# MP2 Specification Document

## 1. Overview: Building a Fault-Tolerant Key-Value Store

In this MP, you will be building a fault-tolerant key-value store. We are providing you with the same template provided for C3 Part 1 Programming Assignment (Membership Protocol), along with an almost-complete implementation of the key-value store, and a set of tests (which don't pass on the released code). This means first you need to use your working version of Membership Protocol from C3 Part 1 Programming Assignment and integrate it with this assignment. Then you need to fill in some key methods to complete the implementation of the fault-tolerant key-value store and pass all the tests.

If you did not complete the MP from C3 Part 1, please do so before you continue with the rest of this document and MP. Here is a link to the zip file containing the MP from C3 Part 1.

First, make sure that you have a working version of C3 Part 1 Programming Assignment. Use the MP1Node.cpp and MP1Node.h from that working version and replace the MP1Node.cpp and MP1Node.h in the template provided to you. Please note that this MUST be your own code. Now that you have figured out how to integrate the Membership Protocol, back to the description of the Fault-Tolerant Key-Value Store.

Concretely, you will be implementing the following functionalities:

- A key-value store supporting **CRUD operations** (Create, Read, Update, Delete).
- **Load-balancing** (via a consistent hashing ring to hash both servers and keys).
- **Fault-tolerance** up to two failures (by replicating each key three times to three successive nodes in the ring, starting from the first node at or to the clockwise of the hashed key).
- **Quorum consistency level** for both reads and writes (at least two replicas).
- **Stabilization** after failure (recreate three replicas after failure).

This programming assignment uses C++. It is one of the more commonly used languages in industry for writing systems code. For this, you will need at least C++11 (gcc version 4.7 and onwards).

**The only files you can change are MP1Node.{cpp,h} and MP2Node.{cpp,h}.**

**Academic Integrity:** All work in this MP must be individual, i.e., no groups. You can talk with others about the MP specification and concepts surrounding the MP, but you can neither discuss solutions or code, nor share code. Please refer to the academic integrity description on the course website. The usual academic integrity rules of Coursera apply here as well.

## 2. What the Code Does and What to Implement

Similar to C3 Part 1 MP, we are providing you with a three-layer implementation framework that will allow you 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: MP1Node (membership protocol) and the MP2Node (the key-value store), and 3) the application layer.

We are providing you with an implementation of an emulated network layer (EmulNet). 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. Think of this like a three-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: *MP1Node* and *MP2Node*. *MP1Node* runs a membership protocol that you implemented in C3 Part1 MP. *MP2Node*, which you will also implement should support all the KV Store functionalities. 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.

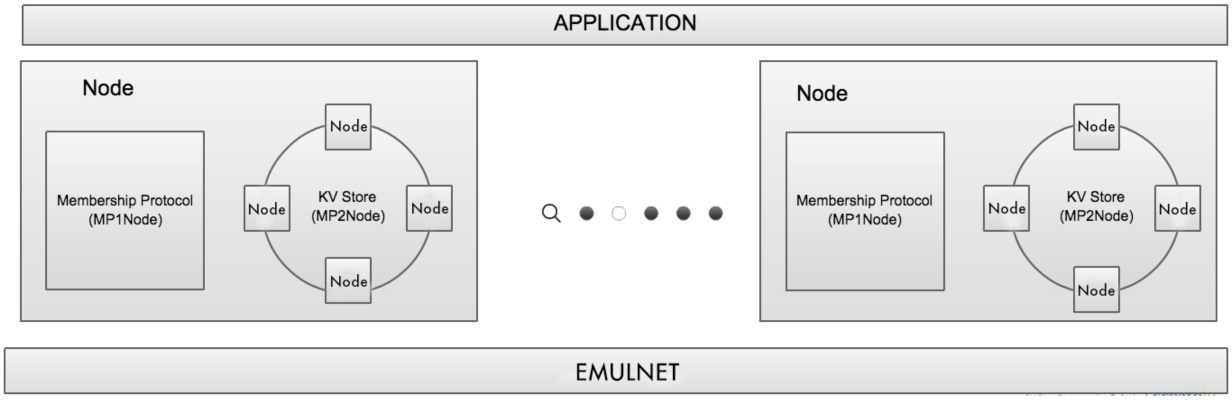

Figure 1: The Three Layers

Please note that the membership list may be stale at nodes! This models the reality in distributed systems. So, code that you write must be aware of this. Also, when you react to a failure (e.g., by re-replicating a key whose replica failed), make sure that there is no contention among the would-be replicas. Do not over-replicate keys!

Each MP2Node should implement both the client-side as well as the server-side APIs for all the CRUD operations. The application layer chooses a non-faulty node randomly as the client. The same node can be considered as the coordinator. You can assume that the coordinator never crashes. Your Key-Value store should accept std::string as key and value.

Some of the important classes in MP2 are:

- **HashTable:** A class that wraps C++11 std::map. It supports keys and values which are std::string. This has already been implemented and provided to you. You can either use this or have your own implementation of a hash table inside MP2Node.
- **Message:** This class can be used for message passing among nodes. This has already been implemented and provided to you. You can either use this or have your own implementation in MP2Node.
- **Entry:** This class can be used to store the value in the key-value store. This has already been implemented and provided to you. You can either use this or have your own implementation in MP2Node.
- **Node:** This class wraps each node's Address and the hash code obtained by consistently hashing the Address. The upcall to MP1Node returns the membership list as a std::vector<Node>.
- **MP2Node:** This class must implement all the functionalities of a key-value store, which include the following:
    - Ring implementation including initial setup and updates based on the membership list obtained from 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

The only file you should change is MP1Node.{cpp,h} and MP2Node.{cpp,h}. Like C3 Part 1 MP, when running a test, your code will be generating a log file (dbg.log) that will be then automatically checked for correctness. Please use the log messages provided in the code to generate the log file.

## 3. What Should I Change?

The methods in MP2Node which you should implement are:

- **MP2Node::updateRing**: This function should set up/update the virtual ring after it gets the updated membership list from MP1Node via an upcall. For more information, look at the function prologue in MP2Node.cpp.
- **MP2Node::clientCreate, MP2Node::clientRead, MP2Node::clientUpdate, MP2Node::clientDelete** : These function should implement the client-side CRUD interfaces. For more information look at the respective function prologue in MP2Node.cpp.
- **MP2Node::createKeyValue, MP2Node::readKey, MP2Node::updateKeyValue, MP2Node::deletekey**: These functions should implement the server-side CRUD interfaces. For more information look at the respective function prologue in MP2Node.cpp.
- **MP2Node::stabilizationProtocol()**: This function should implement the stabilization protocol that ensures that there are always three replicas of every key in the key value store.

**While making your changes, the only files you can add changes to are MP2Node.cpp, MP2Node.h, MP1Node.cpp and MP1Node.h. If you delete or modify anything already in these files, do so at your own risk. Do not change or add to any other file**. **However, if you do not want to use the classes provided by us and want to add your own classes, add the classes to MP2Node.{cpp,h}.**

## 4. Logging

Use the following functions provided to you in Log.h to log either successful or failed CRUD operations (remember that this is how the grader will check for correctness of your implementation):

- Log::logCreateSuccess, Log::logReadSuccess, Log::logUpdateSuccess, Log::logDeleteSuccess: Use these functions to log successful CRUD operations.
- Log::logCreateFail, Log::logReadFail, Log::logUpdateFail, Log::logDeleteFail: Use these functions to log failed CRUD operations.

**What nodes should log the messages ?**

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

## 5. Test Cases

The tests include:

- Basic CRUD tests that test if three replicas respond
- Single failure followed immediately by operations which should succeed (as quorum can still be reached with 1 failure)
- Multiple failures 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 three nodes.

For more information about the test, see the comments in Application.cpp starting at Line 232.

## 6. Testing Your Code

Thoroughly test your solution locally for all the CRUD operations using KVStoreGrader.sh provided to you.

- MP2 is worth a total of 90 points.
- When you run the grader (KVStoreGrader.sh), it tells you whether you passed all the tests (out of 90).
- You should add code into the requisite portions so that your code passes all tests on the Coursera grading platform.

## 7. Submitting Your Solution to Coursera

Submit your solution to Coursera. Check the due date at the top of this document. Please follow the detailed instructions given below:
Once you have tested your solution and are confident it works, then submit your solution to Coursera by following these steps:

- Locate the included python script named file named “submit.py”
- Run the script from the terminal:
  ```
  $ python submit.py
  ```
- The script will ask for login details.
- You can check the score of your submission. Please note that it takes a while (sometimes a few hours) for your submission to be graded. Please be patient!

## 8. If You Finish Your MP Early ...

The MP has been designed for use of porting to a real distributed system. If you finish the MP early and all your tests are passing, you may want to look into making your MP code run on a truly distributed system. Start by changing the EmulNet layer and then perhaps using multithreading. You can also change any of the underlying classes and implementations if you think that will make the code more efficient. If you can get a version working across at least three machines, then you have a fully working key-value store! (There will not be extra credit for this portion, but it’s incredibly useful for you to tell your friends, relatives, and interviewers that you’ve build a real working key-value store!)