README Programming Assignment 2

Registry.go

This file represents the server side of a Peer-to-Peer (P2P) network, focusing on the registration and management of nodes, setup and maintenance of the network overlay, and handling communication and data exchange between nodes.

1. **Describe your method of allocating IDs to nodes**

```
// Generate a unique new node ID (0-127) while ensuring that no duplicate IDs are assigned
var nodeID int
for {
    nodeID = rand.Intn(128)                  // Generate a random node ID (0-127)
    if _, exists := r.Nodes[nodeID]; !exists { // Check if the generated ID is already in use by searching through the Nodes map
        break // Stop the loop if the generated ID is not in use, otherwise continue to generate a new ID
    }
}

// Add the node to the registry after passing all checks
r.Nodes[nodeID] = NodeInfo{ID: nodeID, Addr: fullAddr, Conn: conn}
r.AddrMap[fullAddr] = true // Mark the address as registered
```

This code snippet generates a random integer from 0 to 127 for node IDs using the rand.Intn method. It checks to ensure this ID is not already in use by searching the Nodes map, which maps node IDs to their corresponding 'NodeInfo', which is a data structure which contains all the fields necessary to define a node. If the ID is already in use, the method continues to generate a new ID until a unique one is found. Once that ID is generated, it is assigned to the new node and recorded in the registry's data structure.

2. **Describe the method you have implemented to avoid partitions**

To ensure a consistent ordering of nodes in the overlay network, node IDs are sorted in ascending order in a list representation, *L*, before registering the routing tables for each of them. This helps achieve consistency in the list representations amongst all nodes.

Each node's routing table is constructed to include nodes at progressively doubling distances in the ID space based on the hop formula $(p + 2^{i-1}) \, mod \, |L|$ where *p* is the current index of the node in list L, |L| is the number of nodes in the list *L* and *i* is the index in its routing table. The result of the hop formula yields the index of the next node with hop *i*.

The hop formula ensures that the routing table always stores the index of the subsequent node with respect to the current node *(when i is 1)*. Hence, by representing the overlay network as a graph with nodes being the messaging nodes and the edges

being the connection between each node to every node in its routing table, there exists a directed cycle in the graph. This implies that there will always be a path connecting one node to another in the overlay network, and therefore there exist no partitions in the overlay network.

3. Explain why your solution does not have deadlocks

Mutex locks were used in the following methods handleRegister, handleDeregister, handleNodeRegistryResponse, startMessaging, handleTaskFinish, handleTrafficSummary, listNodes, listRoutes, SetupOverlay. This is because these methods alter the state of the registry and there exists a read/write access to shared attributes in the registry.

The above methods do not depend on any conditions that would never be False and it only sends asynchronous requests to the clients and hence never waits for a response from the clients. Therefore, it will all eventually release the mutex lock, hence there exist no hold and wait of any shared resources.

In handleNodeRegistryResponse, there exist a condition in line 450 as shown in the figure below.

```
446     allSetupComplete := true
447     // Check if all nodes have finished setup successfully
448     for id := range r.Nodes {
449         setupComplete := r.NodesSetupFinished[id]
450         if !setupComplete {
451             allSetupComplete = false
452             break
453         }
454     }
```

Lines 448 to 454 will always complete its execution eventually as there are finite numbers of nodes in the network overlay. In line 450, the condition will either be True or False. If it is True, handleNodeRegistryResponse will continue its execution at line 454, breaking out of the for loop. If it is False, the execution will just move on to the next iteration of the for-loop. Both will eventually lead to the complete execution of handleNodeRegistryResponse, and hence the release of the mutex lock.

As such, deadlock is prevented in our solution.

Messenger.go

This file implements a messaging node for a P2P network which include functionalities like setting up the node, registering with the registry, handling different message types and communicating with other nodes.

1. **Explain why your methods deliver data packets. How do you avoid those packets travelling forever?**

Once the setup command is issued in the registry, the setupOverlay function is called with the specified number of routing entries. If there are none specified, it defaults to three. The function then proceeds to create and assign routing tables for each node based on the hop formula. Once each node's routing table is calculated, the sendNodeRegistry function is called. This function establishes a TCP connection to each node, and sends a message containing the newly constructed routing table. After all nodes have received their routing tables, a message indicating the completion of the overlay setup is printed (line 614). This marks the end of the setup process, and the nodes are now ready for the InitiateTask message. Once the start command is issued, the registry checks if all nodes have completed their overlay setup. If all nodes have finished setup, the StartMessaging function is invoked. This function sends a InitiateTask message to all nodes in the overlay. Once a node receives the InitiateTask message, a separate goroutine is invoked to handle this message (line 279). By creating a separate goroutine to handle the InitiateTask message, it ensures that the node's main execution thread remains responsive. Consider a node tasked with sending out a large number of messages, by creating a new goroutine to handle this message sending, the node can continue to update its state and receive new commands all while the messaging task proceeds in the background. The creation of a separate goroutine allows for a more responsive node. Once the node receives the InitiateTask message, the function handleInitiateTask is called where it randomly selects a destination node and sends a message to that node. This is seen in line 458 in messenger.go. This function then constructs and sends a data packet to the randomly chosen destination node through a function called sendDataPacket.

```
508     // Find the next hop to the destination
509     nextHop, exists := routingTable.Entries[destNodeID]
510     if !exists {
511         log.Printf("No next hop found for destination node %d\n", destNodeID)
512         return
513     }
```

In this code snippet, the function looks up the next node to which the message should be sent based on the destination node's ID. It checks if the destination node ID is in the routing table, and if the ID does not exist, it exists the function. Else if the destination ID exists in the routing table, it calls the function forwardMessage to attempt to forward the message to the next hop. In the forwardMessage function (line 411), it checks if the current node that wants to forward the message is already in the trace of the nodeData message. The `trace` field of nodeData contains the IDs of all nodes the message has traversed, as specified in the NodeData format. If the current node's ID is found in this trace, it would signify that there is a routing loop, and this function logs an error message and returns without forwarding the message, thereby preventing infinite message circulation. When a node receives a NodeData message, the processNodeDataMessage function (line 363) is called to handle this message type. This function checks if the node ID of the node that receives this message is the destination ID of this message, if it is, it means that the message has reached its destination. It then increments the various global counters like receiveTracker and receiveSummation. If the message hasn't arrived at its destination, it uses the calculateDistance function to determine the distance in terms of the number of hops to the destination node. closestNextHop is initially set to 1 to indicate that no suitable next hop has been found yet. The closestDistance variable is then initialized with the value of destinationDistance. This starts the search with the assumption that the current node is the closest to the destination. The function then iterates through the routing table to find the closest next hop towards the destination. For each node in the routing table, it uses the calculateDistance function to calculate how far the current node is from the message's intended destination node. It then checks if this distance is less than the current closestDistance, if it is, this means that the current node is closer to the destination than any other node evaluated before. Hence, we update the closestNextHop to the current node's ID, indicating that so far, this node is the best candidate for the next hop. The variable closestDistance is then updated to nextHopDistance, reflecting that the distance to the destination through this node is the shortest found so far. The nextHop variable is then set to the address of the current node. By evaluating each potential next hop, the node ensures that the message is forwarded in the direction that moves it closer to its destination.

```go
func calculateDistance(source, destination int) int {
    const IDSpaceSize = 128
    distance := (destination - source + IDSpaceSize) % IDSpaceSize
    return distance
}
```

The hop algorithm which uses the formula $(p + 2^{i-1}) \, mod \, |L|$ ensures that we will never overshoot the destination node as we always jump a distance that is smaller than the distance to the destination node. This ensures that the packets will never travel forever as the packets can only approach closer to the destination node with each hop.

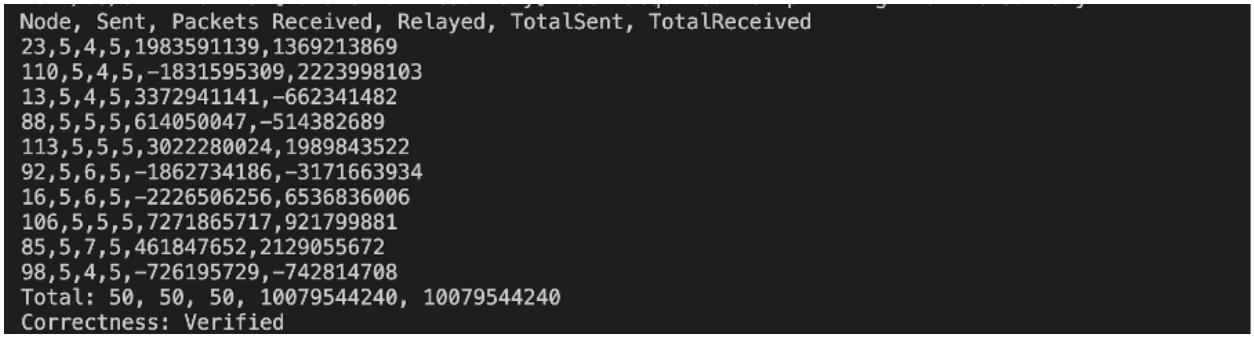
2. **Explain how you avoid duplication in routing the packets**

```go
// This function forwards the given message to the next hop according to the routing table
func forwardMessage(nodeData *minichord.NodeData, nextHop string) error {

    // Check if this node is already in the trace to prevent loops
    currentNodeID := int32(myNodeID)
    for _, id := range nodeData.Trace {
        if id == currentNodeID {
            log.Printf("Loop detected in trace: %v\n, dropping the message to avoid duplicates", nodeData.Trace)
            return nil
        }
    }

    // Add the current node to the trace before forwarding
    nodeData.Trace = append(nodeData.Trace, currentNodeID)
    // Increment the hop count
    nodeData.Hops++
```

Each data packet maintains a trace of nodeIds it has passed through. When a node receives a packet, this function checks if the node is already in the trace. If a node receives a message that includes its ID in the trace, it will drop the message and the loop will break. If not, it will add its ID to the trace and then proceeds to forward the packet to the next node. This ensures that future nodes in the path can also check against duplication and avoid sending the packet back to this node.

3. **Explain why your task completion and retrieval mechanism is working correctly**

Client nodes signal task completion to the registry by sending a TaskFinished message using the method sendTaskFinishedMessage(). The registry, upon receiving a message of type TaskFinished, would set a new entry into the NodesTaskFinished map to track which clients have completed their assigned tasks. Upon all clients completing their tasks, that is, the length of handleTaskFinished is the same as the number of clients in the overlay network, the registry would request the clients to send their traffic summaries to the registry.



The above figure illustrates an example output with 10 clients initiating communications with each other and with the registry.

The total number of packets sent equals the total number of packets received in the overlay network (50 total sent packets are received), which demonstrates that all packets sent have been accounted for in the network. Moreover, the TotalSent and TotalReceived values also tallies, which suggests that the contents of the packets are not corrupted/altered during the transmission.