PNUTS and Weighted Voting

Vijay Chidambaram
CS 380 D (Feb 8)
PNUTS

• Distributed database built by Yahoo
• Paper describes a production system
• Goals:
  • Scalability
  • Low latency, predictable latency
  • Must handle attacks: flash crowds, denial of service
  • High Availability
  • Eventual Consistency
PNUTS

- Data model: relational table
- Pub-Sub system: Yahoo Message Broker
- Each record has a master
- Uses a guaranteed message delivery service
Data and Query Model

- Relational tables
- Each row has a primary row
- Rows can have binary blobs
- Queries:
  - Point access
  - Range access
Consistency Model

- API
  - Read-any
  - Read-critical(version)
  - Read-latest
  - Write
  - Test-and-set-write(version)
Consistency Model

- Per-record “timeline” consistency
- No multi-record guarantee
- Per-record sequential consistency
- All record operations go to a master
Architecture
Data Storage and Retrieval

- Groups of records are called tablets.
- Each server has 100s-1000s of tablet.
- Each tablet is stored in a single server in a region.
- Tablet size: 100s of MB or a few GBs.
Data Storage

- Storage Unit: get(), scan(), set()
- Message broker is where the update is committed
- Router: identifies which tablet and server contain data
- Ordered data: key range sharded into tablets
- Unordered data: do the same with hash(key)
- Mapping information stored in memory
- True source of mapping info: tablet controller
Yahoo Message Broker (YMB)

- Received messages are logged and replicated
- When update has been applied to all replicas, log is pruned
- YMB servers are present in different regions
- Cross-region traffic is limited to YMB
- Messages are ordered within a YMB region
- Across regions, different ordering is possible
YMB Consistency

- Update considered “committed” once YMB acks it
- A committed update may not be visible to other replicas
- Master replica for a given record is stored inside that record
- Tablet master can be different from record master
- Tablet master serializes updates to record
- Record master is the “true” copy of the data
  - Update is considered “committed” once record master gets it
Recovery

- Request copy
- Checkpoint all inflight updates
- Apply copy
Query Processing

- Scatter-gather engine is used
- Server has the engine, not the client
  - Done to reduce network connections to the server
  - Allows optimization over the whole scatter-gather call
- Range queries are broken up
  - Clients keep a continuation object to continue the range query
Notifications

• User can subscribe to notifications
• Built on top of pub/sub architecture
• Accomplished by talking to the YMB broken
• Each tablet has a topic that user subscribe to
• Whenever tablet is updated or split, notifications can be sent out
PNUTS Applications

- User database
- Social Applications
- Metadata for file systems
- Listings Management
- Session Data
Weighted Voting for Replicas
Updating Replicas

• Goal: you want to replicate data, and read any of the replicas to get the data
• Problem: how do you update the replicas?
• Obvious solution: Write to all replicas
• Can we do better?
• Turns out we can
Quorum-based Reads and Writes

- All reads go to R replicas
- All writes go to W replicas
- As long as we have $R+W>N$, we have strong consistency
  - Why? Condition implies at least one overlapping server between R and W
- We need version numbers to detect which is the latest copy of the data
Weighted Voting

- Weighted Voting is similar to Quorums
- Each server gets N votes instead of 1
- Extra read-only copies get no votes at all
- Each file is assigned some number of votes K
  - If each server gets one vote, this is the number of replicas of the file
- To read, you need R votes.
- To write W votes. Condition: \( R + W > K \)
- Can tune R, W, K per file to meet performance requirements
Guarantees

• Every read will always see the latest write

• Tuning:
  • Condition: \( R + W > K \)
  • \( R = 1 \), reads are efficient, writes are slow
    • Every replica has to be updated
  • \( W = 1 \), writes are efficient, reads are slow
    • Every replica has to be read
  • Most systems are read-heavy, as a result \( R \) is set to between 1 and 3
Tuning

- Giving each server one vote: decentralized quorum system with high availability, low performance
- Giving one server all the votes: centralized system with high performance, low availability
<table>
<thead>
<tr>
<th></th>
<th>Example 1</th>
<th>Example 2</th>
<th>Example 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency (msec)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Representative 1</td>
<td>75</td>
<td>75</td>
<td>75</td>
</tr>
<tr>
<td>Representative 2</td>
<td>65</td>
<td>100</td>
<td>750</td>
</tr>
<tr>
<td>Representative 3</td>
<td>65</td>
<td>750</td>
<td>750</td>
</tr>
<tr>
<td>Voting Configuration</td>
<td>&lt;1, 0, 0&gt;</td>
<td>&lt;2, 1, 1&gt;</td>
<td>&lt;1, 1, 1&gt;</td>
</tr>
<tr>
<td>r</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>w</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Read Latency (msec)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>65</td>
<td>75</td>
<td>75</td>
</tr>
<tr>
<td>Blocking Probability</td>
<td>1.0 × 10^{-2}</td>
<td>2.0 × 10^{-4}</td>
<td>1.0 × 10^{-6}</td>
</tr>
<tr>
<td>Write Latency (msec)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>75</td>
<td>100</td>
<td>750</td>
</tr>
<tr>
<td>Blocking Probability</td>
<td>1.0 × 10^{-2}</td>
<td>1.0 × 10^{-2}</td>
<td>3.0 × 10^{-2}</td>
</tr>
</tbody>
</table>
Weak Representatives

- Possibly stale, read-only copies of the data
- If you read only a weak representative, no guarantees are given about the data
- In others words, it is a local cached copy
Atomicity of operations

- Each read or write is an atomic, isolated operation at each copy
- While the read is going on, there is no other writer at that copy (similarly for writes)
Transactional Isolation

- First lock all files the tx wants to read/write
- Perform reads/writes
- Unlock
- This guarantees serializable transactions
- Obtaining the locks has to be done with a total order, otherwise deadlock is possible
- A tx can hold locks for a max time period
Locks Used

Three locks:
read lock, intention-to-write lock, commit lock

<table>
<thead>
<tr>
<th>Input</th>
<th>No Lock</th>
<th>Read</th>
<th>I-Write</th>
<th>Commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Lock</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Read</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>I-Write</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Commit</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
Violet

- All of this was implemented in the Violet distributed system
- Violet was used to sync personal and private calendars
- Think of it as a very primitive Google Calendar or Outlook Calendar