Principles of Software Construction: Objects, Design, and Concurrency

Distributed System Design, Part 4

Charlie Garrod    Christian Kästner
Administrivia

- Homework 6, homework 6, homework 6...

- Upcoming:
  - This week: Distributed systems and data consistency
  - Next week: TBD and guest lecture
  - Final exam: Monday, May 12th, 5:30 – 8:30 p.m. UC McConomy
  - Final exam review session: Saturday, May 10th, 6 – 8 p.m. PH 100
Last time…
Today: Distributed system design, part 4

- General distributed systems design
  - Failure models, assumptions
  - General principles
  - Replication and partitioning
  - Consistent hashing
Types of failure behaviors

- Fail-stop
- Other halting failures
- Communication failures
  - Send/receive omissions
  - Network partitions
  - Message corruption
- Performance failures
  - High packet loss rate
  - Low throughput
  - High latency
- Data corruption
- Byzantine failures
Common assumptions about failures

• Behavior of others is fail-stop (ugh)
• Network is reliable (ugh)
• Network is semi-reliable but asynchronous
• Network is lossy but messages are not corrupt
• Network failures are transitive
• Failures are independent
• Local data is not corrupt
• Failures are reliably detectable
• Failures are unreliably detectable
Some distributed system design goals

- **The end-to-end principle**
  - When possible, implement functionality at the end nodes (rather than the middle nodes) of a distributed system

- **The robustness principle**
  - Be strict in what you send, but be liberal in what you accept from others
    - Protocols
    - Failure behaviors

- **Benefit from incremental changes**

- **Be redundant**
  - Data replication
  - Checks for correctness
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

- **Diagram:**
  - Client → Front-end → Database Server
  - Cache

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

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

- Hierarchical client-side caches:
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, front-end, cache, and database server connections]
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

Client 1  →  Front-end  →  CMU server:
             {cohen:9, bob:42, ...

Client 2  →  Front-end  →  Yale server:
             {alice:90, pete:12, ...

Client 3  →  Front-end  →  MIT server:
             {deb:16, reif:40, ...}
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

```
Client
  └─── front-end

Server 0:
  {reif:40}

Server 1:
  {}  

Server 3:
  {bob:42}

Server 5:
  {pete:12, alice:90}
...
```
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 $x$.hashCode() $\%$ n. Instead, place $x$ in bucket with the same ID as or next higher ID than $x$.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:
    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

client
    └── front-end

Master:

{a-c:[2],
d-g:[3,4],
h-j:[3],
k-z:[1]}

Tablet server 1:

k-z: {pete:12, reif:42}

Tablet server 2:

{a-c:{alice:90, bob:42, cohen:9}}

Tablet server 3:

{d-g:{deb:16}, h-j: {}}

Tablet server 4:

{d-g:{deb:16}}
```
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
Thursday

• Serializability