Fault tolerant Key Value Store

Problem Description

I have built fault-tolerant key value store with following functionalities

CRUD (Create, Read, Update, Delete)

- Key-Value store supporting Create, Update, Delete and Read operations

Load balancing

- I have used consistent hashing ring to hash both servers and keys

Fault tolerant

- The application can handle up to 2 failures. It is handled by replicating each key 3 times to 3 successive nodes in the ring, starting from the first node at or to the clockwise of the hashed key

Consistency

- The application uses Quorum consistency level for both reads and writes (at least 2 replicas)

Stabilization

- After the failure (nodes join/leave/fail) the application tries to stabilize itself by recreating 3 replicas

Working

The application is implemented by three layer framework that will allow to run multiple copies of peers within one process running a single threaded simulation engine. The three layers are 1) the lower EmulNet (network), 2) middle layer including: Gossip Membership Protocol for failure detection and the Key-Value store and 3) Application Layer.

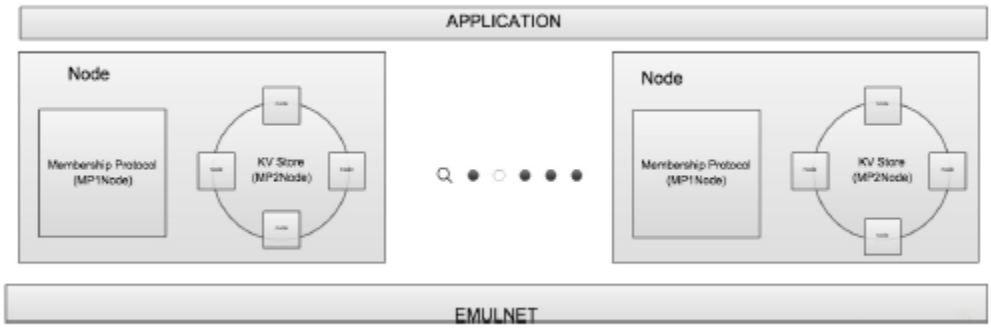

Figure 1. The three layers

The Key- Value store implementation will sit above EmulNet in a peer to peer (P2P) layer, but below an App layer as shown in Figure 1. It is like a 3 layer protocol stack with App, P2P, and EmulNet as the three layers (from top to bottom).

Each node in the P2P layer is logically divided in two components: Membership protocol and Key-Value store. At each node, the Key-Value store talks to the membership protocol and receives from it the membership list. It then uses this to maintain its view of the virtual ring. Periodically, each node engages in the membership protocol to try to bring its membership list up to date. The application provide both client side and server side API for CRUD operations. The application layer chooses a non-faulty node randomly as the client. The same node can be considered as the coordinator.

Implementation

Following are the important classes

HashTable.cpp

Purpose → This class provides following functionalities. It wraps C++11 std::map. It supports keys and values which are std::string.
Methods →

- bool HashTable::create(string key, string value) → This function inserts they (key,value) pair into the local hash table and returns true on SUCCESS and false on FAILURE
- string HashTable::read(string key) → This function searches for the key in the hash table. It returns string value if found else returns NULL
- bool HashTable::update(string key, string newValue) → This function updates the given key with the updated value passed as an argument. It the key is found returns true on SUCCESS and false on FAILURE
- bool HashTable::deleteKey(string key) → This function deletes the given key and the corresponding value if the key is found. It returns true on SUCCESS and false on FAILURE
- bool HashTable::isEmpty() → This function return true if the hash table is empty else return false
- unsigned long HashTable::currentSize() → This function return current size of hash table. It counts the number of element in the hashtable

Message.cpp

Purpose → This class provides constructor for different message type. It helps to pass messages among nodes.
Constructors →

- Message (string message) → Constructs message from a string
- Message (const Message& anotherMessage) → Constructs message from another message
- Message (int _transID, Address _fromAddr, MessageType _type, string _key, string _value) → Constructs Create message
- Message (int _transID, Address _fromAddr, MessageType _type, string _key, string _value, ReplicaType _replica) → Constructs Update message

- Message (int _transID, Address _fromAddr, MessageType _type, string _key) → Constructs read or delete message
- Message (int _transID, Address _fromAddr, MessageType _type, bool _success) → Constructs reply message
- Message (int _transID, Address _fromAddr, string _value) → Constructs read reply message

Entry.cpp

Purpose → This class can be used to store the value in the key value store. This class describes the entry for each key in the Distributed Hash Table. The timestamp field is appended along with value for read repair purpose (ensures always latest written value is read)

Constructors →
- Entry (string entry) → Convert string to get an Entry object
- Entry (string _value, int _timestamp, ReplicaType _replica) → Construct an Entry Object

Node.cpp

Purpose → This class wraps each node's Address and the hash code obtained by consistently hashing the Address. It also provides the methods to compute/set/get hash code and address

Methods →
void Node::computeHashCode() → Computes the hash code of the node address
size_t Node::getHashCode() → Return hash code of the node
Address * Node::getAddress() → Return the address of the node
void Node::setHashCode(size_t hashCode) → Set the hash code of the node
void Node::setAddress(Address address) → Set the address of the node

Params.cpp

Purpose → This class is used to set the parameters required for different test cases. It reads and sets the parameters defined in different files under testcases directory

Files under **testcases** directory →
- create.conf
- delete.conf
- read.conf
- update.conf

Methods →
- void Params::setparams(char *config_file) → Set the parameters for each different testcases

EmulNet.cpp

Purpose → This class provides the functions which are used to send and receive the message

Methods →

- int EmulNet::ENsend(Address *myaddr, Address *toaddr, char *data, int size) → Send the message
- int EmulNet::ENrecv(Address *myaddr, int (* enq)(void *, char *, int), struct timeval *t, int times, void *queue) → Receive the message
- int EmulNet::ENcleanup() → Cleanup the EmulNet. Called exactly once at the end of the program

Member.cpp

Purpose → This class defines the member structure. It includes the membership list along with queue for each individual member. It also has address class and membership list class. Address class stores the address and port number for each node. MemberListEntry class will hold the membership list for each node. The membership list is useful for gossip protocol. The gossip protocol is used to detect failure.

MP1Node.cpp

Purpose → This class provides the implementation of Gossip protocol. The Gossip protocol is used to detect failure.

Methods →

- int MP1Node::recvLoop() → Receives message from the network and pushes into the queue
- void MP1Node::nodeStart(char *servaddrstr, short servport) → Bootstraps the node. Initialize the node address and introduce self to group
- int MP1Node::introduceSelfToGroup(Address *joinaddr) → Try to join the group by sending JOINREQ message to introducer
- void MP1Node::nodeLoop() → Check messages in queue and perform membership protocol duties
- void MP1Node::checkMessages() → Check messages in the queue and call the respective message handler
- bool MP1Node::recvCallBack(void *env, char *data, int size) → Message handler for different message types
- void MP1Node::processJoinReq(void *env, char *data, int size) → Run by the coordinator to process JOIN requests
- void MP1Node::processJoinRep(void *env, char *data, int size) → Message handler for JOINREP. The list will be present in the message. It needs to add the list to its own table. Data parameter contains the entire table
- char* MP1Node::serialize(Member *node) → Serialize the membership list
- char* MP1Node::deserializeAndUpdateTable(const char *msg) → Deserialize the message and update the membership list based on heartbeat received

MP2Node.cpp

Purpose → This class encapsulates all the key-value store functionality including:

- Ring implementation including initial setup and updates based on the membership list obtained from Gossip protocol (**MP1Node**)
- Provide interfaces to the key value store
- Stabilizing the key value store whenever there is a change in membership
- Client side CRUD APIs
- Server side CRUD APIs

Methods →

- void MP2Node::updateRing() →
  - Gets the current membership list from the Membership Protocol (MP1Node). The membership list is returned as a vector of Nodes
  - Constructs the ring based on the membership list
  - Calls the Stabilization Protocol
- vector<Node> MP2Node::getMembershipList() →
  - This function goes through the membership list from the Membership protocol/MP1 and
    - Generates the hash code for each member
    - Populates the ring member in MP2Node class
  - It returns a vector of Nodes. Each element in the vector contain the following fields:
    - Address of the node
    - Hash code obtained by consistent hashing of the Address
- size_t MP2Node::hashFunction(string key) →
  - This functions hashes the key and returns the position on the ring. HASH function used for consistent hashing
- **Client side API →**
  - void MP2Node::clientCreate(string key, string value)
  - void MP2Node::clientRead(string key)
  - void MP2Node::clientUpdate(string key, string value)
  - void MP2Node::clientDelete(string key)
- All the above mentioned Client side API perform following functions
  - Constructs the message
  - Finds the replicas of this key
  - Sends a message to the replica

- **Server side API →**
  - bool MP2Node::createKeyValue(string key, string value, ReplicaType replica, int transID, Address selfAddr)
    - Inserts key value into the local hash table
    - Return true or false based on success or failure
  - string MP2Node::readKey(string key, int transID, Address selfAddr)
    - Read key from local hash table
    - Return value

    - bool MP2Node::updateKeyValue(string key, string value, ReplicaType replica, int transID, Address selfAddr)
        - Update the key to the new value in the local hash table
        - Return true or false based on success or failure
    - bool MP2Node::deletekey(string key, int transID, Address selfAddr)
        - Delete the key from the local hash table
        - Return true or false based on success or failure
- void MP2Node::checkMessages() →
    - Pops messages from the queue
    - Handles the messages according to message types
- vector<Node> MP2Node::findNodes(string key) →
    - Find the replicas of the given key function. This function is responsible for finding the replicas of a key
- int MP2Node::generateTransId(vector<int> transID) →
    - Gets the latest transId from vector of Transaction ID of current Node. Increment the count by 1 and return the new transID
- void MP2Node::stabilizationProtocol(vector<Node> currMemberList) →
    - Runs the stabilization protocol in case of Node joins and leaves. It ensures that there always 3 copies of all keys in the Distributed Hash Table at all times.
    - Ensures that there are three "CORRECT" replicas of all the keys in spite of failures and joins
      (Note: - "CORRECT" replicas implies that every key is replicated in its two neighboring nodes in the ring)

Application.cpp

Purpose → This class provide the application layer functionalities.

Methods →

- int Application::run() → Main driver function of the Application layer
- void Application::mp1Run() → Performs all the membership protocol functionalities
- void Application::mp2Run() → Performs all the key value store related functionalities including
  1) Ring operations
  2) CRUD operations
- void Application::insertTestKVPairs() → Inserts test KV pairs into the system
- void Application::deleteTest() → Test the delete API of the KV store
- void Application::readTest() → Test the read API of the KV store
- void Application::updateTest() → Tests the update API of the KV Store

Logging

I have created common logging for the application. Following are the log function found in Log.h. The API helps to log either successful or failed CRUD operations

Successful Message API →
Log::logCreateSuccess, Log::logReadSuccess, Log::logUpdateSuccess, Log::logDeleteSuccess
Failure Message API →
Log::logCreateFail, Log::logReadFail, Log::logUpdateFail, Log::logDeleteFail

What all nodes should log the messages?

- All replicas (non-faulty only) should log a success or a fail message for all the CRUD operations
- If the coordinator gets quorum number of successful replies then it should log a successful message, else it should log a failure message

Test Cases

Following are the test cases covered in this application

- Basic CRUD tests that test if 3 replicas respond
- Single failure followed immediately by operations which should succeed (as quorum can still be reached with 1 failure)
- Multiple failure followed immediately by operations which should fail as quorum cannot be reached
- Failures followed by a time for the system to re-stabilize, followed by operations that should succeed because the key has been re-replicated again at 3 nodes

Future work

As the application is built for single threaded environment. It can be easily ported to real Distributed System. The EmulNet can be easily replaced by real socket layer and with the help of multi-threading the application will behave fully like real Key-Value store.