CS425 MP2

bmjain2 (Bhavana Jain) and dipayan2 (Dipayan Mukherjee)

We have implemented a distributed group membership using an extended ring topology. Whenever a node wants to join the system, it sends a request to a fixed introducer. The introducer assigns it the next available virtual ID on the ring and sends it some membership records of its monitors so it can start heart beating immediately. Then it disseminates a JOIN message through the system informing other nodes about the newly joined node. We have implemented a Finger Table for each node, where the distance between adjacent entries increases exponentially and hence our algorithm takes $O(logN)$ time to disseminate information to all nodes. Each node upon receiving a JOIN, LEAVE or CRASH message sends it to its monitors and entries in the Finger Table. When a node receives a JOIN message, it welcomes the new node to the system by sending it its member record. This way we distribute the load of populating the newly joined node's membership list to each node. Each node heartbeats to its monitors - one predecessor and two successors - after every second. This is done to ensure that the system can guarantee time-bounded completeness for upto three simultaneous failures. Each node also keeps track of its children (node ids that are supposed to send their heartbeats every second). There is a process in the background that checks the latest heartbeat timestamp every second. If it finds a child timestamp that is not updated since two heart beat periods (2 seconds), it raises suspicion and starts querying child neighbors (the ring topology is very useful here as we can calculate neighbors of the suspected child). When the child neighbor receives a suspicion message it checks if it has received a heartbeat from the suspected node in the last heart beat period. If yes, it replies saying discard your suspicion. This component helps reduce false positive rate. Meanwhile at the querying node, if the suspicion is not cleared within another second - it marks the suspected node as failed. We do not wait for the child neighbors' reply for an indefinite time - this ensures time bounded completeness. We can detect a node's failure within 4 seconds and start disseminating it. We have used the linux signal `SIGQUIT` to implement voluntary LEAVE. To leave, press `Ctrl + \` which sends a `SIGQUIT` signal. This is captured by a process and the node starts disseminating the LEAVE message. Note, whenever a node disseminates a message it marks the origin time of the message and maps it to the affected node. So if it receives a JOIN, LEAVE or CRASH message about someone (let's call it subject node), it checks what was the last message timestamp corresponding to this subject. If the retrieved timestamp is old, it will process the message and disseminate it further to its Finger Table entries. What happens if the introducer fails? When the nodes in the system receive a CRASH/LEAVE message for the introducer they start sending a periodic message containing their member record to the introducer in the background. So when the introducer joins, it reads these messages and populates its

membership list again. These messages contain a field for the maxID of the system. This is sent to ensure that it assigns a correct virtual ID to any subsequent new joinees. There is also a garbage collection that is run every 30 seconds at the introducer to collect virtual ID that are empty (holes in the ring). If there are elements in the garbage set, it can assign a virtual ID from that set.

We have marshaled the message by converting each field to a string and appending it to a message string delimited by a comma (serialization). When this message is read at the listener port, it is deserialized by splitting on the delimiter and converting each field to its corresponding type. We used the golang logging library to log events and annotated them with the node's virtual ID. Now we would run our server.go file from MP1 at each VM and query it using the client.go file using pattern=JOIN or LEAVE or CRASH to check the effect of these events on all the nodes in the system and debug them. We used the `iftop` tool to measure the bandwidth usage. To measure background bandwidth, we checked the outgoing and incoming bandwidth at the heartbeat port (only the heartbeat port = 8080 is active when there are no JOIN, LEAVE, CRASH events, for these events our system uses other port = 8081). The incoming background bandwidth is 46B/s for each child and outgoing bandwidth is 29B/s for each monitor. The average bandwidth usage for JOIN message is 93B/s, for CRASH is 88B/s and for LEAVE is 20B/s. During JOIN - member messages, changes in the monitors and children are propagated through the system. During CRASH - suspicion and status messages, changes to monitors etc are propagated. During LEAVE - only changes to monitors is propagated (hardly 3-4 nodes are affected).