Distributed Systems MP1 **Group 54**

Ambati, Teja(tambati2) Sypereddi, Anoop(anoops2)

February 2019

URL: `https://gitlab.engr.illinois.edu/tambati2/cs425-mp1`
Hash: 2a2445819ef51f0515df8ba28f428bc87c1f6e00

1 Runnning Instructions

mp1.go inside /home/anoops2
Build Instructions: go build mp1.go
Running Instructions: ./mp1 $NAME $PORT $NUM_NODES

All debug output inside of *mp1.log* and if you type -v at the end, you can display verbose output at runtime

2 Protocols

We took advantage of net.Conn in order to inplement our distributed chat system because it is the default networking package included with golang.

2.1 B-multicast

Nodes on our cluster implemented multicast in 2 parts, one as a reviecer and one as a sender. We will first discuss the sending portion of our implementation since it is the simpler of the two.

2.1.1 Sending Messages

When nodes begin the program, they have to input a name, a port number, and the number of nodes *n_nodes* which will be in the cluster. If there is disagreement in *n_nodes*, then the cluster will misbehave. In this assignment we will assume that there is no disagreement and all nodes will have the same *n_nodes*. In the initialization portion of the program when we make connections with all of the other nodes, we end up with an array of $n\_nodes - 1$ net.Conn connections which can be written to and read from. we will have seperate net.Conn for recieving and sending messages in order to not have holdbacks which prevent messages from being delivered and recieved.

In order to send a message, we need to know what to send. Input comes in from stdin but the system will always keep reading. By convention the enter button is the universal 'send' and so we decided to read the contents of the buffer once the enter button is pressed by delimiting by newline characters at which point the contents of the buffer is copied to a string and the contents of the buffer is flushed. This string now holds the message which we intend to send. Once we know what we want to send, the program will iterate through all of the outbound connections inside of the $multicast(connections, message)$ function. This is a B-multicast function, but we have an implementation of R-multicast using a set of messages. Each message we send is encoded in json and contains the name, timestamp and the message. We send messages and store recieved messages in a set. If a message is recieved and it is not in the set, then we will first add the message struct to our set,then multicast to all the outbound nodes and then display the message. Doing this results in a R-multicast scheme which ensures that even if a node dies in the middle of delivering a message, all of the correct processes in a group will reviece the message and this will result in causal ordering.

2.1.2 Recieving Messages

Now that the message is in the pipe, it will be delivered to all the nodes in the cluster regardless of whether they are alive or dead. We are using net.Conn protocol so the communications between the nodes are basically two files. So a reader keeps reading from the file until it reaches a newline which was our delimitation character because it is the universal send button. Once it reads the newline from the pipe, the second node knows that the message has been delivered, in which case it will print out the message to stdout and start listening for new messages.

Unfortunately running this on a single thread would mean that we would need to switch between n_nodes readers continuously and this would not always lead to causal ordering since there could be the chance that p_3's message is delivered before p_1's message, but the current reader is currently p_4 and the next reader it reads form is p_1 which would yield that p_1's message appears before the message from p_3 even though p_3 sent the message first. In order to combat this we employed the use of multithreading. With multithreading, we could be have an individual thread mapped to each outbound node and whenever a message is delivered, the node can simply print the message to stdout. This however does not solve the problem which arises when a message is sent at the exact same time from two nodes. In the current implementation if all 10 nodes sent a message at the exact same time, then all of the nodes would recieve the message, but the order in which the messages are displayed is not consistent between nodes. There is also a mutex lock on stdout so messages will not be jumpled up between nodes.

2.2 Handling Failures

Every node has has *n_nodes* readers running on *n_nodes* different threads. Because the connection is a tcp connection, the reader will not close as long as there are no errors. Upon node failure the reader will detect that the file was closed and print that the specific node has exitted the chat. Then the thread for that specifc node will return. The remaining threads will remain unimpacted and all the nodes will continue functioning normally. Once all the reader threads close (one node left) the program will end. The program for the *n_nodes* threads is in *readerThread*(listener Listener, c chan int) and the threads are called and created in the *spawnReaders*(listeners[] Listener) function. Both calls occur from the main thread. Failures in the middle of a multicast is talked about above in the sending messages section of the paper.