EECS 491
Introduction to Distributed Systems

Fall 2019

Harsha V. Madhyastha
Summary of Techniques

Consistent Replication

- Lamport clock
- Vector clock
- Primary-backup
- Paxos

Performance Enhancements

- Caching
- Leases
- Invalidation
- Leader election
- Version vector
What have we overlooked?

- Say each server can serve $R$ PUTs/sec
- Rate at which RSM in Project 2 can serve PUTs?
- What about in Project 3?

- As you add more servers
  - Every replica has to handle every operation
  - Idle servers unutilized

- Horizontal scaling better than vertical scaling
  - Adding more commodity servers >> beefing up servers
Scaling up

- Assumption so far: **All replicas have entire state**
  - Example: Every replica has value for every key

- What we need instead:
  - **Partition state**
  - Map partitions to servers
Partitioning state

- Example:
  - Store student’s transcripts across 13 servers

- Simple strategy for partitioning:
  - Last name begins with A or B → server 1
  - Last name begins with C or D → server 2
  - ...
  - Last name begins with Y or Z → server 13

- Problem: Skew in load across servers
Partitioning state

- Modulo hashing
  - Apply hash function to key
  - Compute modulo to # of servers (N)
  - Store (key, value) pair at $hash(key) \mod N$

- Student transcripts example:
  - Hash function = $hash(Year\ of\ birth) \mod 13$
  - Hash function = $hash(Date\ of\ birth) \mod 13$
Problem for modulo hashing: Changing number of servers

$$h(x) = x + 1 \pmod{4}$$

Add one machine: $$h(x) = x + 1 \pmod{5}$$

Keys remapped to new nodes \(\rightarrow\) Need to transfer values
Consistent Hashing

- Represent hash space as a circle

- Partition keys across servers
  - Assign every server a random ID
  - Hash server ID
  - Server responsible for keys between predecessor and itself

- How to map a key to a server?
  - Hash key and execute read/write at successor
Adding/Removing Nodes

- Minimizes migration of state upon change in set of servers
  - Server addition: New server splits successor’s shard
  - Server removal: Successor takes over shard
Virtual nodes

- Each server gets multiple (say $v$) random IDs
  - Each ID corresponds to a virtual node
- If $N$ servers with $v$ virtual nodes per server, each virtual node owns $1/(vN)^{th}$ of hash space
- Larger $v \rightarrow$ better load balancing
  - Vary $v$ across servers to account for heterogeneity
Virtual nodes

What happens upon server failure?

- v successors take over
- Each now stores $\frac{(v+1)}{v} \times \frac{1}{N}$ of hash space
Using Consistent Hashing

How does client map keys to servers?

Front-ends must agree on set of active servers
Consistent Hashing Impact

- **Widely used** in key-value stores
  - Memcached
  - Cassandra
  - ...

- **Limited scalability** if strong consistency desired
Announcements

- Project 3 due in two weeks
  - Focus of discussion section tomorrow
  - Design considerations:
    » Minimize latencies for serving PutAppends and Gets
    » Minimize number of messages
    » Robustness
  - You’ll reuse Paxos and PaxosRSM in project 4

- Thinking ahead to midterm exam
  - Happy to review material week before exam
  - Send me requests
Distributed Hash Table

- Scalable lookup of node responsible for any key
  - Scale to thousands (or even millions) of nodes
  - No one node knows all nodes in the system

- Example usage:
  - Trackerless BitTorrent
  - Key = File content hash
  - Value = IP addresses of nodes that have file content
Successor pointers

- If you don’t have value for key, forward to succ.

Downside of approach?

O(N) Lookup
Efficient lookups

- What’s required to enable $O(1)$ lookups?
  - Every node must know all other nodes

- Need to convert linear search to binary search

- Idea: Maintain $\log(N)$ pointers to other nodes
  - Called finger table
  - Pointer to node $\frac{1}{2}$-way across hash space
  - Pointer to node $\frac{1}{4}$-way across hash space
  - …
Finger tables

- i’th entry at node n points to successor of hash(n) + 2^i
  - # of entries = # of bits in hash value

- Binary lookup tree rooted at every node
  - Threaded through others’ finger tables
Finger tables

Node n

Succ of hash(n)
Succ of hash(n)+2
Succ of hash(n)+2^2
Succ of hash(n)+(max hash)/2

How to recursively use finger tables to locate node for key k?
Lookup with finger table

\textbf{Lookup}(\text{key k, node n})

look in local finger table for

\text{highest f s.t.} \text{hash}(f) < \text{hash}(k)

\begin{align*}
\text{if f exists} & \\
\text{call Lookup}(k, f) & \quad //\text{next hop}
\end{align*}

\begin{align*}
\text{else} & \\
\text{return n's successor} & \quad //\text{done}
\end{align*}
Lookups take $O(\log N)$ hops

October 17, 2019
EECS 491 – Lecture 13
Is \(\log(N)\) lookup fast or slow?

- For a million nodes, it’s 20 hops
- If each hop takes 50 ms, lookups take a second
- If each hop has 10% chance of failure, it’s a couple of timeouts
- So \(\log(N)\) is better than \(O(N)\) but not great
Handling churn in nodes

- Need to update finger tables upon addition or removal of nodes

- Hard to preserve consistency in the face of these changes
Next time ...

- Dynamo
  - Amazon’s eventually consistent storage system that uses consistent hashing