Key-Value Store User Manual

Jash Dave

1 Client Installation

To install a client for a particular key-value store go to path "KeyValueStore/Implementation/(Your required key-value store)/client/src_v2/" and run the make file with required permissions. Note that default target for make file is to compile and install. Also note that it may download few dependencies so make sure you have working internet connection.

Installing the client installs a common library kvs_kvstore_v2 and key-value store specific libraries vis kvs_redis_v2 for Redis, kvs_memcached_v2 and memcached for Memcached and kvs_leveldb_v2 for LevelDB. You must link to these libraries while producing your final binary. You must also link boost_serialization and pthread library.

2 Server Installation

To install a server for a particular key-value store go to path "KeyValueStore/Implementation/(Your required key-value store)/server/" run the install_server.sh script with required permission. Note that it may download few dependencies so make sure you have working internet connection.

2.1 Starting a particular key-value store server

Once the installations is successful, you may start the particular key-value store server as follows

- **LevelDB**
  Run the run_server.sh script from "KeyValueStore/Implementation/LevelDB/server/" folder, with socket address as parameter (provide the required permissions). For eg.
  "bash run_server.sh 10.129.26.154:8090"
  Configuration string for client is IP:PORT for eg. "10.129.26.154:8090".

- **Memcached**
  To start Memcached server go to folder
  "KeyValueStore/Implementation/Memcached/server/memcached-1.4.35/"
  and run the command
  "./memcached -p [PORT_NUMBER] -m [MEMORY_IN_MB]"
  Configuration string for client is list of servers prefixed with "–SERVER="
  and separated by space. For eg.
  "–SERVER=10.129.26.154:12000 –SERVER=10.129.28.207:12345"

- **Redis**
  Go to "KeyValueStore/Implementation/Redis/server/" folder. For the very first time you must run the create_servers.sh script with count, IP address and starting port number as parameter for eg.
  "bash create_servers.sh 3 10.129.26.154 8000"
  It will create three servers with given IP as 10.129.26.154 and ports ranging from 8000 to 8002. To start the servers run the script start_servers.sh with count as parameter for eg.
  "bash start_servers.sh 3"
  Once the servers are running, do the following (for first time only) to setup the key distribution. Go to folder
  "KeyValueStore/Implementation/Redis/server/redis-3.2.4/src/"
  and run the command
  "./redis-trib.rb create –replicas 0 IP:PORT IP:PORT IP:PORT ..."
  where IP:PORT must be replaced with server IP and PORT numbers. Note that Redis cluster needs at least 3 server instances, thus you must run at least three servers. For more information please refer
  "https://redis.io/topics/cluster-tutorial"
  Configuration string for client is any one server's IP:PORT for eg.
  "10.129.26.154:8000".

This section only eases the installation of server by proving installation scripts, but you can always install, configure and start the required server your own way. The only thing you must ensure is that the server version must be same or it must be compilable with the version we have used.

3 Client Interface Description

To use the interface you must import the header file "kvstore/KVStore-Header_v2.h". This interface is implemented in the "kvstore" namespace. Most of the interface is designed using templates thus making it generic

enough. KeyType, ValType, Function and Argument are the repsentative template types. Note that the classes used for KeyType and ValueType must implement boost library's serialization function.

To keep the design simple, interface exposes only required features of any data store. Also to make it generic enough it uses template parameters, thus the interface is also independent of changes in LTE-EPC code. It consists of the following classes:

- **KVData<ValType>**
  This class is designed to return the response of any operation. It has the following public data members:
  - **ierr** :
    Represents the integer code of an error. It's value is negative if any error occurred, otherwise it is zero.
  - **serr** :
    It gives the description of error in string from for human readability.
  - **value** :
    It contains the return value of an operation, only if no error has occurred. This field is valid only for data fetch operations (like get, async_get, and sget).
- **KVStore<KeyType, ValType>**
  This class exposes the functions to put, get and delete data from the data store. It requires two template parameters viz KeyType and ValueType during declaration (just like map data structure in C++ STL). It has the following functions:
  - **bool bind(string connection, string tablename) :**
    Its first parameter is a connection string it describes the socket address or server list of the key-value store being connected and is specific to key-value store being connected.

    Its second parameter is table name, and is describes the table on which the KVStore object will perform all its operations.

    It connects to the data store and creates the table with the give table name or binds to the table described by table name, if it already exists. It returns true if the connection was successful

and it was able to bind with given table, otherwise false. All the other operations called on this object will now operate on the attached table.

- **KVData<ValType> put(KeyType key, ValType val) :**
  This function takes in a key and value as its parameters, and pushes it to the attached table in the key-value store. Key and value must be of the type KeyType and ValueType used during declaration of KVStore object. It returns an object of KVData class, whose ierr field can be used to check the success of this put operation.

- **KVData<ValType> get(KeyType key) :**
  This function takes a key as input and fetches its corresponding value from the attached table. It returns an object of KVData class, whose value field can be used to retrive the returned value, if no error occurred.

- **KVData<ValType> del(KeyType key) :**
  This function deletes the value corresponding to given key from the attached table and returns an object of KVData class which can be checked for errors, if any.

- **void async_put(KeyType key, ValType val, Function *f, Argument... args) :**
  It performs the same task as put function, but asynchronously. This function along with a key and a value also takes a callback function as its parameters along with its arguments. The callback function passed must have its last parameter as of type KVData<ValType>. Ones the put request is completed, it invokes the passed callback function with KVData as result passed to the callback function.

- **void async_get(KeyType key, Function *f, Argument... args) :**
  This is the asynchronous version of get function. This function also takes the callback function as the parameter in addition to key as input and fetches its corresponding value from the attached table. Once the fetch is completes or failes it invokes the callback

function along with the result.

- **void async_del(KeyType key, Function *f, Argument... args) :**
  This is also an asynchronous function, and it performs the same task as that of del function. It deletes the value corresponding to given key from the attached table and invokes the callback function once the assigned task gets completed.

- **bool clear() :**
  This function deletes all the data from attached table, and thus must be used with caution. It returns true if it was successful in removing all entries, otherwise false. Note that clear function is not supported by all the data stores and thus it always returns false for these data stores.

- **KVRequest**
  This class is designed to send multiple requests in one go. Thus if multiple operations can be merged in one request, then we must use this to save upon multiple round trip times. It exposes the following functions:

  - **bool bind(string connection) :**
    It takes a string as the parameter describing the socket address or server list of key-value store. It returns true if connection was successful, otherwise false. Note that this class doesn't bind to any table. Table name is to be provided as a parameter to each operation.

  - **void put<KeyType, ValType>(KeyType key, ValType val, string tablename) :**
    It registers a put request to be executed later on the given table. It gets executed when execute function is called on KVRequest object holding it.

  - **void get<KeyType, ValType>(KeyType key, string tablename) :**
    It registers a get request with a given key to be executed later on the given table.

- **void del<KeyType, ValType>(KeyType key, string tablename) :**
  It registers a delete request with a given key to be executed later on the given table.

- **void sput<KeyType, ValType>(KeyType key, string tablename) :**
  Safe put is a special version of put function, it must always be called only after calling a sget function on the same key, otherwise it returns a failure. Safe put only succeeds if the value fetched by the corresponding sget has not yet been changed by anyone. This function ensures that the value fetched by sget and the value updated by sput are the same. It registers a sput request with a given key to be executed later on the given table.

- **void sget<KeyType, ValType>(KeyType key, string tablename) :**
  This is the special version of get function to be used before sput. This function fetches value and the version number associated with the value, to be used later by sput function.

- **KVResultSet execute() :**
  This function executes all the registered requests with the KVRequest object and returns a KVResultSet object, which holds results for all the requests. Note that it does not ensures the sequence of execution of operations, all the operations registered in single request can be executed in any oreder. But the return order is same as that of registering the requests.

- **void async_execute(Function *f, Argument... args) :**
  This is the asynchronous version of execute function, is performs the same task as that of execute function, and invokes the callback function with the ResultSet object, once the assigned task is completed.

- **reset() :**
  It resets the request queue and clears the KVRequest object. It must be called after execute or async_execute function if you don't

want to repeat the same sequence of request again.

- **KVResultSet**
  This class holds the results obtained by executing the requests in queue of KVRequest object. It has the following functions:
  - **int size() :**
    Returns the number of results present in KVResultSet object, it corresponds the operations executed by KVRequest object.
  - **KVData get<ValType>(int index) :**
    It returns the output of the operation executed by KVRequest. Output is returned in the same order as the requests were queued in KVRequest object, i^{th} object represents the result of i^{th} request in the queue, index stating from zero.
  - **string oprType(int idx) :**
    It returns the operation type of the function registered on the given index as a string constant. Constants returned are OPR_TYPE_GET, OPR_TYPE_PUT, OPR_TYPE_DEL, OPR_TYPE_SGET and OPR_TYPE_SPUT.

4 Example Code

4.1 KVStore

```
#include <iostream>
#include <kvstore/KVStoreHeader_v2.h>
using namespace std;
using namespace kvstore;

int main(){
  /* Declare the KVStore object with KeyType and ValType */
  KVStore<int,string> ks;

  /* Establish connection to key-value store*/
  bool succ = ks.bind("10.129.26.154:8090","MyTable");
  if(succ){
    cout<<"Connection successful."<<endl;
  } else {
```

```
    cout<<"Problem connecting."<<endl;
    return -1;
  }

  /* Storage for return value */
  KVData<string> ret;

  /* Put a key in key-value store */
  ret = ks.put(1, "One");
  if(ret.ierr == 0){
    cout<<"Put successful."<<endl;
  } else {
    cout<<"Problem with put. Error Description:"<<ret.serr<<endl;
  }

  /* Get a value from key-value store */
  ret = ks.get(1);
  if(ret.ierr == 0){
    cout<<"Get successful. We got the value "<<ret.value<<endl;
  } else {
    cout<<"Problem with get. Error Description:"<<ret.serr<<endl;
  }

  /* Delete a value from key-value store */
  ret = ks.del(1);
  if(ret.ierr == 0){
    cout<<"Del successful."<<endl;
  } else {
    cout<<"Problem with del. Error Description:"<<ret.serr<<endl;
  }
}
```

4.2 KVRequest

```
#include <iostream>
#include <kvstore/KVStoreHeader_v2.h>
using namespace std;
using namespace kvstore;

int main(){
```

```
KVRequest kr;
KVData<string> ret;

/* Establish connection to key-value store*/
kr.bind("10.129.26.154:8090");

/* Register few requests to be executed */
kr.put<int,string>(1, "One", "MyTable1");
kr.put<int,string>(2, "Two", "MyTable1");
kr.put<int,string>(3, "Three", "MyTable2");

/* Execute the requests */
KVResultSet rs = kr.execute();

/* Clear the request queue */
kr.reset();

/* Parse the result */
for(int i = 0; i < rs.size() ; i++){
  ret = rs.get<string>(i);
  if(ret.ierr != 0) {
    cout<<"Operation␣"<<i<<"␣failed.␣Error:"<<ret.serr<<endl;
  }
}

/* Reuse the KVRequest object */
/* Register few requests to be executed */
kr.get<int,string>(1, "MyTable1");
kr.get<int,string>(2, "MyTable1");
kr.get<int,string>(3, "MyTable2");

/* Execute the requests */
rs = kr.execute();

/* Clear the request queue */
kr.reset();

/* Parse the result */
for(int i = 0; i < rs.size() ; i++){
  ret = rs.get<string>(i);
```

```
    if(ret.ierr != 0) {
      cout<<"Operation␣"<<i<<"␣failed.␣Error:"<<ret.serr<<endl;
    } else {
      cout<<"Value␣for␣operation␣"<<i<<"␣is␣"<<ret.value<<endl;
    }
  }

  /* We can also club multiple type of operations. */
  /* But note that the order of execution is not ensured. */
  kr.put<int,string>(10, "Ten", "MyTable1");
  kr.get<int,string>(10, "MyTable1");
  kr.del<int,string>(10, "MyTable1");

  /* Execute the requests */
  rs = kr.execute();

  /* Clear the request queue */
  kr.reset();

  /* Parse the result */
  for(int i = 0; i < rs.size() ; i++){
    ret = rs.get<string>(i);
    if(ret.ierr != 0) {
      cout<<"Operation␣"<<i<<"␣failed.␣Error:"<<ret.serr<<endl;
    }
  }
}
```

4.3 Using custom data type as key or value

For using the custom data type as key or value you must implement the serialize function of boost serialization library. For more details on boost serialization please refer "http://www.boost.org/doc/libs/1_64_0/libs/serialization/doc/tutorial.html". Example code below shows a simple use case of custom data types.

```
#include <iostream>
#include <kvstore/KVStoreHeader_v2.h>
using namespace std;
using namespace kvstore;
```

```
class MyCustomType{
  private:
    double mydouble;
    string mystring;
  public:
    /* Constructors */
    MyCustomType(){
    }

    MyCustomType(double md, string ms){
      mydouble = md;
      mystring = ms;
    }

    /* Setters and Getters */
    void setMyDouble(double val){
      mydouble = val;
    }
    double getMyDouble(){
      return mydouble;
    }
    void setMyString(string val){
      mystring = val;
    }
    string getMyString(){
      return mystring;
    }

    /* Function to be implemented for boost serialization */
    template<class Archive>
    void serialize(Archive &ar, const unsigned int version)
    {
      /* Add all the variables you wanna save, add them
        to archive separated by & operator */
      ar & mydouble & mystring;
    }
};
```

```
int main(){
  /* Declare the KVStore object with KeyType and ValType */
  KVStore<double,MyCustomType> ks;

  /* Establish connection to key-value store*/
  bool succ = ks.bind("10.129.26.154:8090","MyTable");
  if(succ){
    cout<<"Connection␣successful."<<endl;
  } else {
    cout<<"Problem␣connecting."<<endl;
    return -1;
  }

  /* Storage for return value */
  KVData<MyCustomType> ret;

  /* Put a custom value on key-value store */
  MyCustomType mytype(1.1,"SomeString");
  ret = ks.put(3.14,mytype);
  if( ret.ierr != 0 ){
    cout<<"Error␣in␣put␣:"<<ret.serr<<endl;
  }

  /* Get the value from key-value store */
  MyCustomType retval;
  ret = ks.get(3.14);
  if( ret.ierr != 0 ){
    cout<<"Error␣in␣get␣:"<<ret.serr<<endl;
  } else {
    retval = ret.value;
    cout<<"Got␣mydouble␣:␣"<<retval.getMyDouble()<<endl;
    cout<<"Got␣mystring␣:␣"<<retval.getMyString()<<endl;
  }

  return 0;
}
```

4.4 Using asynchronous functions

Note that none of the classes in kvstore namespace are thread safe. Thus you must ensure no two threads are operating on same object at the same time. Thus you must refrain using synchronous functions along with asynchronous functions, unless you are sure that both are operating in non-overlapping manner. Below listing shows the use of asynchronous function.

```
#include <iostream>
#include <kvstore/KVStoreHeader_v2.h>
using namespace std;
using namespace kvstore;

int main(){
  KVRequest kr;
  KVData<string> ret;

  /* Establish connection to key-value store*/
  kr.bind("10.129.26.154:8090");

  /* Register few requests to be executed */
  return 0;
}
```