ZHT
A Fast, Reliable and Scalable Zero-hop Distributed Hash Table

Tonglin Li
Illinois Institute of Technology, Chicago, U.S.A
Intro: Distributed key-value stores

- What is KVS?
- Why to use?
- Why not to use?
- Who’s using it?
- Design issues
What

- A storage system
- A distributed hash table
- Spread simple structured data to multiple node
- Extremely simple API: <key, value>
  - **Read**: `lookup(string key)`
  - **Write**: `insert(string key, string value)`
  - Others: remove, append etc.
Why to use KVS?

- Simple read/write: put/get
- Small requests
- Fast requests
- Good as a building block of scalable systems
- Excellent scalability
  - Capacity
  - Performance
Why NOT to use KVS?

- Need traditional database: SQL
- No transaction
- No complex query:
  - Relational query
  - Partial key query
  - Range query
Who are using/building KVS?

- Amazon: Dynamo, DynamoDB
- LinkedIn: Voldmort
- Facebook: Memcached
- Google: LevelDB, BigTable*
- Apache: Cassandra*, Hbase*
Design issues

- Client request routing
- Partitioning: consistent hashing, mod
- Consistency model
- Membership management
- Failure handling/recovery
Background: Big problems ask for big systems

- High performance computing systems
  - IBM Sequoia in LLNL: 1.5 M cores, 1.5PB memory
- Big data applications in business
  - Facebook: 300 M photos, 500TB data per day (2013)
  - Amazon: 1.5 B items in retail, 0.5 M shoppers per day (2012)
  - Google: processing 20PB data per day (2009)
Big problem: file systems scalability

- Parallel file system (GPFS, PVFS, Lustre)
  - Separated computing resource from storage
  - Centralized metadata management
- Distributed file system (GFS, HDFS)
  - Specific-purposed design (MapReduce etc.)
  - Centralized metadata management
Proposed work

- A distributed hash table (DHT) for HEC
- As building block for high performance distributed systems
- Performance
  - Latency
  - Throughput
- Scalability
- Reliability
Zero-hop hash mapping

### Diagram Description

- **Client 1 ... n**: The clients are distributed across different nodes.
- **MOD num_nodes**: The hash function is applied to the key to determine the node.
- **Hash function**: The hash function is used to calculate the node index.
- **Persistent Hash table**: Each server maintains a persistent hash table for key-value mapping.
- **Replica 1 - Replica 5**: Each server has replicas of the key-value pairs.
- **Node 1 - Node n**: The nodes are labeled with numbers from 1 to n, representing the clients.
- **Value k Replica 1, Value k Replica 2, Value k Replica 3**: Each value is replicated across different nodes.

### Key Concepts

- **Zero-hop**: The mapping is directly from the client to the server without an intermediate hop.
- **Hashing**: Used to distribute key-value pairs across the network.
- **Replication**: Ensures availability and fault tolerance.

### Example

- Suppose a client wants to access a key-value pair. The key is hashed using the hash function and then taken modulo the number of nodes to determine the server.
- The server calculates the server index based on the hash of the key.
- The server then forwards the request to the replica associated with that server.
- The replica returns the value to the client.
- Name space: $2^{64}$
- Physical node
- Manager
- ZHT Instance
- Partition: n (fixed)
  - $n = \max(k)$
Membership management

- Static: Memcached, ZHT
- Dynamic: Cassandra, Riak, ZHT
  - Logarithmic routing: $O(\log_k n)$ hops
    - 1-to-k networking
    - Small routing table: $O(1)$: $k$
  - Constant routing: $O(1)$ hops
    - Direct routing
    - All-to-all networking
Membership management in ZHT

- Update membership
  - Incremental broadcasting
- Remap k-v pairs
  - Traditional DHTs: rehash all influenced pairs
  - ZHT: Moving whole partition
    - HEC has fast local network!
Consistency

- Updating membership tables
  - Planed nodes join and leave: strong consistency
  - Nodes fail: eventual consistency
- Updating replicas
  - Configurable
  - Strong consistency: consistent, reliable
  - Eventual consistency: fast, availability
Failure handling

- Insert and append
  - Send it to next replica
  - Mark this record as primary copy
- Lookup
  - Get from next available replica
- Remove
  - Mark record on all replicas
Evaluation: test beds

- IBM Blue Gene/P supercomputer
  - Up to 8192 nodes
  - 32768 instance deployed
- Commodity Cluster
  - Up to 64 node
- Amazon EC2
  - M1.medium and Cc2.8xlarge
  - 96 VMs, 768 ZHT instances deployed
Latency distribution

<table>
<thead>
<tr>
<th>Scales</th>
<th>75%</th>
<th>90%</th>
<th>95%</th>
<th>99%</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>713</td>
<td>853</td>
<td>961</td>
<td>1259</td>
</tr>
<tr>
<td>256</td>
<td>755</td>
<td>933</td>
<td>1097</td>
<td>1848</td>
</tr>
<tr>
<td>1024</td>
<td>820</td>
<td>1053</td>
<td>1289</td>
<td>3105</td>
</tr>
</tbody>
</table>
Throughput on BG/P

- **TCP: no connection caching**
- **ZHT: TCP connection caching**
- **UDP non-blocking**
- **Memcached**

**Graph Details:**
- **X-axis:** Scale (# of Nodes)
- **Y-axis:** Throughput (ops/s)
- **Data Points:**
  - 1,000
  - 10,000
  - 100,000
  - 1,000,000
  - 10,000,000

**Legend:**
- **TCP: no connection caching**
- **ZHT: TCP connection caching**
- **UDP non-blocking**
- **Memcached**
Aggregated throughput on BG/P
Latency on commodity cluster

Latency (ms)

Scale (# of nodes)

ZHT
Cassandra
Memcached
ZHT on cloud: latency

<table>
<thead>
<tr>
<th>Node number</th>
<th>Average latency in micro seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ZHT on m1.medium instance (1/node)</td>
</tr>
<tr>
<td></td>
<td>ZHT on cc2.8xlarge instance (8/node)</td>
</tr>
<tr>
<td></td>
<td>DynamoDB</td>
</tr>
</tbody>
</table>

- ZHT on m1.medium instance (1/node)
- ZHT on cc2.8xlarge instance (8/node)
- DynamoDB
**ZHT on cloud: latency distribution**

### ZHT on cc2.8xlarge instance

8 s-c pair/instance

<table>
<thead>
<tr>
<th>Scales</th>
<th>75%</th>
<th>90%</th>
<th>95%</th>
<th>99%</th>
<th>Avg</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>186</td>
<td>199</td>
<td>214</td>
<td>260</td>
<td>172</td>
<td>46412</td>
</tr>
<tr>
<td>32</td>
<td>509</td>
<td>603</td>
<td>681</td>
<td>1114</td>
<td>426</td>
<td>75080</td>
</tr>
<tr>
<td>128</td>
<td>588</td>
<td>717</td>
<td>844</td>
<td>2071</td>
<td>542</td>
<td>236065</td>
</tr>
<tr>
<td>512</td>
<td>574</td>
<td>708</td>
<td>865</td>
<td>3568</td>
<td>608</td>
<td>841040</td>
</tr>
</tbody>
</table>

### DynamoDB: 8 clients/instance

<table>
<thead>
<tr>
<th>Scales</th>
<th>75%</th>
<th>90%</th>
<th>95%</th>
<th>99%</th>
<th>Avg</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>11942</td>
<td>13794</td>
<td>20491</td>
<td>35358</td>
<td>12169</td>
<td>83.39</td>
</tr>
<tr>
<td>32</td>
<td>10081</td>
<td>11324</td>
<td>12448</td>
<td>34173</td>
<td>9515</td>
<td>3363.11</td>
</tr>
<tr>
<td>128</td>
<td>10735</td>
<td>12128</td>
<td>16091</td>
<td>37009</td>
<td>11104</td>
<td>11527</td>
</tr>
<tr>
<td>512</td>
<td>9942</td>
<td>13664</td>
<td>30960</td>
<td>38077</td>
<td>28488</td>
<td>ERROR</td>
</tr>
</tbody>
</table>
ZHT on cloud: throughput

![Graph showing aggregated throughput and hourly cost for different configurations of ZHT and DynamoDB on cloud instances.](image-url)

- **ZHT cost, m1**
- **ZHT cost, cc2**
- **DynamoDB cost (10k ops/s provision)**
- **DynamoDB**
- **ZHT on cc2.8xlarge instance (8/node)**
- **ZHT on m1.medium instance (1/node)**
Amortized cost

<table>
<thead>
<tr>
<th>Hourly cost for 1K ops/s throughput in US dollar</th>
<th>ZHT on m1.medium instance (1/node)</th>
<th>ZHT on cc2.8xlarge instance (8/node)</th>
<th>DynamoDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.01</td>
<td>0.1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0.1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>1.0</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>16</td>
<td>3.2</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>32</td>
<td>6.4</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>64</td>
<td>9.6</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>96</td>
<td>12.8</td>
<td>18</td>
<td>64</td>
</tr>
</tbody>
</table>
Applications

- **FusionFS**
  - A distributed file system
  - Metadata: ZHT

- **IStore**
  - A information dispersal storage system
  - Metadata: ZHT

- **MATRIX**
  - A distributed many-Task computing execution framework
  - ZHT is used to submit tasks and monitor the task execution status
The bottleneck of file systems

- Metadata

Concurrent file creates

<table>
<thead>
<tr>
<th>Scale (# of Cores)</th>
<th>File Create (GPFS Many Dir)</th>
<th>File Create (GPFS One Dir)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>28</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
<td>16</td>
</tr>
<tr>
<td>256</td>
<td>256</td>
<td>64</td>
</tr>
<tr>
<td>1024</td>
<td>1024</td>
<td>256</td>
</tr>
<tr>
<td>4096</td>
<td>4096</td>
<td>1024</td>
</tr>
<tr>
<td>16384</td>
<td>16384</td>
<td>4096</td>
</tr>
</tbody>
</table>
FusionFS result: Concurrent File Creates
## Related work: Distributed Hash Tables

- Many DHTs: Chord, Kademlia, Pastry, Cassandra, C-MPI, Memcached, Dynamo...
- Why another?

<table>
<thead>
<tr>
<th>Name</th>
<th>Impl.</th>
<th>Routing Time</th>
<th>Persistence</th>
<th>Dynamic membership</th>
<th>Append Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cassandra</td>
<td>Java</td>
<td>Log(N)</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>C-MPI</td>
<td>C</td>
<td>Log(N)</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Dynamo</td>
<td>Java</td>
<td>O to Log(N)</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Memcached</td>
<td>C</td>
<td>O</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>ZHT</td>
<td>C++</td>
<td>O to 2</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Conclusion

- ZHT: A distributed Key-Value store
  - light-weighted
  - high performance
  - Scalable
  - Dynamic
  - Fault tolerant
  - Versatile: works from clusters, to clouds, to supercomputers
Questions?

Tonglin Li

tli13@hawk.iit.edu

http://datasys.cs.iit.edu/projects/ZHT/