Distributed Systems

Day 3: Principles Continued

Jan 31, 2019
Semantic Guarantees of RPCs

• Scenarios:
  • Reading from bank account?
  • Withdrawing from bank account?
  • Depositing into bank account?
Non-Idempotent Procedures

Client → Server
Transfer $1000 from Kyle’s Account to Andy’s Account
Server → Client
Done

Client → Server
Do it again and again and again!
Server → Client
Done

At-most-once semantics

Current Balance: 5000
Current Balance: 4000
Current Balance: 3000
Non-Idempotent Versus Idempotent

Client

Transfer $1000 from Kyle’s Account to Andy’s

Current Balance: 5000

Client

Read Kyle’s Balance

Current Balance: 5000

Client

Do it again and again and again!

Current Balance: 4000

Client

Do it again and again and again!

Current Balance: 4000

Client

Done

Current Balance: 3000

Client

Done

Current Balance: 5000
Semantic Guarantees of RPCs

- **Scenarios:**
  - Reading from bank account?
  - Withdrawing from bank account?
  - Depositing into bank account?

- **Reads versus Writes versus Appends**

- **Solutions:**
  - Idempotent calls
  - Maintain history (to filter duplicates)
Idempotent Procedures

Client -> Server
Write File A Block 0
Client -> Server
Done

Client -> Server
(Ahem ...) Write File A Block 0
Client -> Server
Done

Client

Server

At-least-once semantics
Maintaining History

Transfer $1000 from Kyle’s Account to Andy’s

Done

Do it again!

Done (replay)

At-most-once semantics
History Lost due to Server Crash

- Client
  - Transfer $1000 from Kyle’s Account to Andy’s
  - Client
  - Server
  - Done

- Client
  - Do it again!
  - Client
  - Server

- Client
  - Sorry ... Throws Exception!
  - Client
  - Server

At-most-once semantics

CRASH!!
Semantic Guarantees of RPCs

<table>
<thead>
<tr>
<th>Semantics</th>
<th>Request is lost</th>
<th>Response is lost</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Retransmit?</td>
<td>Filter duplicate?</td>
</tr>
<tr>
<td>At-least-once</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>(1 or more calls)</td>
<td></td>
<td>Re-execute function</td>
</tr>
<tr>
<td>At-most-once</td>
<td>Yes</td>
<td>Yes, Use history to filter</td>
</tr>
<tr>
<td>(0 or 1 calls)</td>
<td></td>
<td>Use history to retransmit</td>
</tr>
</tbody>
</table>
RPC in Projects

Project 1,2,3

We provide RPC Skeletons and you fill in functionality

Project 4

You write RPC skeleton And figure out RPC compilation
Replication and Partitioning
Partition

• Cut a file into ``chunks'' (or sharding)

• Reduce impact of file growth
  • Limits amount of processing

• Definition (size) of chunk is app-specific
Replications

- Make many copies of a file
- Provides fault tolerance
  - Always one copy alive
- Provides more CPU/BW to the file
Dealing with Growth
How to Partition the Data?

**Ways of Index Partitioning**

- **By doc**: each shard has index for subset of docs
  - pro: each shard can process queries independently
  - pro: easy to keep additional per-doc information
  - pro: network traffic (requests/responses) small
  - con: query has to be processed by each shard
  - con: $O(K*N)$ disk seeks for $K$ word query on $N$ shards

- **By word**: shard has subset of words for all docs
  - pro: $K$ word query => handled by at most $K$ shards
  - pro: $O(K)$ disk seeks for $K$ word query
  - con: much higher network bandwidth needed
    - data about each word for each matching doc must be collected in one place
  - con: harder to have per-doc information

---

Use-Case: MapReduce
Distributed Data Processing: Index and Rank the World!!

- Strawman solution:
  - Partition data across servers
  - Have every server process local data

- Why won’t this work?

- Inter-data dependencies:
  - Ranking of a web page depends on ranking of pages that link to it
  - Need data from all users who have a certain gene to evaluate susceptibility to a disease
MapReduce

• Distributed data processing paradigm introduced by Google in 2004
  • Spark is an evolution of MapReduce
  • TensorFlow applies dataflow principles

• MapReduce represents
  • A programming interface for data processing jobs
    • Model processing as a data flow
    • Map and Reduce functions (worker functions)

• A distributed execution framework
  • Scalable and fault-tolerant
Fault Tolerance via Master

Partition data:
* Each partition is a split

Master:
- Sends heartbeats to workers
- If worker doesn’t respond in time period
  Assume dead and start a new worker

Worker does processing
An Example

**Goal**  Find frequent search queries to Bing

**How it Works:**

```
SELECT Query, COUNT(*) AS Freq
FROM QueryTable
HAVING Freq > X
```

What the user says:

- **Read**
- **Map**
- **Reduce**
How do you Ensure Good Performance

• Potential problems:
  • Slow (or bad nodes)
    • Some nodes have bad hardware
  • Data imbalance
    • Some blocks are larger than others
  • Bad Network

• Potential solutions:
  • Detect Problem: periodic status updates
    • If progress is slow → problem
  • Potential Solution problem
    • Restart task on a new node.
Project 1: LiteMiner

- Check for node failures
- Progress reports
- Client
- Pool
- Assign work, balance load between miners: e.g., same number of items to mine
- Progress reports
- Detect slow nodes

Miners

Balance Load between Miners: e.g., same number of items to mine
Project 1: LiteMiner
This Class

• System Model
  • Synchronous v. asynchronous
    • Implications
    • Failure types

• Communication overview
  • Networking basics
  • RPC

• Partition/Replication

• MapReduce use-case