Part A

-- Parsing and initialising the structure of the network --

The graph.txt is parsed to create a ring-structure for the nodes, the *nodes* list stores this. Initially, only the neighbour ids of a node are stored. Only after successful initiation of all nodes, the respective "myNeighbours" list of each node is populated with help from the stored neighbour ids. Besides, it's made sure that the ring structure is bi-directional. Election.txt is parsed, the electionRounds HashMap (<electionRound, <nodes>>) is populated.

-- Starting elections and the procedure at each round--

The first round is initiated and incremented until it is an election round. At this point, the relevant nodes announce the start of the election. *Each node has an outgoing message queue* to ensure only one message is sent from one node to another in each round and the remaining messages are not lost and are eventually delivered in the subsequent rounds. *addMessages()* is called after each election round is announced, this method checks for any node with outgoing messages and add them to the network's *msgToDeliver* structure. Then the *deliverMessages()* method uses *msgToDeliver* to deliver the messages to each node's neighbour in the clockwise direction. Only one message is sent per round per neighbour. *Elections conclude once the electionRounds and msgToDeliver are empty.*

-- Robert Chang Algorithm and logging to the log file--

This logic is implemented in the Node class in the *receiveMessage()* method.

Part B

The fail.txt is parsed in a manner similar to election.txt and the *failureRounds* data structure is populated. For this part, additional logic is added to the *NetSimulator()* method. The simulation now only ever finishes after there's no more election rounds, no more messages to deliver and *no more failure rounds.* Whenever the current round happens to be a *failure round,* all the neighbours of the failed node are informed about the node failure and the node is removed from its neighbours' *myNeighbours* list. Then the *isNetworkConnected()* method checks if there will still be a path between the left and right neighbours of the failed node using BFS. If yes, the node is removed from the ring and the left (L) and right (R) nodes simply become each others' neighbours. Other options were explored for network reconstruction - e.g. moving around L and R in the ring, next to their neighbours (when there is no edge between L and R) but in a real system it would result in increased run time and high computational cost as every node will need to be re-placed in the ring. Therefore, simply connecting L and R seems a lot more efficient than the network re-placing every single node in the ring. If the network is not connected, the program is terminated right away.

All the other logic remains the same.