Distributed Systems

CS425/ECE428

April 12 2022

Instructor: Radhika Mittal

Acknowledgements for some of the materials: Indy Gupta
Logistics

• HW5 is due tomorrow (not on Thursday!)
• HW6 will be released on Thursday.
• Final exam is on May 12th, 8-11 am.
  • It will be fully online.
  • Same format as your midterm, but longer.
  • It will be comprehensive – everything covered in class.
Our agenda for the next 3-4 classes

• Brief overview of key-value stores

• Distributed Hash Tables
  • Peer-to-peer protocol for efficient insertion and retrieval of key-value pairs.

• Key-value stores in the cloud
  • How to run large-scale distributed computations over key-value stores?
    • Map-Reduce Programming Abstraction
  • How to design a large-scale distributed key-value store?
    • Case-study: Facebook’s Cassandra
Our focus today

• Brief overview of key-value stores

• Distributed Hash Tables
  • Peer-to-peer protocol for efficient insertion and retrieval of key-value pairs.

• Key-value stores in the cloud
  • How to run large-scale distributed computations over key-value stores?
    • Map-Reduce Programming Abstraction
  • How to design a large-scale distributed key-value store?
    • Case-study: Facebook’s Cassandra
The Key-value Abstraction

• (Business) Key ➔ Value
  • (twitter.com) tweet id ➔ information about tweet
  • (amazon.com) item number ➔ information about it
  • (kayak.com) Flight number ➔ information about flight, e.g., availability
  • (yourbank.com) Account number ➔ information about it
The Key-value Abstraction (2)

- It’s a dictionary data-structure.
  - Insert, lookup, and delete by key
  - E.g., hash table, binary tree

- But *distributed*. 
Isn’t that just a database?

• Yes, sort of.
• Relational Database Management Systems (RDBMSs) have been around for ages
  • e.g. MySQL is the most popular among them
• Data stored in structured tables based on a Schema
  • Each row (data item) in a table has a primary key that is unique within that table.
• Queried using SQL (Structured Query Language).
  • Supports joins.
Relational Database Example

Example SQL queries
1. SELECT zipcode
   FROM users
   WHERE name = "Bob"

2. SELECT url
   FROM blog
   WHERE id = 3

3. SELECT users.zipcode, blog.num_posts
   FROM users JOIN blog
   ON users.blog_url = blog.url

<table>
<thead>
<tr>
<th>user_id</th>
<th>name</th>
<th>zipcode</th>
<th>blog_url</th>
<th>blog_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Alice</td>
<td>12345</td>
<td>alice.net</td>
<td>1</td>
</tr>
<tr>
<td>422</td>
<td>Charlie</td>
<td>45783</td>
<td>charlie.com</td>
<td>3</td>
</tr>
<tr>
<td>555</td>
<td>Bob</td>
<td>99910</td>
<td>bob.blogspot.com</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>id</th>
<th>url</th>
<th>last_updated</th>
<th>num_posts</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>alice.net</td>
<td>5/2/14</td>
<td>332</td>
</tr>
<tr>
<td>2</td>
<td>bob.blogspot.com</td>
<td>4/2/13</td>
<td>10003</td>
</tr>
<tr>
<td>3</td>
<td>charlie.com</td>
<td>6/15/14</td>
<td>7</td>
</tr>
</tbody>
</table>
Mismatch with today’s workloads

- Data: Large and unstructured
- Lots of random reads and writes
- Sometimes write-heavy
- Foreign keys rarely needed
- Joins infrequent
Key-value/NoSQL Data Model

• NoSQL = “Not Only SQL”
• Necessary API operations: get(key) and put(key, value)

• Tables
  • Like RDBMS tables, but …
  • May be unstructured: May not have schemas
    • Some columns may be missing from some rows
  • Don’t always support joins or have foreign keys
  • Can have index tables, just like RDBMSs
Key-value/NoSQL Data Model

- Unstructured
- No schema imposed
- Columns missing from some Rows
- No foreign keys, joins may not be supported

<table>
<thead>
<tr>
<th>user_id</th>
<th>name</th>
<th>zipcode</th>
<th>blog_url</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Alice</td>
<td>12345</td>
<td>alice.net</td>
</tr>
<tr>
<td>422</td>
<td>Charlie</td>
<td>99910</td>
<td>charlie.com</td>
</tr>
<tr>
<td>555</td>
<td></td>
<td></td>
<td>bob.blogspot.com</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>id</th>
<th>url</th>
<th>last_updated</th>
<th>num_posts</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>alice.net</td>
<td>5/2/14</td>
<td>332</td>
</tr>
<tr>
<td>2</td>
<td>bob.blogspot.com</td>
<td>6/15/14</td>
<td>10003</td>
</tr>
<tr>
<td>3</td>
<td>charlie.com</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Key-value/NoSQL Data Model

- NoSQL = “Not Only SQL”
- Necessary API operations: get(key) and put(key, value)

- Tables
  - Like RDBMS tables, but …
  - May be unstructured: May not have schemas
    - Some columns may be missing from some rows
  - Don’t always support joins or have foreign keys
  - Can have index tables, just like RDBMSs
Our focus today

• Brief overview of key-value stores

• Distributed Hash Tables
  • Peer-to-peer protocol for efficient insertion and retrieval of key-value pairs.

• Key-value stores in the cloud
  • How to run large-scale distributed computations over key-value stores?
    • Map-Reduce Programming Abstraction
  • How to design a large-scale distributed key-value store?
    • Case-study: Facebook’s Cassandra
Distributed Hash Tables (DHTs)

• Multiple protocols were proposed in early 1990s.
  • Chord, CAN, Pastry, Tapestry
  • Initial usecase: Peer-to-peer file sharing
    • key = hash of the file, value = file
  • Cloud-based distributed key-value stores reuse many techniques from these DHTs.

• Key goals:
  • Balance load uniformly across all nodes (peers).
  • Fault-tolerance
  • Efficient inserts and lookups.
Distributed Hash Tables (DHTs)

- Multiple protocols were proposed in early 1990s.
  - **Chord**, CAN, Pastry, Tapestry
  - Initial usecase: Peer-to-peer file sharing
    - key = hash of the file, value = file
  - Cloud-based distributed key-value stores reuse many techniques from these DHTs.

- Key goals:
  - Balance load uniformly across all nodes (peers).
  - Fault-tolerance
  - Efficient inserts and lookups.
Chord

- Developed at MIT by I. Stoica, D. Karger, F. Kaashoek, H. Balakrishnan, R. Morris

- Key properties:
  - Load balance:
    - spreads keys evenly over nodes.
  - Decentralized:
    - no node is more important than others.
  - Scalable:
    - cost of key lookup is $O(\log N)$, $N = \text{no. of nodes}$.
  - High availability:
    - automatically adjusts to new nodes joining and nodes leaving.
  - Flexible naming:
    - no constraints on the structure of keys that it looks up.
Chord: Consistent Hashing

- Uses **Consistent Hashing** on node’s (peer’s) address
  - SHA-1(ip_address, port) → 160 bit string
  - Truncated to \( m \) bits (modulo \( 2^m \))
  - Called peer id (number between 0 and \( 2^m-1 \))
  - Not unique but id conflicts very unlikely
  - Can then map peers to one of \( 2^m \) logical points on a circle

![Circle for \( m = 3 \)]
Chord: Consistent Hashing

- Uses Consistent Hashing on node's (peer's) address
  - SHA-1 (ip_address, port) $\rightarrow$ 160 bit string
  - Truncated to $m$ bits (modulo $2^m$)
  - Called peer id (number between 0 and $2^m - 1$)
  - Not unique but id conflicts very unlikely
  - Can then map peers to one of $2^m$ logical points on a circle

Where will N16 be placed on this circle?
Chord: Consistent Hashing

- Uses *Consistent Hashing* on node’s (peer’s) address
  - SHA-1 (ip_address, port) → 160 bit string
  - Truncated to $m$ bits (modulo $2^m$)
  - Called peer id (number between 0 and $2^m - 1$)
  - Not unique but id conflicts very unlikely
  - Can then map peers to one of $2^m$ logical points on a circle

Where will N45 be placed on this circle?
Ring of Peers: Running Example

- Say $m=7$ (128 possible points on the circle – not shown)
- 6 nodes in the system.
Mapping Keys to Nodes

• Use the same consistent hash function
  • SHA-1(key) → 160 bit string (key identifier)
    • Henceforth, we refer to SHA-1(key) as key.
  • The key-value pair stored at the key’s successor node.
  • successor(key) = first peer with id greater than or equal to (key mod $2^m$)
    • Cross-over the ring when you reach the end.
      • $0 < 1 < 2 < 3 \ldots \ldots < 127 < 0$ (for $m=7$)

• Consistent Hashing => with K keys and N peers, each peer stores $O(K/N)$ keys. (i.e., $< c.K/N$, for some constant c)
Ring of Peers: Running Example

Where will the value with key 42 be stored?
Ring of Peers: Running Example

Where will the value with key 42 be stored?

Value with key K42 stored here
Ring of Peers: Running Example

Where will the value with key 115 be stored?

Value with key K115 stored here
Performing Lookups

Suppose N80 receives a request to lookup K42.

What is the value for K42?

Need to ask the successor of K42!
Performing Lookups

• Option 1: Each node is aware of (can route to) any other node in the system.
  • Need a very large routing table.
  • Poor scalability with 1000s of nodes.
  • Any node failure and join will require a necessary update at all nodes.

• Option 2: Each node is aware of only its ring successor.
  • O(N) lookup. Not very efficient.

• Chord chooses a sweet middle-ground.
Performing Lookups

• Chord chooses a sweet middle-ground.
  • Each node is aware of $m$ other nodes.
  • Maintains a finger table with $m$ entries.
  • The $i$th entry of node $n$’s finger table = $\text{successor}(n + 2^i)$
    • $i$ ranges from 0 to $m-1$
Finger Tables

Compute the finger table for N80

ith entry of node n’s finger table = successor(n + 2^i),
i ranges from 0 to m-1

\[
\begin{align*}
\text{N80} & \rightarrow 0 \\
\text{N45} & \rightarrow 1 \\
\text{N32} & \rightarrow 2 \\
\text{N16} & \rightarrow 3 \\
\text{N96} & \rightarrow 4 \\
\text{N112} & \rightarrow 5 \\
\text{N80} & \rightarrow 6
\end{align*}
\]
Finger Tables

Finger Table at N80

<table>
<thead>
<tr>
<th>i</th>
<th>ft[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>96</td>
</tr>
<tr>
<td>1</td>
<td>96</td>
</tr>
<tr>
<td>2</td>
<td>96</td>
</tr>
<tr>
<td>3</td>
<td>96</td>
</tr>
<tr>
<td>4</td>
<td>96</td>
</tr>
<tr>
<td>5</td>
<td>112</td>
</tr>
<tr>
<td>6</td>
<td>16</td>
</tr>
</tbody>
</table>

Say \( m = 7 \)
Performing Lookups

Suppose N80 receives a request to lookup K42.

What is the value for K42?

Need to locate successor of K42!
Which nodes is N80 aware of?

Finger Table at N80

<table>
<thead>
<tr>
<th>i</th>
<th>ft[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>96</td>
</tr>
<tr>
<td>1</td>
<td>96</td>
</tr>
<tr>
<td>2</td>
<td>96</td>
</tr>
<tr>
<td>3</td>
<td>96</td>
</tr>
<tr>
<td>4</td>
<td>96</td>
</tr>
<tr>
<td>5</td>
<td>112</td>
</tr>
<tr>
<td>6</td>
<td>16</td>
</tr>
</tbody>
</table>

Say $m=7$
Performing Lookups

Suppose N80 receives a request to lookup K42.

What is the value for K42?

Need to locate successor of K42!

Forward the query to the most promising node you know of.
Search for key $k$ at node $n$

At node $n$, if $k$ lies in range $(n, \text{next}(n)]$, where $\text{next}(n)$ is $n$'s ring successor then $\text{next}(n) = \text{successor}(key)$. Send query to $\text{next}(n)$

Else, send query for $k$ to largest finger entry $\leq k$

What is the value for $K_{42}$?
Analysis

Search takes $O(\log(N))$ time

Proof Intuition:

• (intuition): at each step, distance between query and peer-with-file reduces by a factor of at least 2 (why?)

• (intuition): after $\log(N)$ forwardings, distance to key is at most $2^m/2^{\log(N)} = 2^m/N$

• Expected number of node identifiers in a range of $2^m/N$:
  • ideally one
  • $O(\log(N))$ with high probability (by properties of consistent hashing)

So using ring successors in that range will use another $O(\log(N))$ hops. Overall lookup time stays $O(\log(N))$. 
Analysis

- $O(\log(N))$ search time holds for file insertions too (in general for routing to any key)
  - “Routing” can thus be used as a building block for
    - all operations: insert, lookup, delete
- $O(\log(N))$ time true only if finger and successor entries correct
- When might these entries be wrong?
  - When you have failures
    - Coming up next!
Search for key k at node n

What is the value for K42?

K42 stored here
If a node fails

What is the value for $K42$?

Lookup fails (N16 does not know N45)

$m=7$

$K42$ stored here
If a node fails

What is the value for $K42$?

Lookup fails (N16 does not know N45)

How do we handle this?
If a node fails

One solution: maintain $r$ multiple ring successor entries. In case of failure, use another successor entries.

What is the value for $K42$?

$K42$ stored here.
Search under node failures

- If every node fails with probability 0.5, choosing $r=2\log(N)$ suffices to maintain lookup correctness (i.e. keep the ring connected) with high probability.
  - Intuition:
    - $\Pr(\text{at given node, at least one successor alive}) = 1 - \left(\frac{1}{2}\right)^{2\log N} = 1 - \frac{1}{N^2}$
    - $\Pr(\text{above is true at all alive nodes}) = \left(1 - \frac{1}{N^2}\right)^{N/2} = e^{-\frac{1}{2N}} \approx 1$
If a node fails

What is the value for K42?

Lookup fails (N45 itself is dead)

K42 stored here
If a node fails

One solution: replicate key-value at $r$ successors and predecessors

What is the value for $K42$?

$K42$ replicated

$m = 7$
Need to deal with dynamic changes

✓ Nodes fail
• New nodes join
• Nodes leave

To be continued in next class!