MP2 Report

Our design for the failure detection is based on the premise that each node needs n+1 monitors to report failures completely. In theory, we can say that we only need n monitors since if all a processes' monitor nodes fail the monitored node is valid, but this means in the next heat of failures there's no monitors, meaning that in a practical system we should have 4 monitors per node to always have one to report. Our introducer is fixed at node 01, with a fixed hash 0. Other than this, nodes joining the cluster are hashed to some position on a ring; a node joining gets the lowest ID currently available. All nodes maintain a "neighbor" list, which is a list with the two successors and two predecessors of the node currently in circulation. We check for failures every 3 seconds; this allows plenty of leeway for avoiding false positives due to the unreliability of UDP protocol.

The way we implement this is with a bunch of threads. Every node has a **listener** to take messages and then demuxes by message to service each message accordingly. This means membership table changes and relaying are handled by offshoots of the listener. We also keep a record of recent messages to check again for repeat messages, since every node gets several copies of each (this is also due to the unreliability of UDP). The **heartbeater** and **monitor** work in conjunction to do failure detection. The monitor keeps a table of timestamps from heartbeats. Heartbeats carry these timestamps every second and update the table through the listener. The monitor checks every 3 seconds to see if the table entry matches the one from the previous check; this means there was no heartbeat in those 3 seconds, and we declare a fail. The listener services also relays information immediately, so it'll take below the 6 second deadline for the change to be reflected on all lists. We handle all 3 types of messages, with the main difference between leaves and failures being the logging. Since the leaves are sent to all monitors, the chance of being declared a fail is low; we can, however, still use the timestamps associated with each message type to resolve conflicts however. As for messages, we have attached to each a timestamp, a node hash key, a node ID (consisting of the IP and a timestamp of when it was created) and finally a message type. The message types are listed below:

HEARTBEAT: Has ID and hash of sender; respond by updating beat table

JOIN: Has ID and hash of joiner; add to list, readjust parameters, and relay to neighbors

FAIL: Has ID and hash of failed node; delete from list, readjust, relay to neighbors

LEAVE: Has ID and hash of quitting node; same as fail but log as leave

JOIN_REQ: Sent to and handled by introducer only. Contains node_ID of requester. Introducer responds with a membership table, node hash, and relays a join message.

Our algorithm is scalable to N nodes in every respect, due to the message based dissemination and on-node management reducing bandwidth over sending full membership lists. Besides, the number of nodes sent to per cycle could scale with N in order to ensure redundancy in fault tolerant design. In our case, it just happened to be 4, but based on bandwidth requirements and time to show results, as well as the importance of determinism in receiving order. We technically use deterministic Gossip protocol, which gives us a 4 nodes jump at the start, and 2 nodes per relay plus every node gets multiple copies in case of packet loss. This is relatively fast dissemination and doesn't require the bandwidth of N to N dissemination. As for marshalling, we used Go's in-built json/marshall library, meaning we transferred jsons around the cluster. The sender and receiver know the formats beforehand. The only restrictions are on maps with ints as keys, but we bypassed this with some on-node parsing.

We output to std_out for easy access for fast testing, but also to give flexibility to piping out to machine.i.log files from MP1. We used grep from MP1 to test more tedious conditions; this included timing, seeing if a fail was propagated past initial monitors, and whether requests were idempotent, as well as the membership tables or the beat tables at any point. It allowed for volume in output and saved time in filtering.

One final thing we should mention is synchronization; we synchronized all the tables with simple Go mutexes.

Measurements:

(i) the background bandwidth usage (in Bps not messages per second) for 6 machines (assuming no membership changes)

Is around 450 bytes per second. This is because our heartbeat messages are the only thing that are going, and they of size ~20B each. With 6 nodes sending 4 heartbeats every second, then each node will receive around 4 messages per second, or 80 bytes per second. Theoretically then the bandwidth should be 480 bytes per second, which is fairly close. For 2 nodes sending the bandwidth is much smaller and only around 50 bytes per second. This is because each node only sends one message to its neighbor every second, or about 40 bytes per second.

(ii) The average bandwidth usage whenever a node joins, leaves or fails (3 different numbers)

When a node leaves the network it sends out 4 message packets that are around 20B each. To each nearby node. The total bandwidth is around 600 bytes per second when a leave is issued as measured. This includes the heartbeats that are already in the background. The same is true for fail messages which got a bandwidth of around 580 bytes per second. Finally the join messages got a little higher bandwidth at 640 bytes per second because of the fact that when it is sent from the source it adds an additional node to the network that the message ripples to.