In-Memory Key-Value Store Live Migration with NetMigrate

Zeying Zhu, Yibo Zhao, Alan Zaoxing Liu
In-Memory Key-Value Stores

• Key-value stores are widely used
  • Feature store of machine learning inference
  • In-memory caching
  • Real-time analytics

• Data amount is large
  • Store billions of records
  • Retrieve millions of records under low latency constraints
Live Migration is A Key Technique

• No service downtime during key-value shard migration between nodes.
• Why migrate data?
  • Load balancing
Live Migration is A Key Technique

• No service downtime during key-value shard migration between nodes.
• Why migrate data?
  • Load balancing
  • Spatial locality
  • Horizontal scaling

![Diagram showing load balancing with KV server 1 and KV server 2, clients 1 and 2, and queries to specific servers to illustrate spatial locality and horizontal scaling.]
Live Migration is A Key Technique

• No service downtime during key-value shard migration between nodes.
• Why migrate data?
  • Load balancing
  • Spatial locality
  • Horizontal scaling

• Existing solutions
  • Source-based
  • Destination-based
  • Hybrid
Source-based Migration

READ: served by source
WRITE: served by source

Source KV

Updated

Client 1
READ

Client 2
WRITE

…

Client n

Migrate all data

Destination KV

Updated

Dirty data logs

RAMCloud [TOCS ‘15], Remus [SIGMOD’22]
Source-based Migration

**READ**: served by source

**WRITE**: served by source

- Low query latency during migration because source node already has the queried data
- Extra dirty data transfer from source to destination
- Downtime when terminating migration

RAMCloud [TOCS ‘15], Remus [SIGMOD’22]
Destination-based Migration

**READ:** served by destination  
**WRITE:** served by destination

![Diagram showing migration process](image-url)

- Source KV ➔ Destination KV
- Migrate all data
- PriorityPull not-migrated data
- Not Migrated
Destination-based Migration

**READ:** served by destination
**WRITE:** served by destination

- Quickly shift source node’s pressure, short migration time
- High query latency due to missed data access in the destination (increase 100%~400%)
- Low throughput (drop 66%)
Hybrid Migration

**READ**: served by both source and destination

**WRITE**: served by destination

- **Client 1**
- **Client 2**
- ... **Client n**

- **Bookkeeping migration states**: migrated keys
- **WRITE**
- **READ**
- **Double-READ**
- **Double-READ**

- **Source KV**
- **Destination KV**
- **Migrate all data**

Fulva [SRDS ‘19]
Hybrid Migration

**READ:** served by both source and destination

**WRITE:** served by destination

- Leverage both so performance is better when most of data is in the source.
- Double-read incurs large bandwidth overhead between clients and servers (~50%) because of no fine-grained state tracking.
Design Goals of NetMigrate:
- Minimal query performance impact
- Low extra overhead from migration
- Acceptable and tunable migration time

Tradeoff between query performance and migration time
Existing solutions don’t know where the data is and pay cost of going to wrong places.

Key Idea: Programmable Top-of-Rack switches to track the migration states.
- Centralized view of all data movement
- Real-time information of who owns the data
ToR Switch Controller

Clients

Query

Look up migration state

Source KV

Destination KV

Storage Servers

Migration Instance

Key-Value Storage Rack
A Typical Programmable Switch Architecture

- Flexible programmability
  - Parse, read and update custom fields at line rate
- Registers
  - Store data
- High line-rate packet processing 12.4 Tbps
Design Challenges of NetMigrate

- **Challenge #1**: How to track fine-grained migration states?
  - On-switch resources are limited (e.g., 64MB SRAM vs. Millions of KV pairs)

- **Challenge #2**: How to query during migration?
  - Maintain data consistency during migration.
    - Read-After-Write, Write-After-Read, Write-After-Write.

- **Challenge #3**: How to support dynamic migration policies?
Design Challenges of NetMigrate

• Challenge #1: How to track fine-grained migration states?
  • On-switch resources are limited (e.g., 64MB SRAM vs. Millions of KV pairs)

• Challenge #2: How to query during migration?
  • Maintain data consistency during migration.
    • Read-After-Write, Write-After-Read, Write-After-Write.

• Challenge #3: How to support dynamic migration policies?
Shrink Record Granularity for Limited Switch Resources

On-switch resources are limited (e.g., 64MB SRAM vs. Millions of KV pairs)

KVS data structure: hash table

Track migration in a coarser record granularity
Three States to Understand Data Location

Group migration states: migrated, ongoing-migration, not-migrated
Probabilistic Ownership Tracking

Bloom Filter

Counting Bloom Filter

BF

CBF

ToR Switch

Track migrated groups

Track ongoing-migration groups

Group

Source KV

A
B

Destination KV
Not Started Migration

Bloom Filter

Counting Bloom Filter

ToR Switch

Track ongoing-migration groups

Track migrated groups

BF

0
0

CBF

0
0

Group

A
B

Source KV

Destination KV
Ongoing Migration

Bloom Filter

Counting Bloom Filter

BF


0
0


CBF

1
1


Track ongoing migration groups

Track migrated groups

ToR Switch

Group

Source KV

Destination KV

Counting Bloom Filter

Group

Source KV

Destination KV
Finished Migration

Bloom Filter

Counting Bloom Filter

ToR Switch

Track ongoing-migration groups

Track migrated groups

Source KV

Destination KV

Bloom Filter

BF

1

1

CBF

0

0

Group

B

A
Design Challenges of NetMigrate

• Challenge #1: How to track fine-grained migration states?
  • On-switch resources are limited (e.g., 64MB SRAM vs. Millions of KV pairs)

• Challenge #2: How to query during migration?
  • Maintain data consistency during migration.
    • Read-After-Write, Write-After-Read, Write-After-Write.

• Challenge #3: How to support dynamic migration policies?
Data is Consistent When Not Started Migration

Source KV

Clients

BF 0 0

CBF 0 0

WRITE

READ

Migrate all data

Destination KV
Data is Consistent When Finished Migration

Source KV → Migrate all data → Destination KV

Clients

<table>
<thead>
<tr>
<th>BF</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBF</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

READ  WRITE
An Inconsistent Example When Ongoing Migration

Read-After-Write

<table>
<thead>
<tr>
<th>BF</th>
<th>CBF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Migrate all data

Source KV

Destination KV

Updated
An Inconsistent Example When Ongoing Migration

Read-After-Write

- Not sure where the key is located because of tracking at group level.
- Not sure whether there was a WRITE due to no tracking on every WRITE.
Data is Consistent When Ongoing Migration

Source KV

Destination KV

Migrate all data

Double-READ

BF

CBF

0 0

1 1

WRITE

Clients
Data is Consistent When Ongoing Migration

• Data consistency is maintained by Double-READ.
• Overhead caused by Double-READ is negligible.
Data is Consistent even with False Positives

Updated due to hash collision

Migrate all data

PriorityPull

(see more details in our paper)
Data is Consistent even with False Positives

Updated due to hash collision

BF

Clients

CBF

READ

WRITE

Migrate all data

PriorityPull

Not Migrated

Overhead from false positives is negligible

(check more details in our paper)
Leveraging probabilistic data structures on the switch to track three migration states.
Query protocol guaranteeing consistency.
The overhead caused by false positives and unsure states is small.
Evaluation

• **Testbed**
  • 6.5 Tbps Intel Tofino switch
  • 3 servers each with an 8-core CPU, a 40G NIC, and 64GB memory

• **Baselines**
  • Source-based migration protocol, Rocksteady, Fulva

• **Workloads**
  • Migrating 256 million KV pairs (~16GB), with 4B key, 64B value
  • YCSB with 0%, 5%, 10%, 20%, 30% write ratio
  • Source CPU budgets: **100%**, 70%, 40%
Overall performance -- Throughput

Setting: YCSB-B (5%) write ratio, source node is not overloaded (100%)

32% to 78% average throughput improvement compared to Source-based, Rocksteady, Fulva with similar migration time.
Overall performance – Median Latency

Setting: YCSB-B (5%) write ratio, source node is not overloaded (100%)

49% to 65% average median latency reduction.
Up to 39% average 99% tail-latency reduction.
## Network Overhead

<table>
<thead>
<tr>
<th>Protocols/Overhead</th>
<th>Client-side</th>
<th>Server-side</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rocksteady</td>
<td>7%~12%</td>
<td>0</td>
</tr>
<tr>
<td>Source-based</td>
<td>0</td>
<td>Proportional to write ratio</td>
</tr>
<tr>
<td>Fulva</td>
<td>~50%</td>
<td>0</td>
</tr>
<tr>
<td>NetMigrate</td>
<td>&lt;0.05%</td>
<td>&lt;5×10⁻⁵%</td>
</tr>
</tbody>
</table>

Extra network bandwidth overhead between clients and servers (client-side) or between servers (server-side)
Conclusions

• Existing KV store live migration techniques still suffer from low query-serving performance and high overhead.

• We propose NetMigrate, a network-based hybrid live migration approach.
  • Track fine-grained migration states in programmable data plane.
  • Provide enhanced throughput and low migration overheads.