Principles of Software Construction: Objects, Design, and Concurrency

Distributed System Design, Part 2

Charlie Garrod    Jonathan Aldrich

© 2012-14  C Kästner, C Garrod, J Aldrich, and W Scherlis
Administrivia

• Homework 5b due tonight
  ▪ Finish by tomorrow (14 Nov) 10 a.m. if you want to be considered as a "Best Framework" for Homework 5c

• 15-413: Software Engineering Practicum

• Homework 3 arena winners in class next week...
Key concepts from Tuesday
Networking in Java

• The java.net.InetAddress:
  static InetAddress getByByName(String host);
  static InetAddress getByAddress(byte[] b);
  static InetAddress getLocalHost();

• The java.net.Socket:
  Socket(InetAddress addr, int port);
  boolean isConnected();
  boolean isClosed();
  void close();
  InputStream getInputStream();
  OutputStream getOutputStream();

• The java.net.ServerSocket:
  ServerSocket(int port);
  Socket accept();
  void close();
  ...
Aside: The robustness vs. redundancy curve
Metrics of success

• Reliability
  ▪ Often in terms of availability: fraction of time system is working
    ▪ 99.999% available is "5 nines of availability"

• Scalability
  ▪ Ability to handle workload growth
Today: Distributed system design

- Introduction to distributed systems, continued
  - Motivation: reliability and scalability
  - Failure models
  - Techniques for:
    - Reliability (availability)
    - Scalability
    - Consistency

- MapReduce: A robust, scalable framework for distributed computation...
  - ...on replicated, partitioned data
Types of failure behaviors

- Fail-stop
- Other halting failures
- Communication failures
  - Send/receive omissions
  - Network partitions
  - Message corruption
- Data corruption
- Performance failures
  - High packet loss rate
  - Low throughput
  - High latency
- 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 ends (rather than the middle) 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

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

```
client → front-end

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

- Hierarchical client-side caches:

```
client -> cache -> cache
client -> cache
client
```

Database:
```
{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 ── cache
  ── front-end ── cache
  ── front-end ── cache
```

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
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

*Common policies may vary in different contexts.
Partitioning for scalability

- Partition data based on some property, put each partition on a different server.
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 $x.hashCode() \% n$. 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}
```
Today: Distributed system design

- Introduction to distributed systems, continued
  - Motivation: reliability and scalability
  - Failure models
  - Techniques for:
    - Reliability (availability)
    - Scalability
    - Consistency

- MapReduce: A robust, scalable framework for distributed computation...
  - ...on replicated, partitioned data
Map from a functional perspective

• **map(f, x[0…n−1])**
  
  • Apply the function f to each element of list x

![Diagram of map operation]

• E.g., in Python:
  
  ```python
def square(x): return x*x
map(square, [1, 2, 3, 4]) would return [1, 4, 9, 16]
```

• Parallel map implementation is trivial
  
  • What is the work? What is the depth?
Reduce from a functional perspective

- `reduce(f, x[0...n-1])`
  - Repeatedly apply binary function `f` to pairs of items in `x`, replacing the pair of items with the result until only one item remains
  - One sequential Python implementation:
    ```python
def reduce(f, x):
    if len(x) == 1: return x[0]
    return reduce(f, [f(x[0],x[1])] + x[2:])
```
  - e.g., in Python:
    ```python
def add(x,y): return x+y
reduce(add, [1,2,3,4])  # would return 10 as
reduce(add, [3,3,4])
reduce(add, [6,4])
reduce(add, [10]) -> 10
```
Reduce with an associative binary function

• If the function $\oplus$ is associative, the order $\oplus$ is applied does not affect the result

1 + ((2+3) + 4)  \quad 1 + (2 + (3+4))  \quad (1+2) + (3+4)

• Parallel reduce implementation is also easy
  - What is the work? What is the depth?
Distributed MapReduce

• The distributed MapReduce idea is similar to (but not the same as!):
  \[ \text{reduce}(f_2, \text{map}(f_1, x)) \]

• Key idea: a "data-centric" architecture
  - Send function \( f_1 \) directly to the data
    - Execute it concurrently
  - Then merge results with reduce
    - Also concurrently

• Programmer can focus on the data processing rather than the challenges of distributed systems
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 with key/value pairs (Google style)

- 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

\[ f1(String\ key1, String\ value): \]
\[
    for\ each\ word\ w\ in\ value:\n        EmitIntermediate(w, 1);
\]

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

Map: (key1, v1) → (key2, v2)*
Reduce: (key2, v2*) → (key3, v3)*
MapReduce: (key1, v1)* → (key3, v3)*

MapReduce: (docName, docText)* → (word, wordCount)*
MapReduce architectural details

- Usually integrated with a distributed storage system
  - Map worker executes function on its share of the data

- Map output usually written to worker's local disk
  - Shuffle: reduce worker often pulls intermediate data from map worker's local disk

- Reduce output usually written back to distributed storage system
Handling server failures with MapReduce

• Map worker failure:
  ▪ Re-map using replica of the storage system data

• Reduce worker failure:
  ▪ New reduce worker can pull intermediate data from map worker's local disk, re-reduce

• Master failure:
  ▪ Options:
    • Restart system using new master
    • Replicate master
    • ...
The beauty of MapReduce

- **Low communication costs (usually)**
  - The shuffle (between map and reduce) is expensive

- **MapReduce can be iterated**
  - Input to MapReduce: key/value pairs in the distributed storage system
  - Output from MapReduce: key/value pairs in the distributed storage system
Another MapReduce example

- E.g., for person 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 ???, values is a list of ???

\[ f1(String\ key1,\ String\ value): \quad f2(String\ key2,\ Iterator\ values): \]

MapReduce: (person, friends)* \rightarrow (pair\ of\ people,\ count\ of\ mutual\ friends)*
Another MapReduce example

- E.g., for person 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)*
And another MapReduce example

- 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):

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

MapReduce: (docName, docText)* → (docName, list of incoming links)*
Coming next…

- More distributed systems
  - MapReduce