MP1 Specification Document

Important Dates

- **Released:** Week 1 of C3 Part 1
- **Due:** End of Week 5 C3 Part 1

1. Overview: What Is This MP About?

This MP (Machine Programming assignment) is about implementing a membership protocol similar to one we discussed in class. Since it is infeasible to run a thousand cluster nodes (peers) over a real network, we are providing you with an implementation of an emulated network layer (EmulNet). Your membership protocol implementation will sit above EmulNet in a peer- to-peer (P2P) layer, but below an App layer. Think of this like a three-layer protocol stack with Application, P2P, and EmulNet as the three layers (from top to bottom). More details are below.

Your protocol must satisfy: i) Completeness all the time: every non-faulty process must detect every node join, failure, and leave, and ii) Accuracy of failure detection when there are no message losses and message delays are small. When there are message losses, completeness must be satisfied and accuracy must be high. It must achieve all of these even under simultaneous multiple failures.

You can choose to implement any of the membership protocols we learned in class: all-to-all heartbeating, gossip-style heartbeating, or SWIM-style membership. We recommend either gossip or SWIM, since you will learn the most this way.

The template code is written in C++. It is one of the most 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 template code can be downloaded from the following URL: https://spark-public.s3.amazonaws.com/cloudcomputing/assignments/mp1/mp1.zip

We will be providing you with the autograder scripts (unit tests) that you can use to test that your program passes all requirements (Section 6).

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, code, nor share code. We might check your code to find similarities in structure and ideas. You will also learn the most if you do this work individually!

2. The Three Layers

The three-layer implementation framework we are providing will allow you to run multiple copies of peers within one process running a single-threaded simulation engine. Here is how the three layers work.

2.1 Emulated Network: EmulNet

EmulNet provides the following functions that your membership protocol above should use:

- `void *ENinit(Address *myaddr, short port);`
- `int ENsend(Address *myaddr, struct address *addr, string data);`
- `int ENrecv(Address *myaddr, int (* enqueue)(void *, char *, int), struct timeval *t, int times, void *queue);`
- `int ENcleanup();`

ENinit is called once by each node (peer) to initialize its own address (myaddr). ENsend and ENrecv are called by a peer respectively to send and receive waiting messages. ENrecv enqueues a received message using a function specified through a pointer enqueue(). The third and fourth parameters (t and times) are unused for now. You can assume that ENsend and ENrecv are reliable (when there are no message losses in the underlying network). ENcleanup is called at the end of the simulator run to clean up the EmulNet implementation. These functions are provided so that they can later be easily mapped onto implementations that use TCP sockets.

Please do not modify the EmulNet.cpp,h files given to you. We will replace it with our own implementations during testing. You should only use the above functions to access the EmulNet layer, and should not access the EmulNet data structures directly.

2.2 Application

This layer drives the simulation. Files Application.{cpp,h} contain code for this. Look at the main() function. This runs in synchronous periods (globaltime variable).

During each period, some peers may be started up, and some may be caused to crash- stop. Most importantly, for each peer that is alive, the function nodeLoop() is called. nodeLoop() is implemented in the P2P layer (MP1Node.{cpp,h}) and basically receives all messages that were sent for this peer in the last period, as well as checks whether the application has any new waiting requests.

Please do not modify the Application.cpp,h files given to you.

2.3 P2P Layer

The functionality for this layer is pretty limited at this time. Files MP1Node.{cpp,h} contain code for this. This is the layer responsible for implementing the membership protocol. As such this is where your code should be implemented. You can very well imagine the P2P layer can be extended to provide functionalities like file insert, lookup, remove etc.

3. What Does the Code Do Currently?

As given to you, the code prints out debugging messages into dbg.log (format is node address [globaltime] message). You can turn debugging on or off by commenting out the #DEFINE DEBUGLOG in stdincludes.h.

Two message types are currently defined for the P2P layer (MP1Node.cpp implementation) - JOINREQ and JOINREP. Currently, JOINREQ messages are received by the introducer. The introducer is the first peer to join the system (for Linux, this is typically 1.0.0.0:0, due to the big-endianness). The best place to start your implementation is to have the introducer reply to a JOINREQ with a JOINREP message. The next section lists all the functionalities you have to implement.

4. What Do I Implement?

All your code will go into the P2P layer in file MP1Node.{cpp,h}. Do not make changes to any other file other than MP1Node.cpp,h as they will be replaced.

You will need to of course implement nodeLoopOps(), and the recvCallBack() functions. Both functions are invoked by nodeLoop() to periodically perform protocol routines. Specifically, they should have functionalities loosely described below, which means you have considerable flexibility in choosing implementation details, except that your final submission should work with the other files unchanged. We will be implementing a membership protocol. Here are the functionalities your implementation must have:

- Introduction: Each new peer contacts a well-known peer (the introducer) to join the group. This is implemented through JOINREQ and JOINREP messages. Currently, JOINREQ messages reach the introducer, but JOINREP messages are not implemented. JOINREP messages should specify the cluster member list. The introducer does not need to maintain a list of all peers currently in the system; a partial list of fixed size can be maintained.
- Membership: You need to implement a membership protocol that satisfies completeness all the time (for joins and failures), and accuracy when there are no message delays or losses (high accuracy when there are losses or delays). We recommend implementing either gossip-style heartbeating or SWIM-style membership, although all to all heartbeating would be fine too (though you'd learn less). See lecture slides for more details.

Some of the things that you will probably need to modify are the struct member and enum MsgTypes in Mp1Node.h. You need to handle the new message types in separate Process functions similar to JOINREQ/JOINREP for this.

4.1 Logging

Logging your events is critical as the grader scripts (Section 6) look at the logs.

Log.{cpp,h} has a LOG() function that prints out node status into a file named dbg.log. Also it implements two functions logNodeAdd and logNodeRemove. Whenever a process adds or removes a member from its membership list, make sure you use logNodeAdd and logNodeRemove to log these respectively. The grader scripts will look for these log entries when running the tests. These functions take two address parameters - pass the address of the recording process as the first parameter and the address of the process getting added/removed as the second parameter.

5. What Are These Other Files?

Params.{cpp,h} contains the setparams() function that initializes several parameters at the simulator start, including the number of peers in the system(EN_GPSZ), and the global time variable globaltime, etc.

The remaining files Member.cpp,h list some necessary definitions and declarations -- see descriptions in the files. Avoid modifying these files.

Why is the Code Structure So Involved? There are two reasons. Firstly, think about the issues involved in converting this into a real application. All EN*() functions can be easily replaced with a different set that sends and receives messages through sockets. Then, once the periodic functionalities (e.g., nodeLoop()) are replaced with a thread that wakes up periodically, and appropriate conversions are made for calling the other functions nodeStart() and recvLoop(), your implementation can be made to run over a real network!

Secondly, this structure allows us to debug (and even measure the performance through traces) the membership protocol easily and on a single host machine. Compare this with the debugging challenge for several hundred processes running on a real network. Once the simulation engine works, you can convert the implementation easily into one for a real network, and it will work.

6. How Do I Test My Code?

6.1 Testing

To compile the code, run make.

To execute the program, from the program directory run: ./Application testcases/<test_name>.conf. The conf files contain information about the parameters used by your application:

```
MAX_NNB: val
SINGLE_FAILURE: val DROP MSG: val
MSG_DROP_PROB: val
```

where MAX_NNB represents the max number of neighbors, SINGLE_FAILURE is a one bit 1/0 variable that sets single/multi failure scenarios, MSG_DROP_PROB represents the message drop probability (between 0 and 1) and MSG_DROP is a one bit 1/0 variable that decides if messages will be dropped or not.

There is a grader script Grader.sh. It tests your implementation of membership protocol in 3 scenarios and grades each of them on 3 separate metrics. The scenarios are as follows:

1. Single node failure
2. Multiple node failure
3. Single node failure under a lossy network.

The grader tests the following things: i) whether all nodes joined the peer group correctly, ii) whether all nodes detected the failed node (completeness) and iii) whether the correct failed node was detected (accuracy). Each of these is represented as configuration files inside the testcases folder.

6.2 Grading

MP1 is worth a total of 90 points. When you run the grader 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.

6.3 How Do I Run the Grader on My Computer?

Use the grader (i.e., Grader.sh) provided to you in the code template to test your solution locally in your computer as many times as you want. Make sure you are using at least C++11 (gcc version 4.7 and onwards).

```
$ chmod +x Grader.sh
$ ./Grader.sh
```

6.4 How Do I Submit to Coursera?

Once you are satisfied with your solution, use the submission script (submit.py) provided to you in the code template to submit your solution to Coursera.

```
$ python submit.py
```

Before you submit, make sure you back up all your files and work in a different directory. The script will prompt for login details. Login id is your **Coursera id**. Password is your **Coursera one-time password**. You can find your Coursera one-time password on the **Programming Assignment** page.

You can also check the score of your submission on the Programming Assignment web page. Please note that it takes a while (sometimes a few hours) for your submission to be graded. Please be patient! If you make multiple submissions, then the submission with the highest score will be considered for grading.