CS162
Operating Systems and Systems Programming

Key Value Storage Systems

November 3, 2014
Ion Stoica
Who am I?

• Ion Stoica
  – E-mail: istoica@cs.berkeley.edu
  – Web: http://www.cs.berkeley.edu/~istoica/

• Research focus
  – Cloud computing (Mesos, Spark, Tachyon)
    » Co-director of AMPLab
  – Past work
    » Network architectures (i3, Declarative Networks, …)
    » P2P (Chord, OpenDHT)
Key Value Storage

• Handle huge volumes of data, e.g., PBs
  – Store (key, value) tuples

• Simple interface
  – put(key, value); // insert/write “value” associated with “key”
  – value = get(key); // get/read data associated with “key”

• Used sometimes as a simpler but more scalable “database”
Key Values: Examples

• Amazon:
  – Key: customerID
  – Value: customer profile (e.g., buying history, credit card, ..)

• Facebook, Twitter:
  – Key: UserID
  – Value: user profile (e.g., posting history, photos, friends, …)

• iCloud/iTunes:
  – Key: Movie/song name
  – Value: Movie, Song
Examples

• Amazon
  – DynamoDB: internal key value store used to power Amazon.com (shopping cart)
  – Simple Storage System (S3)

• BigTable/HBase/Hypertable: distributed, scalable data storage

• Cassandra: “distributed data management system” (developed by Facebook)

• Memcached: in-memory key-value store for small chunks of arbitrary data (strings, objects)

• eDonkey/eMule: peer-to-peer sharing system

• …
Key Value Store

- Also called Distributed Hash Tables (DHT)
- Main idea: partition set of key-values across many machines
Challenges

- **Fault Tolerance:** handle machine failures without losing data and without degradation in performance

- **Scalability:**
  - Need to scale to thousands of machines
  - Need to allow easy addition of new machines

- **Consistency:** maintain data consistency in face of node failures and message losses

- **Heterogeneity** (if deployed as peer-to-peer systems):
  - Latency: 1ms to 1000ms
  - Bandwidth: 32Kb/s to 100Mb/s
Key Questions

• **put(key, value)**: where do you store a new (key, value) tuple?

• **get(key)**: where is the value associated with a given “key” stored?

• And, do the above while providing
  – Fault Tolerance
  – Scalability
  – Consistency
Directory-Based Architecture

• Have a node maintain the mapping between keys and the machines (nodes) that store the values associated with the keys
Directory-Based Architecture

- Have a node maintain the mapping between **keys** and the **machines (nodes)** that store the **values** associated with the **keys**

```
Master/Directory

<table>
<thead>
<tr>
<th>K5</th>
<th>N2</th>
</tr>
</thead>
<tbody>
<tr>
<td>K14</td>
<td>N3</td>
</tr>
<tr>
<td>K105</td>
<td>N50</td>
</tr>
</tbody>
</table>

get(K14)  →  N2  ←  V14

N1

| K5  | V5 |
N2

| K14 | V14|
N3

| K105| V105|
N50

...```

11/3/2014 Ion Stoica CS162 ©UCB Fall 2014
Directory-Based Architecture

- Having the master relay the requests → **recursive query**
- Another method: **iterative query** (this slide)
  - Return node to requester and let requester contact node

![Diagram of directory-based architecture](image-url)
Directory-Based Architecture

• Having the master relay the requests → recursive query
• Another method: iterative query
  – Return node to requester and let requester contact node

![Diagram of Directory-Based Architecture]

get(K14)
N3
V14

Master/Directory

<table>
<thead>
<tr>
<th>K5</th>
<th>N2</th>
</tr>
</thead>
<tbody>
<tr>
<td>K14</td>
<td>N3</td>
</tr>
<tr>
<td>K105</td>
<td>N50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N1</th>
</tr>
</thead>
<tbody>
<tr>
<td>K5</td>
</tr>
<tr>
<td>V5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N2</th>
</tr>
</thead>
<tbody>
<tr>
<td>K14</td>
</tr>
<tr>
<td>V14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N3</th>
</tr>
</thead>
<tbody>
<tr>
<td>K105</td>
</tr>
<tr>
<td>V105</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N50</th>
</tr>
</thead>
</table>

…
Discussion: Iterative vs. Recursive Query

• Recursive Query:
  – Advantages:
    » Faster, as typically master/directory closer to nodes
    » Easier to maintain consistency, as master/directory can serialize puts()/gets()
  – Disadvantages: scalability bottleneck, as all “Values” go through master/directory

• Iterative Query
  – Advantages: more scalable
  – Disadvantages: slower, harder to enforce data consistency
Fault Tolerance

- Replicate value on several nodes
- Usually, place replicas on different racks in a datacenter to guard against rack failures
Fault Tolerance

• Again, we can have
  – **Recursive** replication (previous slide)
  – **Iterative** replication (this slide)
Fault Tolerance

- Or we can use recursive query and iterative replication...

```
put(K14, V14)
```

Master/Directory

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>K5</td>
<td>N2</td>
</tr>
<tr>
<td>K14</td>
<td>N1,N3</td>
</tr>
<tr>
<td>K105</td>
<td>N50</td>
</tr>
</tbody>
</table>

N1 N2 N3 N50

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>K14</td>
<td>V14</td>
</tr>
<tr>
<td>K5</td>
<td>V5</td>
</tr>
<tr>
<td>K14</td>
<td>V14</td>
</tr>
<tr>
<td>K105</td>
<td>V105</td>
</tr>
</tbody>
</table>

...
Scalability

• Storage: use more nodes

• Number of requests:
  – Can serve requests from all nodes on which a value is stored in parallel
  – Master can replicate a popular value on more nodes

• Master/directory scalability:
  – Replicate it
  – Partition it, so different keys are served by different masters/directories
    » How do you partition?
Scalability: Load Balancing

• Directory keeps track of the storage availability at each node
  – Preferentially insert new values on nodes with more storage available

• What happens when a new node is added?
  – Cannot insert only new values on new node. Why?
  – Move values from the heavy loaded nodes to the new node

• What happens when a node fails?
  – Need to replicate values from fail node to other nodes
Consistency

• Need to make sure that a value is replicated correctly

• How do you know a value has been replicated on every node?
  – Wait for acknowledgements from every node

• What happens if a node fails during replication?
  – Pick another node and try again

• What happens if a node is slow?
  – Slow down the entire put()? Pick another node?

• In general, with multiple replicas
  – Slow puts and fast gets
Consistency (cont’d)

- If concurrent updates (i.e., puts to same key) may need to make sure that updates happen in the same order

- put(K14, V14’) and put(K14, V14’’) reach N1 and N3 in reverse order

- What does get(K14) return?
  - Undefined!
Consistency (cont’d)

• Large variety of consistency models:
  – Atomic consistency (linearizability): reads/writes (gets/puts) to replicas appear as if there was a single underlying replica (single system image)
    » Think “one updated at a time”
    » Transactions

  – Eventual consistency: given enough time all updates will propagate through the system
    » One of the weakest form of consistency; used by many systems in practice

  – And many others: causal consistency, sequential consistency, strong consistency, …
Quorum Consensus

• Improve put() and get() operation performance

• Define a replica set of size N
  • put() waits for acknowledgements from at least W replicas
  • get() waits for responses from at least R replicas
  • W+R > N

• Why does it work?
  – There is at least one node that contains the update

• Why you may use W+R > N+1?
Quorum Consensus Example

- N=3, W=2, R=2
- Replica set for K14: {N1, N2, N4}
- Assume put() on N3 fails
Quorum Consensus Example

• Now, issuing get() to any two nodes out of three will return the answer.
Scaling Up Directory

• Challenge:
  – Directory contains a number of entries equal to number of (key, value) tuples in the system
  – Can be tens or hundreds of billions of entries in the system!

• Solution: consistent hashing
• Associate to each node a unique id in an uni-dimensional space 0..2^{m}-1
  – Partition this space across m machines
  – Assume keys are in same uni-dimensional space
  – Each (Key, Value) is stored at the node with the smallest ID larger than Key
Key to Node Mapping Example

- \( m = 8 \rightarrow \) ID space: 0..63
- Node 8 maps keys [5,8]
- Node 15 maps keys [9,15]
- Node 20 maps keys [16,20]
- ...  
- Node 4 maps keys [59,4]
Conclusions: Key Value Store

• Very large scale storage systems
• Two operations
  – put(key, value)
  – value = get(key)
• Challenges
  – Fault Tolerance $\rightarrow$ replication
  – Scalability $\rightarrow$ serve get()'s in parallel; replicate/cache hot tuples
  – Consistency $\rightarrow$ quorum consensus to improve put() performance