Chubby lock service

Badri Nath
Rutgers University
badri@cs.rutgers.edu

1. Chubby
2. Cassandra
Service in a cloud

- Services deployed over a cluster
- Thousands of machines over 100s of racks
- Deployed over cheap hardware
- Need to be highly available
- Replication
- State of the system need to be consistent
- Every client need to have consistent view of
- What is where?
- State
- Who can do what?
Big table

- 1 TB of data with 100 MB tablets
- 10000 tablets,
- Spread out over 100s of machines mi
- Tablet-Map(Tabletid, serverID) need to be consistent
- Map changes due to:
  - Tablets can migrate to a different server
  - New tablets can be created
  - Servers can be added and deleted
- Map should be highly available
- Map should be consistent
- Need to solve Distributed consensus
Big table and Chubby: perfect together

- BT uses chubby for
  1. To ensure exactly one master
  2. To store map of blocks, server id
  3. To discover tablet servers
  4. To store schema for each table and ACL
Approaches

- Solve consensus on a case by case basis
  - Paxos, 2 phase commit, 3 phase commit etc
- Multiple implementations, ad-hoc
- Need to be incorporated into application code
- Instead,
  - Implement a common service that all cluster applications that need consensus can use
  - Implement distributed consensus as a service
  - Uniform interface, highly available, scalable
Typical Locking

Client

Lock Manager

lock(resource,type)

grant()/refuse()

operation(resource)

status

unlock()
Distributed locking

- Difficult to get it right
- Blocking – client holding the lock dies or goes into infinite loop
  - How to recover locks?
- Availability – lock manager is a critical resource, need to be Highly Available
  - How to fail-over lock manager state?
- Granularity – fine grain vs coarse grain
  - What to offer?
  - Load and overhead based on rate of lock operations
The Chubby Lock Service

- Consensus achieved by a centralized locking service
  - Chubby is a lock service
- Chubby Lock manager replicated to provide high availability
  - Still at any time one master with failover
- Load on the chubby master minimized by a number of design decisions
- Coarse grained locking
  - Can hold data for long period of times (over large extents of data)
  - Chubby not bothered for row level locks
Chubby Design Choices

- Rationale
  - Lock Service over Distributed Consensus
  - Library vs Service
  - Coarse Grained over Fine Grained
Locking over Distributed Consensus

- All developers need to be aware of consensus algorithm
- Code initially designed without thinking about consensus
- Code/program may start out to be for a small functionality
- Running consensus code on thousands of processes (Paxos??)
- Need to wait on quorums
- Adding consensus as an after thought is painful
Locking

- A single client can obtain a lock and make progress
- Non-Quorum based decisions – independent progress is possible
- Familiarity – locking is a familiar concept to most developers (graduate students!!), at least that is the perception
  - Need to understand implication of distributed systems
- Relives the complexity of implementing consensus
Library vs service

- Library code has to be included in every program..
  - May not be needed initially
- Need to use several APIs in the library
- Code becomes intertwined with consensus code
- Service; clean interface
- Service can be optimized independently
Coarse grained vs fine grained

- Coarse grain:
  - Less interaction with lock manager
  - Hold for longer periods of time
- Less Load on the lock server
- Once locks are acquired to make a consistent change, client is less dependent on the lock server
- Reduces delays
- Brief unavailability of lock manager does not impede progress
- Low lock-acquisition rate implies lot more clients can be serviced
Coarse grain vs Fine grain

- Fine Grained locks – Adding servers and performance can suffer – transaction rate grows
- If required, a coarse grain lock holder can issue fine grain locks
- Row level locks can be handled by a Chubby client that has a tablet level lock that covers the rows
Implementing consensus as a lock service

- Chubby exports a file system interface
- Clients interact with chubby
- Creation of name space and contents together with synchronization (using lock service)
- Maintain a consistent view in spite of failures and concurrent accesses
Client-chubby interaction

1) Open file
2) File handle
3) Lock request
4) Lock
5) Data request
6) Data
7) Lazy update

CLIENT

CHUBBY CELL
Example: create a Master

Hndlmstr=Open(/TBSS/MSTR,RW)
acquire(hndladd,Xclusive)

......
..... Will fail if there is already a lock
..... Live master exists
   elif
setcontents(hndlmstr,masterIPaddress)
Example: create a server

Hndladd=Open(/TBSS/SRVRs,RW)
acquire(hndladd,Xclusive)
Hndlnew=open(hndladd,SRVX)
Acquire(hndlnew,Xclusive)
Setcontents(hndlnew,myIPaddress)
Release(hndladd)
Determine map

- hndl=Open(/crwltbl/root/mt4/usrtb1)
- Utabledict=Getcontents(hndl)
- Utabledict.get(key)
Chubby in a nutshell

- Consensus, consistency problem reduced to keeping a namespace, contents consistent using locks
  - Once you understand this, you understand chubby
- Next problem
- Keeping the name space, content at chubby resilient to failure
- Soln:
  - Replicate the state
  - Have a set of servers capable of handling state
Structure

- Two main components that communicate via RPC
  - Server
  - Library that applications link against.
- Designed for fault tolerance
- Master and 4 replicas
- Master is chosen using paxos (but only among five nodes!!)
- All requests handled by master
- A Replica takes over in case of failure
Structure

client application  chubby library

...  RPCs  master

client application  chubby library

client processes

5 servers of a Chubby cell
Client-Chubby Communication

- Client needs to locate chubby master
  - Client finds the master by sending a master location request to all the replicas listed in the DNS
  - If a non-master replica receives the request, it responds with the master identity
  - Client continues requests to that master until it stops responding or replies that it is no longer the master
Chubby processing

- **Write Requests** –
  - Propagated to all replicas
  - Acknowledged when write request reaches the majority of the replicas

- **Read Requests** –
  - Satisfied by the master alone
  - Safe provided that the master lease has not expired, as no other master exists in the protocol

- Chubby master is the arbiter for all client requests
Master Election

- Replicas use paxos to elect a master
  - Must obtain votes from the majority of replicas and
  - A promise that they will not elect a new master during the Master Lease (a few seconds)
- Replicas check if the node is still the master
  - Lease renewed periodically
Files, Directories and handles

- Chubby exports a file interface
- Chubby maintains a namespace
- Namespace contains files and directories -- nodes
- Tree of files with names separated by slashes – /ls/singer/beyonce/singlelady
- Clients create nodes, add contents to nodes
- Can acquire locks on nodes
Nodes

- Files and directories are nodes
- Can be permanent or ephemeral (cool idea)
- Ephemeral nodes are deleted if no client has them open
- Use ephemeral nodes in chubby to advertise that a client is alive
- \texttt{Hndl=Open(\ldots/myname, ephemeral)}
- \texttt{Setcontents(hndl)=myipaddress}
- Any client can do a \texttt{getcontents( \ldots/myname)} to infer that a machine whose IP is myipaddress is alive
- If the client is dead, then node will not exist
ACL Files

- RWC (read, write, change ACL) modes
- ACLs are files are located in a separate ACL directory
- Each Mode has a file that consist of simple lists of names of principals
- Users are authenticated by a mechanism built into the RPC system
Events

- Client can subscribe to events when a handle is created.
- Delivered asynchronously from the chubby library
- Events include
  - File Contents Modified
  - Child node added, removed, or modified
  - Chubby Master failed over
  - Handle has become invalid
  - Lock Acquired
  - Conflicting lock requested from another client
Session maintenance

- Session between chubby and client
- Maintained using KeepAlives
- Locks, cache, and handles are valid unless invalidated
- KeepAlive – heartbeat RPC from client
- Master can extend lease (12 secs)
- KA response may piggyback cache invalidations
- Session renewal attached to cache invalidations
Caching

- File data and meta-data are cached at the client
- Cache maintained by a lease
- Invalidations keep data consistent
- Master keeps a list of which clients might be caching
  - When file needs to be modified
  - Send invalidation on pending RPCs
  - Proceed after getting ack or client cache has expired
Locks

- Each Chubby file and directory can act as a reader-writer lock –
- One client may hold the lock in exclusive (writer) mode.
- Any number of clients hold the lock in shared (reader) mode.
- Locks are advisory - conflict only with other attempts to acquire the same lock
- To acquire a lock in either mode, you must have write permission - prevents unprivileged reader from delaying a writer
Sequencers

- Locks, handles, cached data become invalid when session expires
- What happens if there is a pending operation delayed in the network
- The master has released the lock on the item
- The operation may arrive later at the master
- If the item is not protected, the delayed operation may operate on inconsistent data
- Use Sequencers
- Getsequencer(handle)
- Gives you a sequence number for the lock held on the handle
Use of sequencer

- Open(node); lock(node)
- GetSequencer()
- Node: sequencer
- Setcontents(node, value, sequencer)
- Checksequencer()
Sequencers

- A client passes the sequencer to the server if it wants for protected operation
- Server checks if the sequencer
  - Is still valid
    - Checked against chubby cache or against most recent sequencer observed
  - Is in the right mode to perform the request
Sequencers for fine grain locking

- Chubby client obtains a coarse grain lock
  - E.g., a table
- Other applications can get a fine grain lock along with sequencer from chubby client
  - E.g, a range of rows
- When the app server writes to chubby server, provide sequencer
- If sequencer has changed (chubby client has lost the lock) operation will be rejected
Scaling

- Chubby serves individual processes
- 90,000 chubby clients
- Scaling enabled with the following
  - Any number of cells can be created
  - Lease times can be increased 12 S to 60 S
  - Aggressive Caching reduces calls to server
Scaling - Future

- Proxies – trusted processes will pass requests from client to chubby cell
  - Can handle KeepAlives and reads
  - Write requests still need to be handled by the cell
  - Write traffic < 1% of the load
  - KeepAlive traffic dominates
- Partitioning – Name space can be partitioned between servers
- Disjoint directories handled by separate cells
Cassandra
Structured Storage System over a P2P Network

Avinash Lakshman, Prashant Malik, Karthik Ranganathan

Slides from Slideshare, presented in SIGMOD 2008, Vancouver, Canada
The Road to 200 Million

Facebook began as a private network for colleges and universities, but has grown into an international social networking site with almost 200 million members. Joe Peron, a member of Facebook's data team, created maps and network diagrams that show the site's expansion and use.

February 2004 to January 2005
Facebook begins at Harvard, and expands to a few universities at a time.

February 2006 to January 2007
Facebook opens registration to allow anyone to join, which brings in older members.

February 2007 to January 2008
Facebook reaches 50 million users, with Canada and Britain growing fast.

February 2008 to January 2009
Facebook is translated into more than 40 languages. The fastest-growing group of members is people over 35.

Depends How You Define 'Friend'

Although most people have a large network of Facebook friends, members maintain real relationships with a much smaller collection of those friends by reading profiles, sending messages and "wall posts." Here is an example of the way one Facebook employee interacts with his network during one month.

TOTAL NETWORK: 175 FRIENDS
He and his friends actively follow the postings of a smaller group.

READ POSTINGS
And send messages to even fewer people.

ONE-WAY POSTS
Of those, only some reciprocate.

REAL "FRIENDS"
What is cassandra

- Storage system for facebook
- 200 million users (March 2009)
- 100 million users (August 2008)
- 30 to 50 TB with 3x replication
- Need to provide sophisticated Inbox search
- Search based on user, thread, subject
What is the problem?

- Lots of relationships between users and data
- Can’t afford to do too many joins
- Denormalize “table” for efficiency
- Increased write traffic
- All you learnt about db design is kaput
Why Cassandra?

- Lots of data
  - Copies of messages, reverse indices of messages, per user data.
- Many incoming requests resulting in a lot of random reads and random writes.
- No existing production ready solutions in the market meet these requirements.
**Design Goals**

- High availability
- Eventual consistency
  - trade-off strong consistency in favor of high availability
- Incremental scalability
- Optimistic Replication
- “Knobs” to tune tradeoffs between consistency, durability and latency
- Low total cost of ownership
- Minimal administration
Cassandra data model

- Entire system is a giant table with lots of rows
- Each row has a key
- Each row has a column family
- Columnfamily = column/super column
- Column = name, value, timestamp
- Super column = name, column +
Writes

- Write to a log and update in memory
- Periodically writes are flushed to disk
- Write optimizations; redundant writes are eliminated
- Writes are sorted to eliminate random seeks
- Hundreds of gigabytes – minimizing seek time
Cassandra Architecture

- Cassandra API
- Tools
- Storage Layer
- Partitioner
- Replicator
- Failure Detector
- Cluster Membership
- Messaging Layer
### Data Model

#### Column Family 1
- **Name**: MailList
- **Type**: Simple
- **Sort**: Name

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>TimeStamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>tid1</td>
<td>&lt;Binary&gt;</td>
<td>t1</td>
</tr>
<tr>
<td>tid2</td>
<td>&lt;Binary&gt;</td>
<td>t2</td>
</tr>
<tr>
<td>tid3</td>
<td>&lt;Binary&gt;</td>
<td>t3</td>
</tr>
<tr>
<td>tid4</td>
<td>&lt;Binary&gt;</td>
<td>t4</td>
</tr>
</tbody>
</table>

#### Column Family 2
- **Name**: WordList
- **Type**: Super
- **Sort**: Time

<table>
<thead>
<tr>
<th>Name</th>
<th>SuperColumn</th>
</tr>
</thead>
<tbody>
<tr>
<td>aloha</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Column</th>
<th>Value</th>
<th>TimeStamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>V1</td>
<td>T1</td>
</tr>
<tr>
<td>C2</td>
<td>V2</td>
<td>T2</td>
</tr>
<tr>
<td>C3</td>
<td>V3</td>
<td>T3</td>
</tr>
<tr>
<td>C4</td>
<td>V4</td>
<td>T4</td>
</tr>
<tr>
<td>C2</td>
<td>V2</td>
<td>T2</td>
</tr>
<tr>
<td>C6</td>
<td>V6</td>
<td>T6</td>
</tr>
</tbody>
</table>

#### Column Family 3
- **Name**: System
- **Type**: Super
- **Sort**: Name

<table>
<thead>
<tr>
<th>Name</th>
<th>SuperColumn</th>
</tr>
</thead>
<tbody>
<tr>
<td>hint1</td>
<td></td>
</tr>
<tr>
<td>hint2</td>
<td></td>
</tr>
<tr>
<td>hint3</td>
<td></td>
</tr>
<tr>
<td>hint4</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Column</th>
<th>Value</th>
<th>TimeStamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>hint1</td>
<td>&lt;List&gt;</td>
<td></td>
</tr>
<tr>
<td>hint2</td>
<td>&lt;List&gt;</td>
<td></td>
</tr>
<tr>
<td>hint3</td>
<td>&lt;List&gt;</td>
<td></td>
</tr>
<tr>
<td>hint4</td>
<td>&lt;List&gt;</td>
<td></td>
</tr>
</tbody>
</table>

---

**Column Families are declared upfront**

**SuperColumns are added and modified dynamically**

**Columns are added and modified dynamically**
Write Operations

- A client issues a write request to a random node in the Cassandra cluster.
  - Asynchronous write as opposed to quorum write
- The “Partitioner” determines the nodes responsible for the data.
- Harvester migrates data to final destinations
- Locally, write operations are logged and then applied to an in-memory version.
- Commit log is stored on a dedicated disk local to the machine.
Write cont’d

Key (CF1, CF2, CF3)

Commit Log
Binary serialized
Key (CF1, CF2, CF3)

Memtable (CF1)
Memtable (CF2)
Memtable (CF2)

Dedicated Disk

Data file on disk

• Data size
• Number of Objects
• Lifetime

<Key name><Size of key Data><Index of columns/supercolumns><Serialized column family>
---
---

BLOCK Index  <Key Name> Offset, <Key Name> Offset
---
---

<Bloom Filter
(Index in memory)

<Key name><Size of key Data><Index of columns/supercolumns><Serialized column family>
Compactions

Index File

Loaded in memory

K1 Offset
K5 Offset
K30 Offset
Bloom Filter

Data File

MERGE SORT

K1 < Serialized data >
K2 < Serialized data >
K3 < Serialized data >
K4 < Serialized data >
K5 < Serialized data >
K10 < Serialized data >
K30 < Serialized data >

Sorted

K1 Offset
K5 Offset
K30 Offset
Bloom Filter

Sorted

Sorted

Sorted

Sorted
Write Properties

- No locks in the critical path
- Sequential disk access
- Behaves like a write through Cache
- Append support without read ahead
- Atomicity guarantee for a key
- "Always Writable"
  - accept writes during failure scenarios
Read

Client

Query

Result

Cassandra Cluster

Closest replica

Result

Replica A

Digest Query

Digest Response

Replica B

Replica C

Read repair if digests differ

Read repair if digests differ
Partitioning And Replication

\[ h(\text{key1}) \]

\[ h(\text{key2}) \]

\[ N=3 \]
Cluster Membership and Failure Detection

- Gossip protocol is used for cluster membership.
- Send changed state to peers
- Node[i]
- KeepAlive[i]=TS
- Pick a random peer j
- Send KeepAlive
- Node j
- If (KeepAlive == OLD):
  - Send to a random peer with probability 1/K
- Else:
  - send to a random peer if KeepAlive
- Every T seconds each member increments its KeepAlive counter and selects one random member to send
Performance Benchmark

- Random and sequential writes - limited by network bandwidth.
- Read performance for Inbox Search in production:

<table>
<thead>
<tr>
<th></th>
<th>Search Interactions</th>
<th>Term Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min</td>
<td>7.69 ms</td>
<td>7.78 ms</td>
</tr>
<tr>
<td>Median</td>
<td>15.69 ms</td>
<td>18.27 ms</td>
</tr>
<tr>
<td>Average</td>
<td>26.13 ms</td>
<td>44.41 ms</td>
</tr>
</tbody>
</table>
Lessons Learnt

- Add fancy features only when absolutely required.
- Many types of failures are possible.
- Big systems need proper systems-level monitoring.
- Value simple designs
Future work

- Atomicity guarantees across multiple keys
- Distributed transactions
- Compression support
- Granular security via ACL’s