Outline

• Quorum consensus

• Chord

• Mesos

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 might you use W+R > N+1?

Quorum Consensus Example

• N=3, W=2, R=2
• Replica set for K14: \{N_1, N_3, N_4\}
• Assume put() on N_3 fails
Quorum Consensus Example

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

| N_1 | get(K14) | V14 |
| N_2 | get(K14) | NIL |
| N_3 |
| N_4 |

Outline

• Quorum consensus

• Chord

• Mesos

Scaling Up Directory – Challenges

• 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!

Scaling Up Directory – Strawman Solution

• Assume a system with m nodes

• Store (key, value) on node i = key mod M + 1

• No need to keep any metadata!

• Challenge: what happened if you take away a node or add a new node?
Scaling Up Directory – 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 = 6 \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]

Scaling Up Directory

- With consistent hashing, directory contains only a number of entries equal to number of nodes
  - Much smaller than number of tuples
- Next challenge: every query still needs to contact the directory

- Solution: distributed directory (a.k.a. lookup) service:
  - Given a key, find the node storing value associated to the key

- Key idea: route request from node to node until reaching the node storing the request’s key

- Key advantage: totally distributed
  - No point of failure; no hot spot

Chord: Distributed Lookup (Directory) Service

- Key design decision
  - Decouple correctness from efficiency

- Properties
  - Each node needs to know about \( O(\log(M)) \), where \( M \) is the total number of nodes
  - Guarantees that a tuple is found in \( O(\log(M)) \) steps

- Many other lookup services: CAN, Tapestry, Pastry, Kademlia, …
Lookup

• Each node maintains pointer to its successor
• Route packet (Key, Value) to the node responsible for ID using successor pointers
• E.g., node=4 lookups for node responsible for Key=37

Stabilization Procedure

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

```
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 n' is better predecessor, update
```
§ $n = 50$ executes `stabilize()

§ $n$'s successor (58) returns $x = 44$

$n$.stabilize()

$x = \text{succ.pred}$;

if ($x \subseteq (n, \text{succ})$

    succ = $x$;

    succ.notify($n$);

$n = 50$ executes `stabilize()"

$x = 44$

succ = 58

pred = nil

$n$.stabilize()

$x = \text{succ.pred}$;

if ($x \subseteq (n, \text{succ})$

    succ = $x$;

    succ.notify($n$);

$n = 50$ sends to it's successor (58) `notify(50)`

$n = 58$ processes `notify(50)`

pred = 44

$n' = 50$

$n$.notify($n'$)

if (pred = nil or $n' \subseteq (\text{pred}, n)$)

    pred = $n'$
Joining Operation

- \( n = 58 \) processes
  - notify(50)
  - \( \text{pred} = 44 \)
  - \( n' = 50 \)
  - set pred = 50

\[ n.\text{notify}(n') \]
if (pred = nil or \( n' \subseteq (\text{pred}, n) \))
\[ \text{pred} = n' \]

Joining Operation

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

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

Joining Operation

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

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

- $n=44$ runs stabilize()
- $n=44$ sends notify(44) to its successor

```
$n$.stabilize()
x = succ.pred;
if (x ∈ (n, succ))
succ = x;
succ.notify(n);
```

Joining Operation

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

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

Joining Operation (cont'd)

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

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

This completes the joining operation!
Achieving Efficiency: finger tables

Finger Table at 80

Say \( m=7 \)

\[(80 + 2^i) \mod 2^m = 16\]

\( i \)th entry at node with id \( n \) is first peer with id \( \geq n + 2^i \) (mod \( 2^m \))

Achieving Fault Tolerance for Lookup Service

- To improve robustness each node maintains the \( k (>1) \) immediate successors instead of only one successor.
- In the \( \text{pred()} \) reply message, node A can send its \( k-1 \) successors to its predecessor B.
- Upon receiving \( \text{pred()} \) message, B can update its successor list by concatenating the successor list received from A with its own list.
- If \( k = \log(M) \), lookup operation works with high probability even if half of nodes fail, where \( M \) is number of nodes in the system.

Storage Fault Tolerance

- Replicate tuples on successor nodes.
- Example: replicate \((K14, V14)\) on nodes 20 and 32.
- If node 15 fails, no reconfiguration needed:
  - Still have two replicas
  - All lookups will be correctly routed.
- Will need to add a new replica on node 35.
**Administrivia**

- Midterm 3 coming up on Wen 11/29 6:30-8PM
  - All topics up to and including Lecture 25
    - Focus will be on Lectures 17 – 25 and associated readings, and Projects 3
    - But expect 20-30% questions from materials from Lectures 1-16
  - Closed book
  - 2 sides hand-written notes both sides

**Outlines**

- Quorum consensus

- Chord

- Mesos

**Data Deluge**

- Billions of users connected through the net
  - WWW, FB, twitter, cell phones, …
  - 80% of the data on FB was produced last year
  - FB building Exabyte ($2^{60} = 10^{18}$) data centers

- It’s all happening online — could record every:
  - Click, ad impression, billing event, server request, transaction, network msg, fault, fast forward, pause, skip, …

- User Generated Content (Web & Mobile)
  - Facebook, Instagram, Yelp, TripAdvisor, Twitter, YouTube, …
Data Grows Faster than Moore’s Law

Projected Growth

- Moore’s Law
- Particle Accel.
- DNA Sequencers

And Moore’s law is ending!!

The Big Data Solution: Cloud Computing

- One machine can not process or even store all the data!
- Solution: distribute data over cluster of cheap machines
- Cloud Computing provides:
  - Illusion of infinite resources
  - Short-term, on-demand resource allocation
  - Can be much less expensive than owning computers
  - Access to latest technologies (SSDs, GPUs, …)

The Berkeley AMPLab

- January 2011 – 2016
  - 8 faculty
  - > 50 students
  - 3 software engineer team
- Organized for collaboration

AMPCamp (since 2012)

3 day retreats (twice a year)

400+ campers (100s companies)

What Can You do with Big Data?

- Crowdsourcing
- Physical modeling
- Sensing
- Data Assimilation

= http://traffic.berkeley.edu
The Berkeley AMPLab

- Governmental and industrial funding:

Goal: Next generation of open source data analytics stack for industry & academia:
Berkeley Data Analytics Stack (BDAS)

Generic Big Data Stack

Processing Layer

Resource Management Layer

Storage Layer

Hadoop Stack

- Hive
- Pig
- Storm
- Impala
- Giraph
- HadoopMR
- Yarn
- HDFS

BDAS Stack

- Spark Core
- Spark Streaming
- Spark SQL
- SparkR
- GraphX
- MLBase
- MLlib
- Velox

- Mesos
- Hadoop Yarn
- Succinct
- Tachyon
- HDFS, S3, Ceph, …

- BDAS Stack
- 3rd party
Today’s Lecture

<table>
<thead>
<tr>
<th>Spark Streaming</th>
<th>Spark Core</th>
<th>Storage/Res. Mgmt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Succinct</td>
<td>Sample</td>
<td>Clean</td>
</tr>
<tr>
<td>Spark</td>
<td>SparkR</td>
<td>Velox</td>
</tr>
<tr>
<td>SQL</td>
<td>GraphX</td>
<td>HDFS, S3, Ceph, …</td>
</tr>
<tr>
<td>Spark</td>
<td>MLBase</td>
<td></td>
</tr>
<tr>
<td>MLlib</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MLBase</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Succinct</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tachyon</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

BDAS Stack

3rd party

Summary

- Mesos
- Spark

Motivation

- Rapid innovation in cloud computing (aka 2008)
- No single framework optimal for all applications
- Each framework runs on its dedicated cluster or cluster partition

A Short History

- Started at UC Berkeley in Spring 2009
  - A class project of cs294 (Cloud Computing: Infrastructure, Services, and Applications)
- Open Source: 2010
- Apache Project: 2011
- Today: one of the most popular cluster resource management systems (OS for datacenters)
Computation Model: Frameworks

- A framework (e.g., Hadoop, MPI) manages one or more jobs in a computer cluster.
- A job consists of one or more tasks.
- A task (e.g., map, reduce) is implemented by one or more processes running on a single machine.

<table>
<thead>
<tr>
<th>Job 1: tasks 1, 2, 3, 4</th>
<th>Job 2: tasks 5, 6, 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Executor 1</td>
<td>Executor 1</td>
</tr>
<tr>
<td>task 1</td>
<td>task 5</td>
</tr>
<tr>
<td>task 2</td>
<td>task 6</td>
</tr>
<tr>
<td>Executor 2</td>
<td>Executor 2</td>
</tr>
<tr>
<td>task 3</td>
<td>task 7</td>
</tr>
<tr>
<td>Executor 3</td>
<td>Executor 3</td>
</tr>
<tr>
<td>task 4</td>
<td>(e.g., Job Tracker)</td>
</tr>
</tbody>
</table>

One Framework Per Cluster Challenges

- Inefficient resource usage
  - E.g., Hadoop cannot use available resources from Pregel’s cluster
  - No opportunity for stat. multiplexing
- Hard to share data
  - Copy or access remotely, expensive
- Hard to cooperate
  - E.g., Not easy for Pregel to use graphs generated by Hadoop

Solution: Apache Mesos

- Common resource sharing layer
  - abstracts (“virtualizes”) resources to frameworks
  - enable diverse frameworks to share cluster

Fine Grained Resource Sharing

- Task granularity both in time & space
  - Multiplex node/time between tasks belonging to different jobs/frameworks
- Tasks typically short; median ~= 10 sec, minutes
- Why fine grained?
  - Improve data locality
  - Easier to handle node failures
**Mesos Goals**

- High utilization of resources
- Support diverse frameworks (existing & future)
- Scalability to 10,000's of nodes
- Reliability in face of node failures
- Focus of this talk: resource management & scheduling

**Approach: Global Scheduler**

Organization policies
Resource availability
Job requirements
- Response time
- Throughput
- Availability
- ...

Global Scheduler

Job execution plan
- Task DAG
- Inputs/outputs

Approach: Global Scheduler

Organization policies
Resource availability
Job requirements
Job execution plan
Estimates
- Task durations
- Input sizes
- Transfer sizes
**Approach: Global Scheduler**

- **Advantages:** can achieve optimal schedule
- **Disadvantages:**
  - Complexity → hard to scale and ensure resilience
  - Hard to anticipate future frameworks’ requirements
  - Need to refactor existing frameworks

**Resource Offers**

- **Unit of allocation:** resource offer
  - Vector of available resources on a node
  - E.g., node1: <1CPU, 1GB>, node2: <4CPU, 16GB>

- Master sends resource offers to frameworks
- Frameworks select which offers to accept and which tasks to run

Push task scheduling to frameworks

**Our Approach: Distributed Scheduler**

- **Advantages:**
  - Simple → easier to scale and make resilient
  - Easy to port existing frameworks, support new ones

- **Disadvantages:**
  - Distributed scheduling decision → not optimal

**Mesos Architecture: Example**

- Slaves continuously send status updates about resources
- Framework executors launch tasks and may persist across tasks
- Framework scheduler selects resources and provides tasks

- Pluggable scheduler to pick framework to send an offer to
- Task allocation:
Why does it Work?

- A framework can just wait for an offer that matches its constraints or preferences!
  - Reject offers it does not like

- Example: Hadoop’s job input is blue file

Accept: both S2 and S3 store the blue file

Reject: S1 doesn’t store blue file

Accept: both S2 and S3 store the blue file

Dynamic Resource Sharing

- 100 node cluster

Apache Mesos Today

- Hundreds of contributors

- Hundreds of deployments in productions
  - E.g., Twitter, GE, Apple
  - Managing 10K node datacenters!

- Mesosphere, startup to commercialize Apache Spark

Summary (1/2)

- Quarum consensus:
  - N replicase
  - Write to W replicas, read from R, where W + R > N
  - Tolerate one failure

- Chord:
  - Highly scalable distributed lookup protocol
  - Each node needs to know about O(log(M)), where M is the total number of nodes
  - Guarantees that a tuple is found in O(log(M)) steps
  - Highly resilient: works with high probability even if half of nodes fail
Summary (2/2)

- Mesos: large scale resource management systems:
  - Two-level schedulers
  - First level: performance isolation across multiple frameworks, models
  - Second level: frameworks decide which tasks to schedule, when, and where