Scalable Consistency in Scatter
A Distributed Key-Value Storage System

Lisa Glendenning
Ivan Beschastnikh
Arvind Krishnamurthy
Thomas Anderson

University of Washington

Supported by NSF CNS-0963754

SOSP October 2011
Internet services depend on distributed key-value stores.
Scatter:
Goals

✓ linearizable consistency semantics
✓ scalable in a wide area network
✓ high availability
✓ performance close to existing systems
Scatter: Approach

combine ideas from:

- **scalable peer-to-peer systems**
  - ✓ distributed hash table
  - ✓ self-organization
  - ✓ decentralization

- **consistent datacenter systems**
  - ✓ consensus
  - ✓ replication
  - ✓ transactions
Distributed Hash Tables: Background

**core functionality:** partition and assign keys to nodes

**system structure:** knowledge of system state is distributed among all nodes

**system management:**
- nodes coordinate locally to respond to churn, e.g.,
  - give keys to new nodes
  - take over keys of failed nodes

links between nodes form overlay
Example: c joins between a and b

c.pred = a
c.succ = b
a.succ = c
b.pred = c
b.keys = (k_c, k_b]
c.keys = (k_a, k_c]
Distributed Hash Tables: Faults Cause Inconsistencies

**Example:** c joins between a and b

![Diagram](Diagram.png)

**what could go wrong?**

<table>
<thead>
<tr>
<th>FAULT</th>
<th>OUTCOME</th>
</tr>
</thead>
<tbody>
<tr>
<td>communication fault between b and c</td>
<td>both b and c claim ownership of ((k_a, k_c))</td>
</tr>
<tr>
<td>c fails during operation</td>
<td>no node claims ownership of ((k_a, k_c))</td>
</tr>
<tr>
<td>communication fault between a and c</td>
<td>routes through a skip over c</td>
</tr>
</tbody>
</table>
Distributed Hash Tables: Weak Atomicity Causes Anomalies

DHTs use ad-hoc protocols to add and remove nodes

what happens if...

- two nodes join at the same place at the same time
- two adjacent nodes leave at the same time
- during a node join the predecessor leaves
- one node mistakenly thinks another node has failed
Scatter: Design Overview

How is Scatter different?
use *groups* as building blocks instead of nodes

What is a group?
set of nodes that cooperatively manage a key-range

What does this give us?
- nodes within a group act as a single entity
- a group is much less likely to fail than an individual node
- distributed transactions for operations involving multiple groups
Scatter:
Group Anatomy

- group replicates all state among members with Paxos
  
  nodes = \{a, b, c\}
  keys = (k_z, k_c]
  values = {...}

- changes to group membership are Paxos reconfigurations:
  - include new nodes
  - exclude failed nodes

- key-range further partitioned among nodes of group for performance

- each node orders client operations on its keys
Scatter: Self-Reorganization

Some problems can’t be handled within a single group:
- Small groups are at risk of failing
- Large groups are slow
- Load imbalance across groups

Multi-group operations:
- **Merge** two small groups into one
- **Split** one large group into two
- Rebalance keys and nodes between groups

Distributed transactions coordinated locally by groups
Example: Group Split

2PC

- a
- b: split?
- c
Example: Group Split

2PC

a

b split?

c

a ok!

b

c ok!
Example: Group Split

2PC

- a
- b
- c (split)

- a
- b
- c (ok)

- a
- b
- c (split)
Example: Group Split

2PC

- a
- b: split?
- c

- a
- b
- c: ok!

- a
- b: split!
- c

- a
- b
- c

Example: Group Split

2PC

- Split (b) -> Split (c) -> Split (b)
- Split (a) -> Split (b)
- Split (b) -> Split (c)

- Split (c) -> Split (b)
- Split (b) -> Split (c)
- Split (a) -> Split (b)

- Split (b) -> Split (c)
- Split (c) -> Split (b)
- Split (a) -> Split (b)
Example: Group Split

2PC

```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>split?</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>ok!</td>
<td></td>
<td>ok!</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>split!</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Example: Group Split

2PC

- **Split**
  - a
  - b
  - c

- **Split**
  - a
  - ok!
  - b
  - c
  - ok!

- **Split**
  - a
  - b
  - c
  - split!

- **RECONFIGURE**
  - b
  - split!
  - ok!

- **RECONFIGURE**
  - b
  - split!
  - ok!

- **COMMITTED**

- **RECONFIGURE**
  - b
  - split!
  - ok!

- **RECONFIGURE**
  - b
  - split!
  - ok!
Example: Group Split

2PC

RECONFIGURE!

committed
Scatter

✓ linearizable consistency semantics
  ...group consensus, transactions
✓ scalable in a wide area network
  ...local operations
✓ high availability
  ...replication, reconfiguration
✓ performance close to existing systems
  ...key partitioning, optimizations
Evaluation:
Overview

Questions:
1. How robust is Scatter in high-churn peer-to-peer environment?
2. How does Scatter adapt to dynamic workload in datacenter environment?

Comparisons:

<table>
<thead>
<tr>
<th>Environment</th>
<th>P2P</th>
<th>Datacenter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparison System</td>
<td>OpenDHT</td>
<td>ZooKeeper</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Comparison: OpenDHT

Layered OpenDHT’s recursive routing on top of Scatter groups

Implemented a Twitter-like application, Chirp

Experimental Setup:
- 840 PlanetLab nodes
- injected node churn at varying rates
- Twitter traces as a workload
- tweets and social network stored in DHT
Comparison: OpenDHT

Consistency

Scatter has zero inconsistencies and high availability even under churn

Availability
Comparison: OpenDHT

**Latency**

- Scatter
- OpenDHT

Scalable consistency is cheap
Comparison: Replicated ZooKeeper

ZooKeeper:
small-scale, centralized coordination service

Replicated ZooKeeper:
statically partitioned global key-space to multiple, isolated ZooKeeper instantiations

Experimental Setup:
- testbed: Emulab
- varied total number of nodes
- no churn
- same Chirp workload
Comparison: Replicated ZooKeeper

Scalability

Dynamic partitioning adapts to changes in workload
Scatter: Summary

- consensus groups of nodes as fault-tolerant building blocks
- distributed transactions across groups to repartition the global key-space
- evaluation against OpenDHT and ZooKeeper shows strict consistency, linear scalability, and high availability