Graham Hoyes, Danial Hasan, Daniil Orekhov

ECE419 M2 Design Document

External Configuration Service

The ECS handles scaling of the distributed storage service by adding and removing nodes, transferring data between them, and acting as a centralized source for storage service metadata. The ECS maintains two state elements, a node pool and a hash ring, both of which are wrappers for an object representing a server node. The ECS communicates with KVServers via ZooKeeper.

The KVServer nodes, via the `ECSNode` class, store the name, port, hostname, and predecessor hash of a node in the hash ring. This class computes node hashes, and has methods for determining whether the node is responsible for a given key. The ECS has a queue of offline `ECSNodes`, from which it pulls nodes to launch with calls to `addNodes()`. When nodes are removed, they are placed at the back of the offline nodes queue.

A `HashRing` object stores the global metadata of the storage service, consisting of a list of `ECSNode`s. The nodes are stored in order of increasing name hash, making finding the predecessor and successor nodes given a node index simple. Nodes also store the hash of their predecessor directly, for quick key membership checks. Looking up a node in the hash ring by node name or hash requires an O(n) search through all nodes. Adding or removing a node involves inserting the node into the appropriate position of the list, as well as updating the predecessor hash for the successor node.

Each KVServer and client library stores a copy of the `HashRing` as well. Determining the server responsible for a given key also involves a linear search through all nodes in the hash ring, calling `node.isNodeResponsible(key)`. A future area for improvement is to use a tree structure for sub-linear node lookups.

ZooKeeper

ZooKeeper has two main roles in the storage service: to transmit admin messages and responses between the KVServers and the ECS, and to store a heartbeat for online KVServers. Each online node has 3 ZNodes: `/servers/<serverID>`, `/servers/<serverID>/admin`, and `/heartbeats/<serverID>`. When a KVServer is first added, it creates an ephemeral heartbeat ZNode. After launching the KVServer (over SSH or as a local process), the ECS blocks and waits for a watcher on the heartbeat node to be triggered, indicating that the node was brought online. After sending the shutdown command to a node, the ECS again waits for a watcher on the heartbeat to be triggered by its deletion, indicating successful shutdown.

Messages are passed by the `/servers/<serverID>` and `/servers/<serverID>/admin` ZNodes. Admin messages, represented by the `AdminMessage` class, are json-serialized strings which contain an action, a message (in the case of errors), the `HashRing` metadata object, and sender and receiver `ECSNode`s (for data transfer). All of the included fields are themselves serializable and deserializable, via the GSON package.

On initialization, KVServers create a watcher on their respective admin ZNodes. This watcher is responsible for handling the instructions given by the ECS (start, stop, transfer data, receive data, etc). After the message processing is complete, the KVServer writes a response `AdminMessage` to its server ZNode with an ACK or error status. After writing the response, it re-registers the watcher for the next admin message. As a result, KVServers cannot handle simultaneous admin messages.

Prior to sending an admin message, the ECS sets up a watcher on the KVServer's server ZNode for the response. Setting the watcher for the response before sending the message assures that the response is not missed in the time it would otherwise take to set the watcher after sending the message. After setting the watcher, the ECS writes an admin message to the KVServer's admin ZNode. It then blocks for a `CountDownLatch` on the response watcher, thus making otherwise asynchronous responses synchronous.

Data Transfer

When adding or removing nodes from the storage service, data transfer must occur to hand off key-value pairs to the new or successor node in the hash ring. When data is to be transferred between servers, the ECS first sets the write lock on the sending node and confirms the setting of the lock before proceeding. Then an admin message of type `RECEIVE_DATA` is sent to the receiving server; which is the newly added server when creating a new node, or the successor of the node being removed, when removing a node. Upon receiving a `RECEIVE_DATA` message, the receiving server finds an open port, sets up a `DataReceiver` class that will run on a new thread, and acknowledges back to the ECS with the port waiting to receive data.

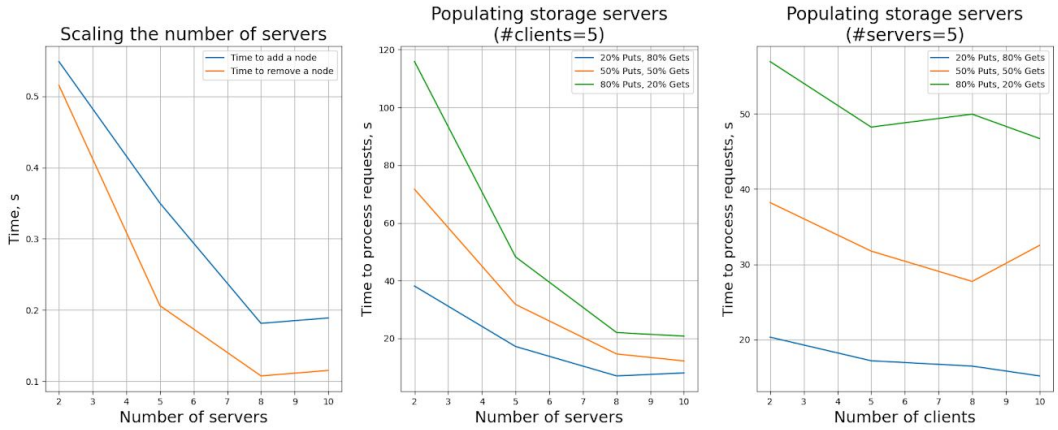
After receiving acknowledgement from the receiving server, the ECS will send a `MOVE_DATA` admin message, which contains the new hash ring metadata for the sending node, the `ECSNode` object for the receiving server and the port the receiving node is listening on. The successor receives the `MOVE_DATA` message and proceeds to copy its currently stored key-value pairs into two new files, a `send_file` for the key-values that will be sent to the new server, and a `keep_file` for the key-values that will remain on the sending server as per its new hash range. This preserves the original key-value storage of the sending server until after the data transfer is complete. The sending node then connects to the receiving node at the designated ip address and port, and uses a `BufferedOutputStream` to send the key-value data to the receiving node. Once the sender node has successfully sent all the key-value data, it sends back an acknowledgement to the ECS to confirm completion. The ECS will then release the write lock on the sender, and notify the sender node to remove the old data and replace with the `keep_file` data.

Client Library

The client library (`client`) has been updated to store a copy of the `HashRing` object, as well as the current `ECSNode` (aka KVServer, node within the hash ring) it is connected to. For each key, the client determines the correct server to connect to. If it is different from the current one, it disconnects and connects to the correct server. On receiving a `SERVER_NOT_RESPONSIBLE` message, the server updates its hash ring and silently retries. If the KVServer is write locked or in the stopped state, the user is informed to try again later, rather than blocking until success.

Our clients are resilient against the KVServer they are connected to being shut down by the ECS. When disconnected, the client will try successive servers in the hash ring until a successful connection is made. Even if the next operation is a success, clients are required to update their hash ring to prevent re-connecting to the disconnected server. We accomplish this simply by sending metadata with every response from the KVServer. To save on network bandwidth, this should be broken out into a separate `GET_METADATA` client message, used on top of metadata updates with `SERVER_NOT_RESPONSIBLE`.

Performance Testing



To evaluate performance, we first populate the servers with 500 entries to saturate the data storage, and, then perform 500 total GET and PUT operations on the service, varying the proportion of each. The GET operations are performed on the first 500 saturation entries, ensuring that the data is available on the servers. GET and PUT requests are interleaved with each other throughout the performance evaluation, so as time progresses the database continues to saturate. No deletion is performed, so these numbers represent a worst case scenario for PUT requests which take O(n) time to place new values. This process is repeated for different numbers of servers and clients used. Once all the entries have been saved in the storage, we also add a node and remove a node to test the system scaling up and down.

As the total number of servers increases, the time needed to add or remove a node decreases, as the amount of data stored per server decreases and there is less data to move around. Similarly, as the number of servers increases, it takes less time to populate them, since more data can be stored in parallel. To test the impact of increasing the number of clients, we fix the number of servers to 5 and measure the time to process all requests as the number of clients increases. We note that the performance improves as the number of clients increases, which indicates that by having more clients perform requests in parallel we better distribute the load across all servers and better saturate server usage.

Graham Hoyes, Danial Hasan, Daniil Orekhov

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 (38 total, new M2 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
- **(new)** 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 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 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