Principles of Software Construction: Objects, Design and Concurrency

Distributed System Design, Part 3

Jonathan Aldrich       Charlie Garrod
Administrivia

• Homework 5: The Framework Strikes Back
  ▪ 5c plug-ins due Tuesday, 11:59 p.m.
  ▪ 2 plug-ins for teams of 2 members
  ▪ 4 plug-ins for teams of 3 members
  ▪ Chosen-frameworks available tonight, details via Piazza
Key topics from Tuesday
Key topics from Tuesday

- Failure models
- Distributed system design principles
- Replication and partitioning for reliability and scalability
- Consistent hashing
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

Master:
{a-c:2,
d-g:3,
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:{      }
```

client

Front-end

Front-end
Today

- MapReduce: a robust, scalable framework for distributed computation
Goal: Robust, scalable 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 \( x \)

- 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, [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 \( \ell \) is associative, the order \( \ell \) is applied does not affect the result

\[
1 + ((2+3) + 4) = 1 + (2 + (3+4)) = (1+2) + (3+4)
\]

- 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
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: \( \text{key1} \) is a document name, \( \text{value} \) is the contents of that document
  - For Reduce: \( \text{key2} \) is a word, \( \text{values} \) is a list of the number of counts of that word

\[
\begin{align*}
\text{f1(String key1, String value):} & \\
& \text{for each word } w \text{ in value:} \\
& \quad \text{EmitIntermediate}(w, 1);
\end{align*}
\]

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

- Map: \((\text{key1}, v1) \rightarrow (\text{key2}, v2)^*\)
- Reduce: \((\text{key2}, v2^*) \rightarrow v2^*\)
- MapReduce: \((\text{docName}, \text{docText})^* \rightarrow (\text{word}, \text{wordCount})^*\)
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 ???

$$\begin{align*}
\text{f1}(\text{String key1, String value}) : & \quad \text{f2}(\text{String key2, Iterator values}) : \\
\text{Map: } (\text{key1}, v1) & \rightarrow (\text{key2}, v2)^* \\
\text{Reduce: } (\text{key2}, v2^*) & \rightarrow v2^* \\
\text{MapReduce: } (\text{key1}, v1)^* & \rightarrow (\text{key2}, v2^*)^*
\end{align*}$$

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

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

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

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

MapReduce: (key1, v1)* → (key2, v2*)*

MapReduce: (person, friends)* → (pair of people, count of mutual friends)*
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 `???

\[
\begin{align*}
\text{f1(String key1, String value):} & \quad \text{f2(String key2, Iterator values):} \\
\text{Map: (key1, v1) } & \rightarrow (key2, v2)^* \\
\text{Reduce: (key2, v2*) } & \rightarrow v2^* \\
\text{MapReduce: (key1, v1)* } & \rightarrow (key2, v2*)^* \\
\text{MapReduce: (docName, docText)* } & \rightarrow (docName, list of incoming links)*
\end{align*}
\]
Next week

- Static analysis