Principles of Software Construction: Objects, Design, and Concurrency

Distributed System Design, Part 3

Charlie Garrod  Christian Kästner
Administrivia

• Homework 6...
• 15-313
Last time…
MapReduce with key/value pairs (Google style)

- **Master**
  - Assign tasks to workers
  - Ping workers to test for failures

- **Map workers**
  - Map for each key/value pair
  - Emit intermediate key/value pairs

- **Reduce workers**
  - Sort data by intermediate key and aggregate by key
  - Reduce for each key

the shuffle:
MapReduce to count all words in a corpus

- E.g., for each word on the Web, count the number of times that word occurs
  - For Map: key1 is a document name, value is the contents of that document
  - For Reduce: key2 is a word, values is a list of the number of counts of that word

```java
f1(String key1, String value):
    for each word w in value:
        EmitIntermediate(w, 1);

f2(String key2, Iterator values):
    int result = 0;
    for each v in values:
        result += v;
    Emit(key2, result);
```

<table>
<thead>
<tr>
<th>Map: (key1, v1) \rightarrow (key2, v2)*</th>
<th>Reduce: (key2, v2*) \rightarrow (key3, v3)*</th>
</tr>
</thead>
<tbody>
<tr>
<td>MapReduce: (key1, v1)* \rightarrow (key3, v3)*</td>
<td>MapReduce: (docName, docText)* \rightarrow (word, wordCount)*</td>
</tr>
</tbody>
</table>
MapReduce to count mutual friends

- E.g., for pair in a social network graph, output the number of mutual friends they have
  - For Map: key1 is a person, value is the list of her friends
  - For Reduce: key2 is a pair of people, values is a list of 1s, for each mutual friend that pair has

\[ \text{f1(String key1, String value):} \]
\[
\text{for each pair of friends in value:} \\
\text{EmitIntermediate(pair, 1);} \\
\]

\[ \text{f2(String key2, Iterator values):} \]
\[
\text{int result = 0;} \\
\text{for each v in values:} \\
\text{result += v;} \\
\text{Emit(key2, result);} \\
\]

MapReduce: (person, friends)* → (pair of people, count of mutual friends)*
Today: Distributed system design, part 3

- MapReduce, continued

- General distributed systems design
  - Failure models, assumptions
  - General principles
  - Replication and partitioning
  - Consistent hashing
MapReduce to count incoming links

- E.g., for each page on the Web, count the number of pages that link to it
  - For Map: `key1` is a document name, `value` is the contents of that document
  - For Reduce: `key2` is `??`, `values` is a list of `??`

\[
\text{MapReduce: (docName, docText)* } \rightarrow \text{ (docName, number of incoming links)*}
\]
MapReduce to count incoming links

- E.g., for each page on the Web, count the number of pages that link to it
  - For Map: key1 is a document name, value is the contents of that document
  - For Reduce: key2 is ???, values is a list of ???

```java
f1(String key1, String value):
   for each link in value:
      EmitIntermediate(link, 1)

f2(String key2, Iterator values):
   int result = 0;
   for each v in values:
      result += v;
   Emit(key2, result);
```

MapReduce: (docName, docText)* → (docName, number of incoming links)*
MapReduce to create an inverted index

- E.g., for each page on the Web, create a list of the pages that link to it
  - For Map: key1 is a document name, value is the contents of that document
  - For Reduce: key2 is ???, values is a list of ???

```java
f1(String key1, String value):
  for each link in value:
    EmitIntermediate(link, key1)

f2(String key2, Iterator values):
  Emit(key2, values)
```

MapReduce: (docName, docText)* → (docName, list of incoming links)*
List the mutual friends

- E.g., for each pair in a social network graph, list the mutual friends they have
  - For Map: key1 is a person, value is the list of her friends
  - For Reduce: key2 is ???, values is a list of ???

f1(String key1, String value):

f2(String key2, Iterator values):

MapReduce: (person, friends)* → (pair of people, list of mutual friends)*
List the mutual friends

• E.g., for each pair in a social network graph, list the mutual friends they have
  ▪ For Map: key1 is a person, value is the list of her friends
  ▪ For Reduce: key2 is ???, values is a list of ???

\[
\text{f1}(\text{String key1, String value}): \\
\text{for each pair of friends} \\
\text{in value:} \\
\text{EmitIntermediate(pair, key1)};
\]

\[
\text{f2}(\text{String key2, Iterator values}): \\
\text{Emit(key2, values)}
\]

MapReduce: (person, friends)* → (pair of people, list of mutual friends)*
Count friends + friends of friends

- E.g., for each person in a social network graph, count their friends and friends of friends
  - For Map: key1 is a person, value is the list of her friends
  - For Reduce: key2 is ???, values is a list of ???

MapReduce: (person, friends)* → (person, count of f + fof)*
Count friends + friends of friends

- E.g., for each person in a social network graph, count their friends and friends of friends
  - For Map: key1 is a person, value is the list of her friends
  - For Reduce: key2 is ???, values is a list of ???

\[
f1(\text{String key1, String value}): 
\begin{align*}
&\text{for each friend1 in value:} \\
&EmitIntermediate(friend1, key1) \\
&\text{for each friend2 in value:} \\
&EmitIntermediate(friend1, friend2); \\
\end{align*}
\]

\[
f2(\text{String key2, Iterator values}): 
\begin{align*}
&\text{distinct_values} = \{\} \\
&\text{for each v in values:} \\
&\quad \text{if not v in distinct_values:} \\
&\quad \quad \text{distinct_values.insert(v)} \\
&Emit(key2, \text{len(distinct_values)})
\end{align*}
\]

MapReduce: \((\text{person, friends})^* \to (\text{person, count of f + fof})^*\)
Friends + friends of friends + friends of friends of friends

- E.g., for each person in a social network graph, count their friends and friends of friends and friends of friends of friends
  - For Map: key1 is a person, value is the list of her friends
  - For Reduce: key2 is ???, values is a list of ???

\[
\text{MapReduce: (person, friends)*} \rightarrow (\text{person, count of f + fof + fofof})*
\]
Problem: How to reach distance 3 nodes?

- **Solution:** Iterative MapReduce
  - Use MapReduce to get distance 1 and distance 2 nodes
  - Feed results as input to a second MapReduce process

- **Also consider:**
  - Breadth-first search
  - PageRank
  - ...
Dataflow processing

- High-level languages and systems for complex MapReduce-like processing
  - Yahoo Pig, Hive
  - Microsoft Dryad, Naiad
- MapReduce generalizations...
Today: Distributed system design, part 3

- MapReduce, continued
- General distributed systems design
  - Failure models, assumptions
  - General principles
  - Replication and partitioning
  - Consistent hashing
Recall passive primary-backup replication

- **Architecture before replication:**
  
  ```
  client  front-end  database server:
  {alice:90, bob:42, ...}
  client  front-end  database server:
  {alice:90, bob:42, ...}
  client  front-end  database server:
  {alice:90, bob:42, ...}
  ```

- **Problem:** Database server might fail

- **Solution:** Replicate data onto multiple servers
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

```
client ———— front-end ———— database server:
{alice:90, bob:42, ...}

client ———— front-end ———— database server:
{alice:90, bob:42, ...}
```

```
client ———— cache ———— front-end ———— database server:
{alice:90, bob:42, ...}

client ———— cache ———— front-end ———— database server:
{alice:90, bob:42, ...}
```
Replication for scalability: Client-side caching

- Hierarchical client-side caches:

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

```
client  front-end  database server:
        {alice:90, bob:42, ...}
client  front-end  database server:
        {alice:90, bob:42, ...}
```

client  front-end  cache  database server:
        {alice:90, bob:42, ...}
client  front-end  cache  database server:
        {alice:90, bob:42, ...}
client  front-end  cache  database server:
        {alice:90, bob:42, ...}
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

Front-end

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

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

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: \( x\text{.hashCode}() \mod n \)
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 \rightarrow \text{front-end}\rightarrow Server 0:  \{\text{reif:40}\}

Client \rightarrow \text{front-end}\rightarrow Server 3:  \{\text{bob:42}\}

Server 1:  \{\}

Server 5:  \{\text{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'
Next week

• More distributed systems...
• Serializability