ZooKeeper Continued and MapReduce

Zookeeper and Chubby
Today

• Zookeeper Wrap Up

• Map Reduce (big data analytics at Google)

• Grades

• Closing Remarks

• Critical reviews
## API

<table>
<thead>
<tr>
<th>ZooKeeper</th>
<th>Chubby</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Close/Open()</td>
</tr>
<tr>
<td>delete(path, expectedVersion)</td>
<td>Delete()</td>
</tr>
<tr>
<td>create(path, data, acl, flags)</td>
<td>getContent()</td>
</tr>
<tr>
<td>setData(path, data, expectedVersion)</td>
<td>getContentAndStat()</td>
</tr>
<tr>
<td>getData(path, watch)</td>
<td>use getContent() on directory</td>
</tr>
<tr>
<td>getChildren(path, watch)</td>
<td></td>
</tr>
<tr>
<td>exists(path, watch)</td>
<td></td>
</tr>
<tr>
<td>sync(path)</td>
<td></td>
</tr>
</tbody>
</table>

### Lock Related Calls

- Acquire()/TryAcquire()
- Release

### Sequence Number calls

- Implicitly Managed:
  - Flag passed to create() requests version
  - ZK increments ID after creating files
  - ID is used as expectedVersion
- Explicit Managed:
  - GetSequencer()
  - SetSequencer()
  - CheckSequencer()
**API**

<table>
<thead>
<tr>
<th>ZooKeeper</th>
<th>Chubby</th>
</tr>
</thead>
<tbody>
<tr>
<td>delete(path, expectedVersion)</td>
<td>Close/Open()</td>
</tr>
<tr>
<td>create(path, data, acl, flags)</td>
<td>Delete()</td>
</tr>
<tr>
<td>setData(path, data, expectedVersion)</td>
<td>setContent()</td>
</tr>
<tr>
<td>getData(path, watch)</td>
<td>getContentAndStat()</td>
</tr>
<tr>
<td>getChildren(path, watch)</td>
<td>use getContent() on directory</td>
</tr>
<tr>
<td>exists(path, watch)</td>
<td></td>
</tr>
<tr>
<td>sync(path)</td>
<td></td>
</tr>
</tbody>
</table>

**Lock Related Calls**

- Acquire()/TryAcquire()
- Release

**Sequence Number calls**

- **Implicitly Managed:**
  - Flag passed to create() requests version
  - ZK increments ID after creating files
  - ID is used as expectedVersion

- **Explicit Managed:**
  - GetSequencer()
  - SetSequencer()
  - CheckSequencer()

No locks

No need for open/close because All calls have path in them
Read/Write Interaction

- **Reads:**
  - In Zookeeper can be served locally by any node. In Chubby must go through leader
  - Weaker read semantics

- **Requests:**
  - Clients can send multiple requests at a time
  - Requests served in a FIFO order
    - TCP provides FIFO

- **Writes:**
  - Linearizable (go through leaders)

---

Read: blue
Write: red
**Leader changing Node configuration**
- First delete “/theoapp/ready”
- Clients get notification of deletion. If client is trying to read, once the notification arrives the client stops
- Update “/theoapp/config”
- Create “/theoapp/ready”
- Clients get notification of creation. Now, they know configs have changed and they can read the configs file

**Leader Election (getting Locks)**
- Try to create file: first to create is leader “/theoapp/leader”
- File is ephemeral. If leader dies. File dies.
- Monitor file and when it disappears, you try to create and become leader
- Everyone detects and tries to grab create file at the same time.
- Creates lots of inefficiency

**Group Membership**
- Ephemeral files
- Nodes watch for updates
- Nodes read the children
Zookeeper V. Chubby

• Lessons from Chubby
  – Most requests are read/keep-alive
  – Few developers use locks
  – File system API is easy to use

• Zookeeper = Chubby with weaker read semantics and no locks
  – Weaker read semantics: client can read from any node
  – Writes must still go through leader
  – While no lock, you can implement locks
  – Enable Asynch requests: FIFO execution
MapReduce:
BigData Analytics at Google
Google Environment

• Lots (tens of thousands) of computers
  – all more-or-less equal
    • processor, disk, memory, network interface
  – no specialized servers
  – even if only .01% down at any one moment, many will be down

• Simple jobs become complicated
  – Lots of servers —> scale to many nodes
    • Partition data
    • Partition processing/computing
  – Commodity server —> fault tolerance
MapReduce

• MapReduce: language API library to hide complexity
  – Performance
  – Availability
  – Scalability

• All queries models as Map and Reduce
  – MAP: take a set of data entries and apply an operator on them
  – Reduce: take intermediate data to combine them
MapReduce

- **map**
  - for each pair in a set of key/value pairs, produce a set of new key/value pairs

- **reduce**
  - for each key
    - look at all the values associated with that key and compute a smaller set of values
Implementation Sketch (1)

- **Input (on GFS)**
  - split 0
  - split 1
  - ...
  - split M-1

- **Map phase**: M workers
  - worker
  - worker
  - worker

- **Intermediate files**: (on local disks) partitioned into R pieces
  - worker

- **Reduce phase**: R workers
  - worker
  - worker

- **Output files (on GFS)**
Example

Goal count the number of times a word appears in all documents

map(String key, String value) {
    // key: document name
    // value: document contents
    for each word w in value
        EmitIntermediate(w, 1);
}

reduce(String key, Iterator values) {
    // key: a word
    // values: a list of counts
    for each v in values
        result += v;
    Emit(result);
}
Implementation Sketch (2)

- Map’s input pairs divided into $M$ splits
  - stored in GFS
  - Split allows for parallelism
- Output of Map/Input of Reduce divided into $R$ pieces
- One master process is in charge: farms out work to $W$ ($<< M+R$) worker machines

Three types of processes
* Master
  * Reducer (worker)
  * Mapper (worker)

Input (on GFS)

map
phase:

worker

worker

worker

worker

worker
Implementation Sketch (3)

- Master partitions splits among some of the workers
  - each worker passes pairs to user-supplied map function
  - results stored in local files
    - partitioned into pieces
      - e.g., hash(key) mod R
  - remaining workers perform reduce tasks
    - the R pieces are partitioned among them
    - place remote procedure calls to map workers to get data
    - put output in GFS

```
split 0
split 1
...
split M-1
```

Input (on GFS)

map phase: M workers

intermediate files (on local disks) partitioned into

reduce phase: R workers

Stored locally and will be lost on worker failure

output files (on GFS)
Committing Data

- **Map task**
  - output kept in R local files
  - locations send to master only on task completion

- **Reduce task**
  - output stored on GFS using temporary name
  - file atomically renamed on task completion (to final name)
Coping with Failure (1)

• Master maintains state of each task
  – idle (not started)
  – in progress
  – Completed

• Master pings workers periodically to determine if they’re up
Coping with Failure (2)

• Worker crashes
  – *in-progress* tasks have state set back to idle
    • all output is lost
    • restarted from beginning on another worker
  – *completed* map tasks
    • all output lost
    • restarted from beginning on another worker
    • reduce tasks using output are notified of new worker
Coping with Failure (3)

• Worker crashes (continued)
  – completed reduce tasks
    • output already on GFS
    • no restart necessary

• Master crashes
  – could be recovered from checkpoint
  – in practice
    • master crashes are rare
    • entire application is restarted
Retrospective

• MapReduce → Yahoo Hadoop (now Apache)
  – Years of research created Spark

  1) a giant step backward in the programming paradigm for large-scale data intensive applications
  2) a sub-optimal implementation, in that it uses brute force instead of indexing
  3) not novel at all — it represents a specific implementation of well known techniques developed nearly 25 years ago
  4) missing most of the features that are routinely included in current DBMS
  5) incompatible with all of the tools DBMS users have come to depend on
Current Grading

• HW2 and HW3 done. Will be released today
  – HW2: median: 90, std-dev: 10
  – HW3: median: 91, std-dev: 9

• Projects
  – Tapestry being finished
  – Will start raft grading this weekend
  – Grading take a while due to partial credits
Final Grades

• Course Rubics
  – Projects: 50%
  – HWS: 20%
  – Midterm: 10%
  – Final: 20%

• Individual projects/midterm have been curved

• Final grade will also be curved
Closing Remarks

• Distributed systems: art of providing consensus while tackling failures while providing high performance

• Failure V. Performance V. Correctness/consensus
  – Different type of failures → different implications (different detectors)
    • Mostly heart beats
  
  – Depends on the application: sometimes you don’t need linearizable (total ordering) on all events
    • Zookeeper: reads are not linearizable
    • Dynamo/Cassandra: reads/writes are causally consistent

  – Performance: Shard and distribute
    • Consistent hashing to find shards

• Final on Monday 5/14 at 2pm