Principles of Software Construction: Objects, Design and Concurrency

Distributed System Design, part 2

Jonathan Aldrich  Charlie Garrod
• **Homework 8 due tonight**
  - Don't forget to turn in work from Lab 8

• **Homework 9 coming soon, due next Thursday**
  - You may work with a partner

• **Lab 9 on Monday, also due Thursday**
  - You must do this part alone
  - You and your partner may re-use code you develop for the lab
Last time: Distributed systems

• General principles
  ▪ Failure modes
  ▪ Dealing with complexity

• Replication for reliability
  ▪ Passive primary-backup replication
Last time: Distributed systems

- General principles
  - Failure modes
  - Dealing with complexity

- Replication for reliability
  - Passive primary-backup replication

```
client

front-end

{alice:90, bob:42, ...}

front-end

primary:

{alice:90, bob:42, ...}

backup:

{alice:90, bob:42, ...}

backup:
```
Last time: A case of contradictions: RAID

- **RAID**: Redundant Array of Inexpensive Disks
  - Within a single computer, replicate data onto multiple disks
  - e.g., with 5 1TB disks can get 4TB of useful storage and recover from any single disk failure

- Does Google use RAID?
Today: More distributed systems designs

- Replication for scalability
  - Caching

- Partitioning for scalability
  - Horizontal partitioning
  - Distributed hash tables
  - Consistent hashing
  - Master/tablet-based systems
Recall: Facebook and replication

- Replication for scalability:
  - Read-any, write-all
  - Palo Alto, CA is primary replica
Replication for scalability: Client-side caching

- **Architecture before replication:**

  - **Problem:** Server throughput is too low

  - **Solution:** Cache responses at (or near) the client
    - Cache can respond to repeated read requests

  ```
  database server:
  {alice:90, bob:42, ...}
  ```

  ```
  {alice:90, bob:42, ...}
  ```
Replication for scalability: Client-side caching

- Hierarchical client-side caches:

- The image shows a network of clients and caches, with data flowing from the clients to the cache and then to the database server. The cache structure is hierarchical, with caches at different levels to improve scalability.

- The database server contains data for multiple users, such as Alice and Bob, with additional users indicated by "...".
Replication for scalability: Server-side caching

- **Architecture before replication:**
  - **Problem:** Database server throughput is too low
  - **Solution:** Cache responses on multiple servers
    - Cache can respond to repeated read requests

![Diagram showing client communication with front-end, database server, and caches](image)
Cache invalidation

• Time-based invalidation (a.k.a. expiration)
  ▪ Read-any, write-one
  ▪ Old cache entries automatically discarded
  ▪ No expiration date needed for read-only data

• Update-based invalidation
  ▪ Read-any, write-all
  ▪ DB server broadcasts invalidation message to all caches when the DB is updated

• What are the advantages and disadvantages of each approach?
Cache replacement policies

• Problem: caches have finite size

• Common* replacement policies
  ▪ Optimal (Belady's) policy
    • Discard item not needed for longest time in future
  ▪ Least Recently Used (LRU)
    • Track time of previous access, discard item accessed least recently
  ▪ Least Frequently Used (LFU)
    • Count # times item is accessed, discard item accessed least frequently
  ▪ Random
    • Discard a random item from the cache
Partitioning for scalability

- Partition data based on some property, put each partition on a different server.

Example:

**CMU server:**
{cohen:9, bob:42, ...}

**MIT server:**
{deb:16, reif:40, ...}

**Yale server:**
{alice:90, pete:12, ...}

**Client** → **Front-end**
Horizontal partitioning

• a.k.a. "sharding"

• A table of data:

<table>
<thead>
<tr>
<th>username</th>
<th>school</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>cohen</td>
<td>CMU</td>
<td>9</td>
</tr>
<tr>
<td>bob</td>
<td>CMU</td>
<td>42</td>
</tr>
<tr>
<td>alice</td>
<td>Yale</td>
<td>90</td>
</tr>
<tr>
<td>pete</td>
<td>Yale</td>
<td>12</td>
</tr>
<tr>
<td>deb</td>
<td>MIT</td>
<td>16</td>
</tr>
<tr>
<td>reif</td>
<td>MIT</td>
<td>40</td>
</tr>
</tbody>
</table>
Recall: Basic hash tables

- For $n$-size hash table, put each item $x$ in the bucket: $x$.hashCode() % $n$
Partitioning with a distributed hash table

- Each server stores data for one bucket
- To store or retrieve an item, front-end server hashes the key, contacts the server storing that bucket
Consistent hashing

• **Goal:** Benefit from incremental changes
  - Resizing the hash table (i.e., adding or removing a server) should not require moving many objects

• **E.g., Interpret the range of hash codes as a ring**
  - Each bucket stores data for a range of the ring
    • Assign each bucket an ID in the range of hash codes
    • To store item \( x \) don't compute \( x.\text{hashCode}() \mod n \). Instead, place \( x \) in bucket with the same ID as or next higher ID than \( x.\text{hashCode}() \)
Problems with hash-based partitioning

- Front-ends need to determine server for each bucket
  - Each front-end stores look-up table?
  - Master server storing look-up table?
  - Routing-based approaches?

- Places related content on different servers
  - Consider range queries:
    ```sql
    SELECT * FROM users WHERE lastname STARTSWITH 'G'
    ```
Master/tablet-based systems

- Dynamically allocate range-based partitions
  - Master server maintains tablet-to-server assignments
  - Tablet servers store actual data
  - Front-ends cache tablet-to-server assignments

```
client
  └── front-end
    └── Master:
        {a-c:2,
         d-g:3,
         h-j:3,
         k-z:1}

client
  └── front-end
    └── Tablet server 1:
        k-z: {pete:12, reif:42}

client
  └── front-end
    └── Tablet server 2:
        a-c: {alice:90, bob:42, cohen:9}

client
  └── front-end
    └── Tablet server 3:
        d-g: {deb:16}
        h-j: {}
Combining approaches

• Many of these approaches are *orthogonal*

• E.g., For master/tablet systems:
  - Masters are often partitioned and replicated
  - Tablets are replicated
  - Meta-data frequently cached
  - Whole master/tablet system can be replicated
Next time

• Data consistency