Joshua Lew, Larry Liu
NetIDs: jtlew2, lliu65
CS425 / ECE428 Spring 2017
February 28, 2017

MP1 Design Documentation

Basic Design:

In our implementation of a chat client system, we start-up the system by having all nodes initiate connections to other clients with lower host name numbers than itself, and accept any incoming connections from other clients who have higher host name numbers. The advantage of this method for boot-up and the application itself is that we do not require a centralized node to control all connections and manage failure detection. The chat client requires that all nodes that desire to participate, must run the python script at around the same time (~5 seconds). For more detailed information around all aspects of our chat client, please see our comments in `chat.py`.

Total Ordering:

Our implementation utilizes the ISIS total ordering algorithm. The only potential alternative we are aware of is a leader/sequencer algorithm, but we wanted to avoid a single point of failure for increased reliability and functionality. In the leader algorithm, we would simply select an arbitrary node as the primary sequencer, which would evaluate the proposed priorities across all other nodes. However, we thought that this would have increased complexity if our system were to experience frequent failures. This would involve selecting another correct server as the new primary sequencer every time the leader process failed. Thus, we instead followed the standard algorithm that was discussed in the multicasting lecture. For receiving messages, we created a protocol with the following format: *<OP>%<H.ID>%<Priority>%<Message>#*. We used the # as a delimiter for properly reading in messages from our socket through TCP. The opcodes, 1-3, represent the three primary steps of instructions for multicasting a message in the ISIS algorithm. *H.ID* is the host number with a decimal point and the local message number from that host. This allowed us to quickly find messages in the queue without having to string compare the entire message. The priority is exactly as described in the algorithm, with the host number as the decimal point. For our priority queue, we used a regular python List. This allows us to easily find and update a message's priority in the queue. After marking a new message as deliverable, we sort the list by priority, and then deliver all the deliverable messages at the front of the list. To handle compiling the proposed priorities, we created two Lists representing the messages received and current max priority. We update these values after every response from a node. Once the node count equals the current nodes in the systems, we multicast the final priority.

Failure Detection:

In order to compensate for incorrect nodes, we will examine a server's sendall() method every time it attempts to multicast a message. Because our servers communicate via TCP, we are able to catch exceptions raised by invalid connections. Additionally, we had separate data structures to keep track of which processes have failed and which ones are correct. Therefore, we will never continue waiting on responses from processes that will never send their proposed priorities. For additional safety, we cleaned the priority queues from messages that have been sent by failed processes. This is to ensure we do not have an item in at the head of the queue that will never be marked as deliverable. Messages from a particular process that were delivered to some of the correct processes before failing are multicasted to other correct processes from those that received the message in order to preserve total ordering.

Performance Metrics:

Discussed in this section, are our measurements for several parameters of performance metrics on our chat client system. The bandwidth for starting up our system is 3 messages. This is because TCP utilizes a 3-way handshake to establish a connection between 2 endpoints. One node A will first send a TCP SYN data packet to node B. Where node B is accepting connections from node A's address. Node B then sends a SYN-ACK data packet to notify node A that it has received the SYN data packet. Finally, the third message is the ACK data packet sent from node A to node B to acknowledge that they have now established a connection. Also, the bandwidth of our failure detection implementation when no failure happens is 2 messages. In TCP, each message sent from one endpoint to another requires an acknowledge from the recipient of the message. Otherwise, an exception will be raised from the sender's send() or sendall() attempt to notify the application that the message was not properly sent. Failure detection time: 2 * [1-way message transmission time + added artificial network delay (e.g. the delays incurred from the provided command generating 50 - 100 ms)]. Our maximum throughput our multicast system can handle exactly 32 messages per process at a time. This is because we set the data structures holding the response values of messages sent (per server) is limited to size 32. We thought this was reasonable, since in this machine problem involving only 8 nodes, it is impractical for 1 specific node to send an enormous number of messages at a time, when it has to process messages received from other processes. However, if a process does decide to send a maximum of 32 messages at a time and wait on responses for proposed priorities, the processing of messages (and the ones that end up being delivered) will get significantly slower due to requirement of sorting a larger priority queue.

Resources Utilized:

Provided below is a list of major resources we referenced to throughout the course of development of the two parts of MP1. Major new concepts learned that were not introduced in lecture or previous classes include: basic socket programming, writing python programs, delivering messages via TCP, and general details about the TCP/IP architecture.

General Python Documentation:
https://docs.python.org/2/howto/sockets.html

Detecting a closed socket connection TCP:
http://stackoverflow.com/questions/4899593/python-doesnt-detect-a-closed-socket-until-the-second-send

Global variables in Python:
http://stackoverflow.com/questions/10588317/python-function-global-variables

Handling socket errors:
http://www.networkcomputing.com/applications/python-network-programming-handling-socket-errors/1384050408

Multicast Lecture Slides:
https://courses.engr.illinois.edu/cs425/fa2016/L14.FA16.pdf

Socket Timeout:
http://stackoverflow.com/questions/2719017/how-to-set-timeout-on-pythons-socket-recv-method

Simulating delayed and dropped packets:
http://stackoverflow.com/questions/614795/simulate-delayed-and-dropped-packets-on-linux/615757#615757

TCP 3-way Handshake:
http://www.inetdaemon.com/tutorials/internet/tcp/3-way_handshake.shtml

Understanding TCP System Call Sequences:
https://www.ibm.com/developerworks/aix/library/au-tcpsystemcalls/