CS425/ECE 428 Fall 2014
Machine Program 2 - Fault-tolerant Key-value Store

Important Dates
Released: November 7th, 2014
Due: 11.59 PM, Dec 3rd, 2014 (Wednesday)

1. What is this MP about?

In this MP, you will be building a fault-tolerant key-value store. We are providing you with a template that contains a working solution to MP1 (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). You need to fill in some key methods to complete the implementation and pass all the tests.

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 2 failures (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).
- **Quorum consistency level** for both reads and writes (at least 2 replicas).
- **Stabilization** after failure (recreate 3 replicas after failure).

In response to student feedback, MP2 is in 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). If you're using an EWS Linux system, run "module load gcc" first in the shell -- this will put a 4.8.2 version of gcc at the start of your path.

The only files you can change are 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. We will check your code to find similarities in structure as well as ideas. Please refer to the academic integrity description on the course website.

2. What the code does, and What to Implement?

Similar to MP1, 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 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: *MP1Node* and *MP2Node. MP1Node* runs a membership protocol similar to the one you implemented in MP1. This has already been implemented for you. *MP2Node,* which you will 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.

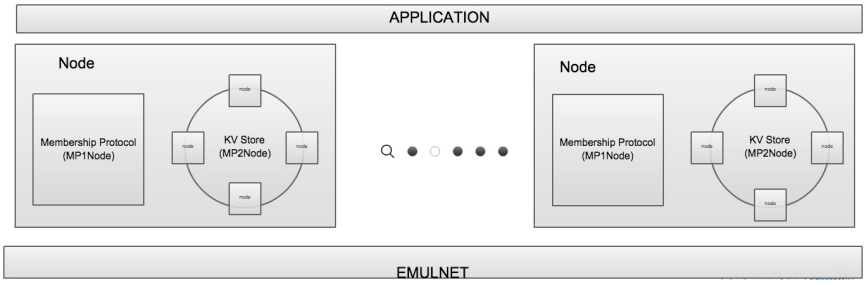

Figure 1. The three layers

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 distributed hash table (DHT) 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 DHT. 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<Nodes>.
- 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 DHT
    - Stabilizing the DHT whenever there is a change in membership
    - Client-side CRUD APIs
    - Server-side CRUD APIs

The only file you should change is MP2Node.{cpp,h}. Like MP1, 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 DHT.

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.

The tests include:

- 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.

For more information about the test look at the comments in Application.cpp starting from Line 232.

The grading scripts will be released a few days after Nov 7th, 2014.

While making your changes the only files you can add changes to are MP2Node.cpp and MP2Node.h. If you delete or modify anything already in these two 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. Submitting

We will upload detailed submission instructions soon. Like MP1, you will be testing your code via the Coursera server, and also submitting your code via SVN.

5. I am itching to code !! How do I start ?

Checkout out the code from SVN by running this command:

svn co https://subversion.ews.illinois.edu/svn/fa14-cs425/_shared
cd MP2
cp -R ./* <directory_on_your_linux_box>

6. 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 3 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!)

All the best !