Erasure Coding for Small Objects in In-Memory Key-Value Storage

Matt M. T. Yiu, Helen H. W. Chan, Patrick P. C. Lee
The Chinese University of Hong Kong

SYSTOR 2017
Introduction

- In-memory key-value (KV) stores are widely deployed for scalable, low-latency access
  - Examples: Memcached, Redis, VoltDB, RAMCloud

- Failures are prevalent in distributed storage systems
  - Replication in DRAM?
    - High storage overheads
  - Replication in secondary storage (e.g., HDDs)?
    - High latency to replicas (especially for random I/Os)
  - Erasure coding
    - Minimum data redundancy
    - Redundant information is stored entirely in memory for low-latency accesses → fast recovery under stragglers and failures
Erasure Coding

- Divide data to \( k \) data chunks
- Encode data chunks to additional \( n-k \) parity chunks
  - Each collection of \( n \) data/parity chunks is called a stripe
- Distribute each stripe to \( n \) different nodes
  - Many stripes are stored in large-scale systems

- **Fault tolerance**: any \( k \) out of \( n \) nodes can recover file data
- **Redundancy**: \( \frac{n}{k} \)
Challenges

- Erasure coding is expensive in data updates and failure recovery
  - Many solutions in the literature

- Real-life in-memory storage workloads are dominated by small-size objects
  - Keys and values can be as small as few bytes (e.g., 2-3 bytes of values) [Atikoglu, Sigmetrics’12]
  - Erasure coding is often used for large objects

- In-memory KV stores issue decentralized requests without centralized metadata lookup
  - Need to maintain data consistency when failures happen
Our Contributions

- Build MemEC, a high-availability, erasure-coding-based in-memory KV store that aims for:
  - Low-latency access
  - Fast recovery (under stragglers/failures)
  - Storage-efficient

- Propose a new all-encoding data model

- Ensure graceful transitions between normal mode and degraded mode

- Evaluate MemEC prototype with YCSB workloads
Existing Data Models

- **All-replication**
  - Store multiple replicas for each object in memory
  - Used by many KV stores (e.g., Redis)
Existing Data Models

- **Hybrid-encoding**
  - Assumption: Value size is sufficiently large
  - Erasure coding to values only
  - Replication for key, metadata, and reference to the object
  - Used by LH*RS [TODS‘05], Cocytus [FAST‘16]
Our data model: All-encoding

- Apply erasure coding to objects in entirety
- Design specific index structures to limit storage
All-encoding: Data Organization

- Divide storage into fixed-size chunks (4 KB) as units of erasure coding
- A unique fixed-size chunk ID (8 bytes) for chunk identification in a server
All-encoding: Data Organization

- Each data chunk contains multiple objects
- Each object starts with fixed-size metadata, followed by variable-size key and value
Append new objects to a data chunk until the chunk size limit is reached, and \textit{seal} the data chunk.

Sealed data chunks are encoded to form parity chunks belonging to the same stripe.
**All-encoding: Data Organization**

- Chunk index maps a chunk ID to a chunk reference
- Object index maps a key to an object reference
- Use *cuckoo* hashing
- No need to keep redundancy for both indexes in memory

- Key-to-chunk mappings are needed for failure recovery, but can be stored in secondary storage
All-encoding: Chunk ID

- Chunk ID has three fields:
  - **Stripe list ID**: identifying the set of \( n \) data and parity servers for the stripe
    - Determined by hashing a key
  - **Stripe ID**: identifying the stripe
    - Each server increments a local counter when a data chunk is sealed
  - **Chunk position**: from 0 to \( n - 1 \)

- Chunks of the same stripe have the same stripe list ID and same stripe ID.
Analysis

- All-encoding achieves much lower redundancy

(a) $K = 8$, $(n, k) = (10, 8)$

(b) $K = 32$, $(n, k) = (10, 8)$
MemEC Architecture

Only in degraded mode

Client ➔ Object ➔ Coordinator ➔ Server

Proxy ➔ Unified memory

Object

SET / GET / UPDATE / DELETE

Server

15
Fault Tolerance

- In normal mode, requests are **decentralized**
  - Coordinator is not on I/O path

- When a server fails, proxies move from decentralized requests to **degraded requests** managed by coordinator
  - Ensure data consistency by reverting any inconsistent changes or replaying incomplete requests
  - Requests that do not involve the failed server remain decentralized

- **Rationale**: normal mode is common case; coordinator is only involved in degraded mode
Server States

- Coordinator maintains a state for each server and instructs all proxies how to communicate with a server.
Server States

- All proxies and working servers share the same view of server states

- Two-phase protocol:
  - When coordinator detects a server failure, it notifies all proxies to finish all decentralized requests (intermediate state)
  - Each proxy notifies coordinator when finished
  - Coordinator notifies all proxies to issues degraded requests via coordinator (degraded state)

- Implemented via atomic broadcast
Evaluation

- Testbed under commodity settings:
  - 16 servers
  - 4 proxies
  - 1 coordinator
  - 1 Gbps Ethernet

- YCSB benchmarking (4 instances, 64 threads each)
  - Key size: 24 bytes
  - Value size: 8 bytes and 32 bytes (large values also considered)
  - Do not consider range queries
Impact of Transient Failures

Failures occur before load phase:
- Latency of SET in load phase increases by 11.5% with degraded request handing
- For Workload A, latencies of UPDATE and GET increase by 53.3% and 38.2%, resp.
Impact of Transient Failures

Failures occur after load phase:

- Latencies of GET and UPDATE increase by 180.3% and 177.5%, resp.
- Latency of GET in Workload C only increase by 6.69%
State Transition Overhead

Average elapsed times of state transitions with 95% confidence

<table>
<thead>
<tr>
<th>State transition</th>
<th>Elapsed time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Single failure</td>
</tr>
<tr>
<td>$T_{N \rightarrow D}$</td>
<td></td>
</tr>
<tr>
<td>With req.</td>
<td>4.77 ± 0.79</td>
</tr>
<tr>
<td>No req.</td>
<td>1.74 ± 0.09</td>
</tr>
<tr>
<td>$T_{D \rightarrow N}$</td>
<td></td>
</tr>
<tr>
<td>With req.</td>
<td>628.5 ± 43.9</td>
</tr>
<tr>
<td>No req.</td>
<td>0.91 ± 0.46</td>
</tr>
</tbody>
</table>

Difference between two elapsed times is mainly caused by reverting parity updates of incomplete requests.

Elapsed time includes data migration from the redirected server to the restored server, so increases a lot.
Conclusion

- A case of applying erasure coding to build a high-available in-memory KV store: MemEC
  - Enable fast recovery by keeping redundancy entirely in memory

- Two key designs:
  - Support of small objects
  - Graceful transition between decentralized requests in normal mode and coordinated degraded requests in degraded mode

- Prototype and experiments

- Source code: https://github.com/mtyiu/memec