Graham Hoyes, Danial Hasan, Daniil Orekhov

ECE419 M4 Design Document

Proposed Feature: Key Expiry

A common application of key-value stores is as a cache, such as Redis or Memcached. Since caches are generally used to store data which is persisted elsewhere (such as in a database), a common feature of key-value caches is the ability to set an expiry time for keys. In a persistent key-value store context such as this project, key expiry is also useful for applications such as storing authentication tokens, or expiring items in one's ecommerce carts.

To enable this use case, we have extended our key-value server on top of the replication functionality from Milestone 3 to allow for timed expiry of keys on controllers and replicators.

Key Expiry Specification

Client Site

Clients should be able to insert or update key-value pairs with Time-to-Live (TTL) values through the KVClient command line using the `putttl` command: `putttl <ttl seconds> <key> <value>`. Beyond the extra ttl parameter, the use of this command is identical to the original `put` command. Users can insert or update keys, including changing the TTL value. Key deletion is still handled through the `put <key> null` syntax.

The KVClient library behind the CLI similarly supports a new `putTTL` method, with the same parameters as above. The original `put` method now simply calls `putTTL`, with a null TTL value.

Server Side

Our key-value server provides *guaranteed expiry* on controllers and either *eventual expiry* or *approximate expiry* on replicators.

- **Expiry:** an expired key will return a `GET_ERROR` message and will have their key-value pair deleted from the storage eventually.
- **Guaranteed expiry:** ensures that keys that have surpassed their TTL will be expired.
- **Eventual expiry:** ensures that keys on replicators will eventually be expired by the key's controller server after the key has surpassed its TTL.
- **Approximate expiry:** ensures that keys on replicators will be expired if the key has surpassed its TTL relative to the replicator clock.

Controllers

Keys are expired on the **controller** in two ways: *actively* and *passively*:

- **Active key expiry** occurs when a client issues a `GET` request for a key that has surpassed its TTL but has not yet been deleted. The server will retrieve the key, check the expiry timestamp and expire (and delete) the key.

    - Active key expiry ensures that keys are guaranteed to be expired after their TTL from the client perspective.
- **Passive key expiry** uses a passive watcher thread on the controller that expires a random portion of keys in the storage periodically.
    - This acts to ensure that keys that are placed onto the server with a TTL but not accessed again are eventually deleted from storage to clear up space.

Replicators

Keys are expired on **replicators** in two ways: *active (optional)* and *passive*:

- **Active key expiry** on replicators is an *optional* setting that acts akin to active expiry on controllers; keys are expired when clients issue `GET` requests for a key that has surpassed its TTL but has not been released.
    - To set this option, the `set replicatorsExpireKeys on` command is issued to the ECS.
    - Active key expiry provides an *approximate expiry* guarantee on replicators as it relies on the replicator's local clock to determine when the TTL for a key has elapsed and the key should be expired.
    - This feature ensures that users are not able to access keys on the replicator within the margin of clock drift between replicators and controllers. This may be desirable if passive deletion via the controller is too slow for an application.
- **Passive key expiry** on replicators always occurs and is triggered by the controller when either a) the controller's key expiry watcher expires keys or b) when the controller actively deletes the key on a `GET` request. In either case, the controller will update replicators to delete expired keys.
    - Passive key expiry provides an eventual expiry guarantee on replicators.
    - Expired keys may still be accessed on the replicators if the key has not been actively or passively expired on the controller or if the change has not yet been processed on replicators.

Key Expiry Implementation

Controllers

Adding Key-Values

Our persistent storage implementation from Milestone 1 stores data as JSON objects with `key` and `value` keys, one per line in the storage file. To support expiry, a new `expiryTime` field has been added. This stores the unix timestamp when the key expires.

On receiving a put request, the `ttl` field of the `KVMessage` either contains the TTL in seconds, or null. If the field is not null, the KVServer adds this TTL to the current unix timestamp, and sets the `expiryTime` field of the stored object. If ttl is null, then `expiryTime` is set to null as well.

Active Expiry

When clients issue `GET` requests to the controller, the controller will locate the key-value pair as previously implemented, but now the controller will store the request time and compare it to the `expiryTime` attribute of the key-value pair. If the request time exceeds the expiry time, the key-value pair is deleted from the storage and a `KeyNotFound` exception is issued to the client. Additionally, a delete command is asynchronously issued to replicators to delete the expired key-value pair and ensure eventual expiry on replicators.

Passive Expiry

To implement passive expiry, a `ScheduledExecutorService` is used to run a passive expiry watcher thread at a set interval of 30 seconds. This thread will check the server's storage file for expired keys, remove the expired key-value pairs from storage, and store the expired keys into the write log to send to replicators. If any keys have expired, then the controller will send the write log to replicators to inform them of keys that have expired.

Replicators

Adding Key-Values

Key-value pairs are replicated in the same manner as in M3; i.e. controllers will update replicators with `PUT`, `UPDATE`, and `DELETE` requests.

Active Expiry (Optional)

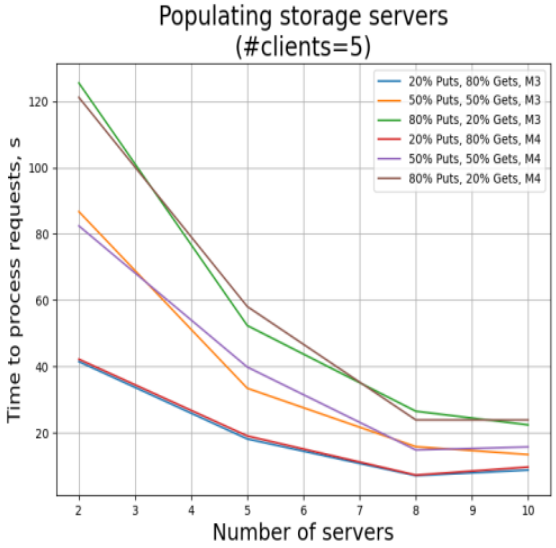
When clients issue `GET` requests to replicators and the user has enabled `replicatorsExpireKeys`, the replicators will locate the key-value pair and expire it in the same manner as controllers. This feature will compare the expiry time of the key to the local time of the replicator, and thus may be inaccurate due to clock drift. Active key expiry from a replicator will not propagate key deletion to other replicators, which must either wait for a get request for that key or for passive expiry.

Passive Expiry

When keys are deleted on the controller, either actively (on a `GET` request) or passively (using the expiry watcher thread) replicators are notified and subsequently delete expired keys. The deletion of these keys utilizes the replication mechanisms implemented in M3 to delete keys on replicators.

Graham Hoyes, Danial Hasan, Daniil Orekhov

Performance Evaluation



As with milestones 2 and 3, we first populate the servers with 500 entries to saturate the data storage, then perform 500 total GET and PUT operations on the service, varying the proportions of each. GET and PUT operations are interleaved, so as time progresses the database continues to saturate. No deletion is performed, so these numbers represent a worst case scenario for operations which are O(n).

When compared to performance of M3, we see that there is very little difference. Generally, we can say that the M4 takes slightly more time to process the request if compared to M3. This makes sense, since M4 is built upon M3 and it adds extra functionality. The new feature, key expiry, requires the servers to acquire locks every 30 seconds to check and delete keys during passive expiry, which slows the performance of the system slightly. Additionally, during active expiry every GET request now has a check for the corresponding key, which also makes the process slightly slower. However, these decreases in performance are marginal, since there are lots of requests going through the servers.

Appendix

Appendix A: Unit Tests

Unit tests were written in JUnit 4 style (the provided test cases were modified to conform to this when required).

Provided tests (unmodified, other than formatting):

- Connection tests
    - Valid connection
    - Unknown host
    - Invalid port
- KVStore Client Interaction tests
    - Successful PUT insert
    - Successful PUT update
    - PUT after a disconnect
    - Successful delete
    - Successful GET
    - GET with unset key

Additional tests (60 total, new M4 tests at the bottom):

- JSON Message tests: Test the initialization, serialization, and deserialization of JSON messages
    - Initialization with only a status type
    - Deserialization from valid json
    - Throws DeserializationException for deserialization from invalid json
    - Initialization from a json string
    - Sets missing optional fields to null
    - Raises DeserializationException for missing status type
- Communication protocol tests: Test the byte communication protocol used by the TCP connections (JSON serialization tested separately, above)
    - Test message sending: Test that protocol version, message length, header unused space, and message payload bytestream are properly constructed
    - Test message receiving: Test that a message can be properly received from a valid communication bytestream
    - Test that IOException is thrown when input connection is closed
    - Test that IOException is thrown when message payload is shorter than indicated in header tag
- KVStore tests: Testing the persistent storage class for permanent storage of key-values
    - Test invalid keys for put/get/delete: Test that the storage throws an exception for keys that are invalid i.e. the key-value pair does not exist
    - Test put/get: Test that a put request for a new key-value pair is correctly stored and that a subsequent get request returns the correct value
    - Test put update: Test that a put request correctly updates an existing key-value pair
    - Test delete: Test that key-value pairs are deleted as intended

    - Test storage file deletion: Test that a correct exception is thrown if the storage file is moved/deleted
    - Test invalid data format: Test that an exception is thrown for invalid data formatting within the key-value persistent storage file
- Hash Ring tests: Testing actions on the global hash ring metadata object
    - Test single node hash ring: ensure that a single node is responsible for the entire hash ring
    - Test update hash ring: start hash ring with one node, ensure that adding another node splits the ring accordingly
    - Test remove from hash ring: start hash rings with multiple nodes, remove a node and ensure its successor adopts the removed node's range
    - Test hashing: ensure that nodes are responsible for keys in their range
    - Test that the hash ring is aware of which nodes replicate with other nodes
    - Test that server nodes can determine equality between each other
    - Test that the hash ring can be serialized
    - Test that the hash ring can be deserialized, and properly rebuilds the linked list
    - Test getting replicators of a node
    - Test getting controllers of a node
- ECS Tests: Test the ECS functionality of managing the hash ring, starting/stopping nodes, adding/removing nodes
    - Test single server startup: launch, setup, and start a single server and ensure that a client can connect to it
    - Test ECS stop server: ensure that ECS can stop a launched server, and that the server responds with a SERVER_STOPPED message to clients
    - Test metadata updates: ensure the ECS updates metadata for all servers when adding a new node
    - Test add node write lock release: test that newly added nodes to an already instantiated ECS are writable to after they have been started
    - Test moving of data: test that data in the respective hash range is moved to new nodes
    - Test data is moved after removing a nodes: ensure that data is moved to successor node in hash ring after a node is removed
    - Test adding multiple nodes: ensure that multiple nodes can be added and that data is moved accordingly
    - Test removal of multiple nodes: ensure that multiple nodes are removed and their data is moved to correct successors
    - Test replicator get: Test that get requests can be served by a replicator
    - Test replicator put: Test that a replicator eventually adds a new key-value pair put on its controller
    - Test replicator put update: Test that a replicator eventually updates a key-value pair that was updated on its controller
    - Test replicator delete: Test that a replicator eventually deleted a key-value pair that was deleted on its controller.
    - Test move relevant data: Test that when moving data, only the data the new server is responsible for is sent
- Key Expiry Tests: Testing key expiry behaviour
    - **(new)** Test that putttl does not prematurely expire keys

- **(new)** Test expired keys deleted actively by coordinator
- **(new)** Test expired keys not deleted actively by replicator by default
- **(new)** Test that the coordinator propagates expired key deletion to replicators after a get
- **(new)** Test enabling replicators expire keys
- **(new)** Test disabling replicators expire keys
- **(new)** Test that keys are eventually removed by the watcher on controller
- **(new)** Test that keys are eventually removed by the watcher on replicators
- **(new)** Test replacing a TTL key with no TTL
- **(new)** Test replacing a no TTL key with TTL