

- **Stable updates**:
  - From **Timestamp table**
  - To **Executing log**

- **Replica manager**:
  - Stores **log** and **Timestamp table**
  - Processes updates from **FEs**

- **FEs**:
  - send updates to **Replica manager**
  - Receive gossip messages from other replica managers
Lazy Replication: Challenges

• Log message can grow to an infinite size

• Messages exchanged can be quite large

• May be unable to order clocks
Approaches to Replication

Active Replication

Server A
Server B
Server C

Passive Replication

Server A (leader)
Server B (follower)
Server C (follower)

Lazy Replication

Server A
Server B
Server C
• 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

- **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
Where Does Raft lie?
Where Does Lazy Replication lie?

- **Query**: return value only if local timestamp is higher than client’s timestamp
  - 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
Distributed Systems

Day 11: Replication [Part 3 – Raft]

To survive failures you need a raft...