DHTs and Cloud Computing

April 25, 2011
Ion Stoica
http://inst.eecs.berkeley.edu/~cs162

Distributed Hash Tables (DHTs)

- Distribute (partition) a hash table data structure across a large number of servers
  - Also called, key-value store

- Two operations
  - put(key, data); // insert “data” identified by “key”
  - data = get(key); // get data associated to “key”

Distributed Hash Tables (DHTs) (cont’d)

- Just need a lookup service, i.e., given a key (ID), map it to machine n
  n = lookup(key);

- Invoking put() and get() at node m

  m.put(key, data) {
      n = lookup(key); // get node “n” mapping “key”
      n.store(key, data); // store data at node “n”
  }

  data = m.get(key) {
      n = lookup(key); // get node “n” storing data associated to “key”
      return n.retrieve(key); // get data stored at “n” associated to “key”
  }

Distributed Hash Tables (DHTs) (cont’d)

- Many lookup proposals: CAN, Chord, Pastry, Tapestry, Kademlia, …

- Used in practice:
  - p2p: eDonkey (based on Kademlia)
  - Dynamo (Amazon)
  - Cassandra (Facebook)
  - …
Challenges

- System churn: machines can fail or exit the system any time
- Scalability: need to scale to 10s or 100s of thousands of machines
- Heterogeneity:
  - Latency: 1ms to 1000ms
  - Bandwidth: 32Kb/s to 100Mb/s
  - Nodes stay in system from 10s to a year

Chord Lookup Service

- Associate to each node and item a unique id/key in an uni-dimensional space 0..2^m-1
- Partition this space across N machines
- Each id is mapped to the node with the smallest largest ID (consistent hashing)
- Key design decision
  - Decouple correctness from efficiency
- Properties
  - Routing table size O(log(N)), where N is the total number of nodes
  - Guarantees that a file is found in O(log(N)) steps

Identifier to Node Mapping Example (Consistent hashing)

- Node 8 maps [5,8]
- Node 15 maps [9,15]
- Node 20 maps [16,20]
- Node 4 maps [59, 4]
- Each node maintains a pointer to its successor

Lookup

- Each node maintains pointer to its successor
- Route packet (ID, data) to the node responsible for ID using successor pointers
- E.g., node=4 lookups for node responsible for ID=37
Stabilization Procedure

- Periodic operation performed by each node $n$ to maintain its successor when new nodes join the system

```plaintext
n.stabilize()
  x = succ.pred;
  if (x ∈ (n, succ))
    succ = x;  // if x better successor, update
    succ.notify(n); // n tells successor about itself

n.notify(n')
  if (pred = nil or n' ∈ (pred, n))
    pred = n';  // if n' is better predecessor, update
```

Joining Operation

- Node with id=50 joins the ring
- Node 50 needs to know at least one node already in the system
  - Assume known node is 15

- $n=50$ sends join(50) to node 15
- $n=44$ returns node 58
- $n=50$ updates its successor to 58

- $n=50$ executes stabilize()
- $n$'s successor (58) returns $x = 44$
Joining Operation

- \( n=50 \) executes stabilize()
  - \( x = 44 \)
  - \( \text{succ} = 58 \)

\[
\begin{align*}
&\text{n.stabilize()}
\text{x} = \text{succ} \cdot \text{pred}; \\
&\text{if } (x \in (n, \text{succ}))
\text{succ} = x; \\
&\text{succ.notify(n);} \\
&\text{n.notify(n')} \text{ if } (\text{pred} = \text{nil} \text{ or } n' \in (\text{pred}, n)); \\
&\text{pred} = n'
\end{align*}
\]
Joining Operation

- n = 44 runs stabilize()
- n's successor (58) returns x = 50

\[ x = \text{succ}.\text{pred}; \]
\[ \text{if} \ (x \subseteq (n, \text{succ})) \]
\[ \text{succ} = x; \]
\[ \text{succ.notify}(n); \]

Joining Operation

- n = 44 runs stabilize()
- succ = 58
- pred = 35

\[ x = 50 \]
\[ \text{succ.notify}(n); \]

Joining Operation

- n = 44 sends notify(44) to its successor
- succ = 58
- pred = 35

\[ x = \text{succ}.\text{pred}; \]
\[ \text{if} \ (x \subseteq (n, \text{succ})) \]
\[ \text{succ} = x; \]
\[ \text{succ.notify}(n); \]
Joining Operation

- $n=50$ processes
  - notify(44)
  - pred = nil

```
n.notify(n')
  if (pred = nil or n'∈ (pred, n))
  pred = n'
n.succ = succ
  pred = succ
```

Joining Operation (cont’d)

- This completes the joining operation!

```
n.notify(n')
  if (pred = nil or n'∈ (pred, n))
  pred = n'
n.succ = succ
  pred = succ
```

Achieving Efficiency: finger tables

Finger Table at 80

<table>
<thead>
<tr>
<th>i</th>
<th>$f[i]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>96</td>
</tr>
<tr>
<td>1</td>
<td>96</td>
</tr>
<tr>
<td>2</td>
<td>96</td>
</tr>
<tr>
<td>3</td>
<td>96</td>
</tr>
<tr>
<td>4</td>
<td>96</td>
</tr>
<tr>
<td>5</td>
<td>112</td>
</tr>
<tr>
<td>6</td>
<td>20</td>
</tr>
</tbody>
</table>

Say $m=7$

```
80 + 2^m = 16
```

```
\text{ith entry at peer with id } n \text{ is first peer with id } \geq n + 2^m (mod 2^m)
```
**Achieving Robustness**

- To improve robustness each node maintains the $k$ (> 1) immediate successors instead of only one successor.

- Successor $S$ of a node $N$ can send its $K$-1 successors to $N$ during $N$’s stabilize() procedure.

---

**Administrivia**

- Project 4 design due tomorrow: Tuesday, April 26.

- Project 3 code available.

- Final exam: Friday, May 13, 8-11am (2060 VLSB)
  - Provide some exam question examples next lecture.

---

**What is Cloud Computing?**

- “Cloud” refers to large Internet services running on 10,000s of machines (Google, Facebook, etc).

- “Cloud computing” refers to services by these companies that let external customers rent cycles:
  - Amazon EC2: virtual machines at 8.5¢/hour, billed hourly
  - Amazon S3: storage at 10-15¢/GB/month
  - Windows Azure: applications using Azure API

- Attractive features:
  - Scale: 100s of nodes available in minutes
  - Fine-grained billing: pay only for what you use
  - Ease of use: sign up with credit card, get root access

---

**What Can You Run in Cloud Computing?**

- Almost everything!
- Virtual Machine instances
- Storage services
  - Simple Storage Service (S3)
  - Elastic Block Storage (RBS)
- Databases:
  - Database instances (e.g., mySQL, SQL Server, …)
  - SimpleDB
- Content Distribution Network: CloudFront
- **MapReduce**: Amazon Elastic MapReduce
- …
What is MapReduce?

- Data-parallel programming model for clusters of commodity machines
- Pioneered by Google
  - Processes 20 PB of data per day
- Popularized by Apache Hadoop project
  - Used by Yahoo!, Facebook, Amazon, ...

What is MapReduce Used For?

- At Google:
  - Index building for Google Search
  - Article clustering for Google News
  - Statistical machine translation
- At Yahoo!
  - Index building for Yahoo! Search
  - Spam detection for Yahoo! Mail
- At Facebook:
  - Data mining
  - Ad optimization
  - Spam detection

Example: Facebook Lexicon (discontinued, February 2010)

www.facebook.com/lexicon
MapReduce Goals

- **Scalability** to large data volumes:
  - Scan 100 TB on 1 node @ 50 MB/s = 24 days
  - Scan on 1000-node cluster = 35 minutes

- **Cost-efficiency:**
  - Commodity nodes (cheap, but unreliable)
  - Commodity network (low bandwidth)
  - Automatic fault-tolerance (fewer admins)
  - Easy to use (fewer programmers)

Typical Hadoop Cluster

- 40 nodes/rack, 1000-4000 nodes in cluster
- 1 Gbps bandwidth in rack, 8 Gbps out of rack
- Node specs (Facebook):
  - 8-16 cores, 32-48 GB RAM, 10×2TB disks

Challenges of Cloud Environment

- Cheap nodes fail, especially when you have many
  - Mean time between failures for 1 node = 3 years
  - MTBF for 1000 nodes = 1 day
  - **Solution:** Build fault-tolerance into system

- Commodity network = low bandwidth
  - **Solution:** Push computation to the data

- Programming distributed systems is hard
  - **Solution:** Restricted programming model: users write data-parallel “map” and “reduce” functions, system handles work distribution and failures
Hadoop Components

- Distributed file system (HDFS)
  - Single namespace for entire cluster
  - Replicates data 3x for fault-tolerance
- MapReduce framework
  - Runs jobs submitted by users
  - Manages work distribution & fault-tolerance
  - Colocated with file system

Hadoop Distributed File System (HDFS)

- Files split into 128MB blocks
- Blocks replicated across several datanodes (often 3)
- Namenode stores metadata (file names, locations, etc)
- Optimized for large files, sequential reads
- Files are append-only

MapReduce Programming Model

- Data type: key-value records

- Map function:
  \[(K_{in}, V_{in}) \rightarrow \text{list}(K_{inter}, V_{inter})\]

- Reduce function:
  \[(K_{inter}, \text{list}(V_{inter})) \rightarrow \text{list}(K_{out}, V_{out})\]

Example: Word Count

```python
def mapper(line):
    for word in line.split():
        output(word, 1)

def reducer(key, values):
    output(key, sum(values))
```
Word Count Execution

An Optimization: The Combiner

- Local reduce function for repeated keys produced by same map
- For associative ops. like sum, count, max
- Decreases amount of intermediate data

- Example: local counting for Word Count:

```python
def combiner(key, values):
    output(key, sum(values))
```

Word Count with Combiner

MapReduce Execution Details

- Mappers preferentially scheduled on same node or same rack as their input block
  - Minimize network use to improve performance
- Mappers save outputs to local disk before serving to reducers
  - Allows recovery if a reducer crashes
  - Allows running more reducers than # of nodes
Fault Tolerance in MapReduce

1. If a task crashes:
   – Retry on another node
     » OK for a map because it had no dependencies
     » OK for reduce because map outputs are on disk
   – If the same task repeatedly fails, fail the job or ignore that input block

➢ Note: For the fault tolerance to work, user tasks must be deterministic and side-effect-free

Fault Tolerance in MapReduce

2. If a node crashes:
   – Relaunch its current tasks on other nodes
   – Relaunch any maps the node previously ran
     » Necessary because their output files were lost along with the crashed node

Fault Tolerance in MapReduce

3. If a task is going slowly (straggler):
   – Launch second copy of task on another node
   – Take the output of whichever copy finishes first, and kill the other one

• Critical for performance in large clusters (many possible causes of stragglers)

Takeaways

• By providing a restricted data-parallel programming model, MapReduce can control job execution in useful ways:
  – Automatic division of job into tasks
  – Placement of computation near data
  – Load balancing
  – Recovery from failures & stragglers
Conclusions

• The key challenge of building wide area P2P systems is a scalable and robust directory/lookup service
  – Napster: centralized location service
  – Gnutella: broadcast-based decentralized location service
  – CAN, Chord, Tapestry, Pastry: efficient-routing decentralized solution

• Cloud computing
  – Pay-as-you go services
  – Rapidly scale up the service
  – Commodity hardware, large scale: failures become the norm
  – MapReduce: Data-parallel programming model for clusters of commodity machines