Design:

We have one type of node in our network. It holds a list of all other nodes, and a queue of nodes that it is waiting for a response from. On startup, we pick up the local timestamp, and write a line to our logfile indicating that we are joining a pool. If we are the dedicated introducer node, represented by vm 01, we look for an existing\_nodes file to rejoin to an existing ring. If it exists, we load the existing\_nodes into our list of other nodes. In order to accommodate leaving and re-joining nodes, we created an ID class that contains a string for the hostname, and an integer timestamp that differentiates between the different instances of a node. If we are not the dedicated introducer node, we ping the introducer on a slow loop until we receive an ack response full of the other nodes in the list. At that point, we add those nodes to our other nodes list and proceed to the main loop.

Our main loop occurs at a rate of (1 second / (number of other nodes + 2)) to accommodate the time-bounded completeness requirement. Because we are required to guarantee time-bounded completeness, a random element of ordering of pings is unnecessary, and would require us to double our ping rate. Because of this, each node must ping each other node at least once per second, and a node that has not responded in 2 seconds or more is declared timed out. This allows us to guarantee that a failed node will be detected within 3 seconds. Once a node has failed, the failure detection message is sent to all other nodes in the membership list, guaranteeing that all non-failed nodes receive this information within 6 seconds. We are assuming a lossless network, as we could not make any guarantees under lossy network conditions.

In our main loop, we ping a node and append it to our waiting queue (if it is not already in that queue). If we are the introducer node, we write a quick state of the network to our existing\_nodes file, in case of introducer failure. We then receive the pings and messages that have been sent to our node, and process them, adding and removing nodes to our list and our waiting queue, as needed. If we catch a ctrl-C command, we treat this as a node leave request, and multicast our leave message to all other nodes in the list.

After receiving messages, we check our waiting queue. If a node in our queue has left or been reported as failed, we remove that node from our queue. If a node in the queue has been in that queue for more than 2 seconds, we declare this node as failed, and send that failure notification to all other nodes in our list.

Our program scales to large N because of the small size of most of our messages. Pings and Acks are only large enough to indicate the ID of the sender. However, because of the time-bounded completeness requirement, we could not simply randomize our ping order and populate data through random selection of nodes and pings and acks populated with data from other nodes. In this way, each additional node will add O(n^2) messages to the background, because it will have to ping every node within a certain time. Coordinating nodes to communicate to all members to ensure that all nodes will be pinged every X time with a high volume of joins and leaves was too difficult to solve for this problem.

We used MP1 to debug our program briefly, but found that just printing out what we would have put in the logfile to terminal was easier, because it allowed us to see results in real-time.

Our messages in the following format:

PING, ACK, LEAVE: int message\_type (PING, ACK, LEAVE), int timestamp of sending

node (to complete the ID of sender).

INTRO: int message\_type, int timestamp of sending node, sent only to introducer

INTRO\_ACK: int message\_type, int timestamp of sending node, ID of all existing nodes

in ring (so that we don’t have to put other IDs in the much more common PING or ACK). Sent only from introducer on a node join.

TIMEOUT: int message\_type, int timestamp of sending node, ID of node reported as failed.

(i) Background bandwidth usage for 4 nodes without membership changes, and with perfect network behavior: each node sends 6 pings and 6 acks per second.

12 messages / sec \* 2 integers = 96 bytes/second

(ii) bandwidth

- When a node joins: 1 INTRO message, 1 INTRO\_ACK message

INTRO message is 8 bytes, INTRO\_ACK is 8 bytes + N \* (4 + 1 \* len(HOSTNAME +2)):

4 for timestamp, message sent as string, 1 byte per character for IP address and 2 delimiting characters for each other node in the pool. So, assuming introducer is telling the joiner about 4 other nodes in the pool, HOSTNAME’s size is no more than 33, the total bandwidth would be <= 164 bytes.

- When a node leaves: to N nodes: 8 bytes each.

- When a node fails: to N nodes: 46 bytes each.

(iii) failure rate

Over a ten-minute time period, with N+2 messages per second, we had 1 failure with 2 nodes and a 3% loss rate, giving a false positive rate of very nearly 0%, and because of the low incident rate, a very high uncertainty of the failure frequency. Over a five-minute time period, with N+2 messages per second, we had 8 failures with 2 nodes and a 10% loss rate, giving a false positive rate of 0.1%. Over a five-minute time period, with N+2 messages per second, we had 30 failures with 2 nodes and a 30% loss rate, giving a false positive rate of 10%. Over a five-minute time period, with N+2 messages per second, we had 2 failures with 4 nodes and a 3% loss rate, giving a false positive rate of almost 0%, and because of the low incident rate, a very high uncertainty of the failure frequency. Over a five-minute time period, with N+2 messages per second, we had 24 failures with 4 nodes and a 10% loss rate, giving a false positive rate of 0.2%. Over a five-minute time period, with N+2 messages per second, we had 118 failures with 4 nodes and a 30% loss rate, giving a false positive rate of about 15%.

The data shown above is what was expected. A higher message loss rate yields a higher false positive rate, where the false positive rate is exponential in terms of message loss rate, but the false positive rate is significantly less than the message loss rate because we would need at least 3 consecutive messages to be dropped before we declare a failed node. We also expected a higher false positive rate with more machines, as the higher number of potential false positives increases with the number of other nodes to communicate to.