Principles of Software Construction: Objects, Design, and Concurrency

Distributed System Design, Part 2. MapReduce

Charlie Garrod  Christian Kästner
Administrivia

- Homework 5c due tonight
- Homework 6 coming tomorrow
Road map from last time…

• Application-level communication protocols
• Frameworks for simple distributed computation
  ▪ Remote Procedure Call (RPC)
  ▪ Java Remote Method Invocation (RMI)
• Common patterns of distributed system design
• Complex computational frameworks
  ▪ e.g., distributed map-reduce
Today: Distributed system design, part 2

- Introduction to distributed systems
  - Motivation: reliability and scalability
  - Replication for reliability
  - Partitioning for scalability

- MapReduce: A robust, scalable framework for distributed computation...
  - ...on replicated, partitioned data
You need to restart your computer. Hold down the Power button for several seconds or press the Restart button.

Veuillez redémarrer votre ordinateur. Maintenez la touche de démarrage enfoncee pendant plusieurs secondes ou bien appuyez sur le bouton de réinitialisation.

Sie müssen Ihren Computer neu starten. Halten Sie dazu die Einschalttaste einige Sekunden gedrückt oder drücken Sie die Neustart-Taste.

コンピュータを再起動する必要があります。パワーボタンを数秒間押し続けるか、リセットボタンを押してください。
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
A case study: Passive primary-backup replication

• Architecture before replication:

- Problem: Database server might fail
A case study: Passive primary-backup replication

- **Architecture before replication:**
  - Problem: Database server might fail
  - Solution: Replicate data onto multiple servers

[Diagram showing client, front-end, and database server connections, with data replication from primary to backup servers.]
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, ...}

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

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

Front-end → 
  - Tablet server 1:
  - Tablet server 2:
  - Tablet server 3:
  - Tablet server 4:
  
Master:

```
{k-z:{pete:12, reif:42}}
{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:[2], d-g:[3,4], h-j:[3], k-z:[1]}
```

Tablet server 3:

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

Tablet server 4:

```
{d-g:{deb:16}}
```
Today: Distributed system design, part 2

- Introduction to distributed systems
  - Motivation: reliability and scalability
  - Replication for reliability
  - Partitioning for scalability

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

- \( \text{map}(f, x[0...n-1]) \)
  - Apply the function \( f \) to each element of list \( x \)

![Diagram of map function](map-reduce-images.png)

- 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
  ```python
reduce(add, [1,2,3,4])
reduce(add, [3,3,4])
reduce(add, [6,4])
reduce(add, [10]) -> 10
```
Reduce with an associative binary function

- If the function $\ast$ is associative, the order $\ast$ 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?
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

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

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

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

MapReduce: (docName, docText)* \rightarrow (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
    - ...

```plaintext
client

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

Map/Reduce worker 1: k-z: {pete:12, reif:42}

Map/Reduce worker 2: a-c: {alice:90, bob:42, cohen:9}

Map/Reduce worker 3: d-g: {deb:16, h-j: }

client
```
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: \texttt{key1} is a person, \texttt{value} is the list of her friends
  - For Reduce: \texttt{key2} is ???, \texttt{values} is a list of ???

\begin{align*}
\text{f1(String key1, String value):} & \quad \text{f2(String key2, Iterator values):} \\
\end{align*}

MapReduce: \((\text{person, friends})^* \rightarrow (\text{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

```java
f1(String key1, String value):
    for each pair of friends in value:
        EmitIntermediate(pair, 1);

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

```
f1(String key1, String value):
f2(String key2, Iterator values):
```

MapReduce: `(docName, docText)* → (docName, list of incoming links)*`
Thursday

• More distributed systems..