Programming Assignment 2

Tómas Magnússon & Marcel Kyas

Version 1.1.1, 07 February 2023

This assignment accounts for 20 % of your final grade. We have estimated that typical students in a team of two should each allocate 18 h to work together to complete the assignment.

This assignment is based on Shrideep Pallickara's assignment at Colorado State University.[1]

Objectives and Overview

The objective of this assignment is to get you familiar with peer-to-peer networks, *overlay* networks, and communication between nodes. You should have a reusable components that may be useful in Programming Assignment 3.

Some Context

You are building a simplified version of a structured P2P system based on distributed hash tables (DHTs); the routing here is a simplified implementation of the well-known Chord P2P system (see Stoica et al. (2003) and van Steen and Tanenbaum (2017, Note 2.5, pp. 83f.)). In most DHTs node identifiers are 128-bits (when they are based on UUIDs) or 160-bits (when they are generated using SHA-1). In such systems the identifier space ranges from 0 to 2^{128} or 2^{160}. Structured P2P systems are important because they have demonstrably superior scaling properties.

As part of this assignment you be implementing routing schemes for packets in a structured peer-to-peer (P2P) overlay system. This assignment requires you to:

- construct a logical overlay over a distributed set of nodes, and then
- use partial information about nodes within the overlay to route packets.

[1] You can access his version here: `https://www.cs.colostate.edu/~cs455/CS455-Spring20-HW1-PC.pdf`.

The assignment demonstrates how partial information about nodes comprising a distributed system can be used to route packets while ensuring correctness and convergence.

Nodes within the system are organized into an overlay i.e. you impose a logical structure on the nodes. The overlay encompasses how nodes are organized, how they are located, and how information is maintained at each node. The logical overlay helps with locating nodes and routing content efficiently.

The overlay will contain at least 10 messaging nodes, and each messaging node will be connected to some other messaging nodes.

Once the overlay has been setup, messaging nodes in the system will select a node at random and send a message to that node (also known as the sink or destination node). Rather than send this message directly to the sink node, the source node will use the overlay for communications. Each node consults its routing table, and either routes the packet to its final destination or forwards it to an intermediate node closest (in the node ID space) to the final destination. Depending on the overlay, there may be zero or more intermediate messaging nodes between a particular source and sink that packets must pass through. Such intermediate nodes are said to relay the message. The assignment requires you to *demonstrate* correctness of packet exchanges between the source and sink by ensuring that:

- the number of messages that you send and receive within the system match, and
- these messages have not been corrupted in transit to the intended recipient.

Message exchanges happen continually in the system.

The assignment must be implemented in the *Go programming language.* You must develop all functionality yourself. You are not permitted to import packages outside of the go-distribution.[2] You may use code examples from this document or the lecture to implement your functions.

All communications in this assignment shall use *TCP* as transport method. The messages use Google protobuf format. We provide a format definition from which you can generate marshal/unmarshal code. You *must not* change the message format.[3]

[2]If you are not sure if a package may be used, do not hesitate to ask.

[3]Protobuf and its interface to Go are described at:

- `https://developers.google.com/protocol-buffers/docs/proto3`,
- `https://developers.google.com/protocol-buffers/docs/gotutorial`
- `https://developers.google.com/protocol-buffers/docs/reference/go-generated`,
- `https://pkg.go.dev/google.golang.org/protobuf/proto`, and
- `https://developers.google.com/protocol-buffers/docs/reference/go/faq`.

This assignment may be modified to clarify any questions (and the version number incremented), but the crux of the assignment and the distribution of points (see Section 6) will not change.

1 Components

There are two components that you will be building as part of this assignment: a registry and a messaging node. There is exactly one instance of the registry and multiple instances of the messaging nodes.

1.1 Registry

There is exactly one registry in the system. The registry provides the following functions:

- Allows messaging nodes to register themselves. This is performed when a messaging node starts up for the first time.
- Assign random identifiers (between 0-127) to nodes within the system; the registry also has to ensure that two nodes are not assigned the same IDs i.e., there should be no collisions in the ID space.
- Allows messaging nodes to deregister themselves. This is performed when a messaging node leaves the overlay.
- Enables the construction of the overlay by populating the routing table at the messaging nodes.

The routing table dictates the connections that a messaging node initiates with other messaging nodes in the system. The registry maintains information about the registered messaging nodes; you can use any data structure for managing this information but make sure that your choice can support all the operations that you will need. The registry does not play any role in the routing of data within the overlay. Interactions between the messaging nodes and the registry are via request-response messages. For each request that it receives from the messaging nodes, the registry will send a response back to the messaging node (based on the IP address associated with the socket's input stream) where the request originated. The contents of this response depend on the type of the request and the outcome of processing this request.

1.2 The messaging node

Unlike the registry, there are multiple messaging nodes (minimum of 10) in the system. A messaging node provides two closely related functions: it initiates and accepts both communications and messages within the system. Communications that nodes have with each other are based on TCP. Each messaging node needs to automatically configure the port over which it listens for communications i.e. the server-socket port numbers shall not be hard-coded or specified at the command line.[4]

Use `net.Listen` to accept incoming TCP communications. Once the initialization is complete, the node should send a registration request to the registry. Each node in the system has a routing table that is used to route content along to the sink. This routing table contains information about a small subset of nodes in the system. Nodes should use this routing table to forward packets to the sink specified in the message. Every node makes local decisions based on its routing table to get the packets closer to the sink. Care must be taken to ensure you don't change directions or overshoot the sink: in such a case, packets may continually traverse the overlay.

2 Interactions between the components

This section will describe the interactions between the registry and the messaging nodes. The message format is specified using Google's protobuf format. The full specification is provided in a file called `minichord.proto`. The meaning of each message and field is explained in this section.

You can compile protobuf specifications into code in different programming languages. This code implements the necessary functions to convert a native data structure into the protobuf wire format and to parse a message and convert it to a language native data structure. We provide the corresponding go file in `minichord.pb.go`.

In principle, you marshall such a message using the subsequent piece of code:

```go
// import the proto package
import "google.golang.org/protobuf/proto"
import minichord


// Marshal end send message
data, err := proto.Marshal(message)
if err != nil {
    // handle error
}
_, err := conn.Write(data)
```

[4]Check the `"math/rand"` for random number generators. Can you find out if another process listens to messages on a TCP port? It is fine to set a port on the command line for debugging purposes.

```
if err != nil {
    // handle error
}
```

and unmarshal it with the following piece of code:

```
data := make([]byte, 65535)
length, err := conn.Read(data)
if err != nil {
    // handle error
}
message := &minichord.MiniChord{}
err := proto.Unmarshal(data[:length], message)
if err != nil {
    // handle error
}
```

All messages are a `message MiniChord`, which is defined to be one of the following messages.

```
message MiniChord {
        oneof Message {
                Registration registration  = 17;
                RegistrationResponse registrationRespone = 18;
                Deregistration deregistration = 19;
                DeregistrationResponse deregistrationResponse = 20;
                NodeRegistry nodeRegistry = 21;
                NodeRegistryResponse nodeRegistryResponse = 22;
                InitiateTask initiateTask = 23;
                NodeData nodeData = 15;
                TaskFinished taskFinished = 24;
                RequestTrafficSummary requestTrafficSummary = 25;
                TrafficSummary reportTrafficSummary = 26;
        }
}
```

2.1 Registration

Upon starting up, each messaging node shall register its IP address, and port number with the registry. It should be possible to register messaging nodes that are running on the same host but are listening on different ports. There shall be one fieldin this registration request, as displayed in Table 1: When a

Address	A `string` holding the clients address in a format accepted by `net.Dial` (typically, `"host:port"`).

Table 1: `message Registration` format

registry receives this request, it checks to see if the node had previously registered and ensures that the IP address in the message matches the address where the request originated. The registry issues an error message under two circumstances:

- If the node had previously registered and has a valid entry in its registry.
- If there is a mismatch in the address that is specified in the registration request and the IP address of the request (the socket's input stream).

If there is no error, the registry generates a unique identifier (between 0-127) for the node while ensuring that there are no duplicate IDs being assigned.

The contents of the response message generated by the registry are depicted in Table 2. The success or failure of the registration request should be indicated in the status field of the response message.

Result	A `sfixed32` signed integer describing the status. If registration is successful, this number is the node's ID. A negative number indicates failure.
Info	A `string` that describes the error or provides other information.

Table 2: `message RegistrationResponse` format

In the case of successful registration, the registry should include a message that indicates the number of entries currently present in its registry. A sample information string is 'Registration request successful. The number of messaging nodes currently constituting the overlay is (5).' If the registration was unsuccessful, the message from the registry should indicate why the request was unsuccessful.

NOTE: In the rare case that a messaging node fails just after it sends a registration request, the registry will not be able to communicate with it. In this case, the entry for the messaging node should be removed from the data structure maintained at the registry.

2.2 Deregistration

When a messaging node exits it should deregister itself. It does so by sending a Deregistration message to the registry. This deregistration request includes the fields in Table 3.

The registry should check to see that request is a valid one by checking

1. where the message originated, and
2. whether this node was previously registered

Error messages should be returned in case of a mismatch in the addresses or if the messaging node is not registered with the overlay. You should be able to test the error-reporting functionality by de-registering the same messaging node twice. The registry will respond with a `message DeregistrationResponse` that is similar to the `message RegistrationResponse` message in Table 2.

2.3 Peer node manifest

Once the `setup` command (see Section 3.2) is issued at the registry it must perform a series of actions that lead to the creation of the overlay with:

1. a routing table being installed at every node, and

2. messaging nodes initiating connections with each other

Messaging nodes await instructions from the registry regarding the messaging nodes that they must establish connections to. Messaging nodes only initiate connections to nodes that are part of its routing table.

The registry is responsible for populating state information at nodes within the system. It does so by propagating state information: this is used to populate the routing table (both the node IDs and the corresponding logical addresses) at individual nodes and also to inform nodes about other nodes (only the node IDs) in the system.

The registry must ensure two properties. First, it must ensure that the size of the routing table at every messaging node in the overlay is identical; this is a configurable metric (with a default value of 3) and is specified as part of the `setup` command. Second, the registry must ensure that there is *no partition* within the overlay i.e. it should be possible to reach any messaging node from any other messaging node in the overlay.

If the routing table size requirement for the overlay is N_R, each messaging node will have links to N_R other messaging nodes in the overlay. The registry selects these N_R messaging nodes that constitute the peer-messaging nodes list for a messaging node such that the first entry is one hop away in the `ID` space, the second entry is two hops away, and the third entry is 4 hops away.

Consider a network overlay comprising nodes with the following identifiers: 10, 21, 32, 43, 54, 61, 77, 87, 99, 101, 103. The routing table at 10 includes information about nodes <21, 32, and 54> while the routing table at node 101

Id	A `sfixed32` signed integer that holds the nodes ID. The value must be non-negative.
Address	A `string` holding the clients address in a format accepted by `net.Dial`.

Table 3: `message Deregistration` format

includes information about nodes <103, 10, 32>; notice how the ID space wraps around after 103. A messaging node should initiate connections to all nodes that are part of its routing table. A check should be performed to ensure that the list does not include the targeted messaging node i.e. a messaging node should not have to connect to itself.

The registry also informs each node about the IDs (it should not include IP addresses) of all nodes in the system. This information is used in the testing part of the overlay to randomly select sink nodes that the messages should be sent to.

The registry includes all this information in a `message NodeRegistry`. The contents of the message are different for each messaging node (since the routing table at every messaging node would be different).

The message format is described in Table 4.[5]

NR	A `fixed32` integer defining the number of entries in the routing table.
Peers	A `repeated Deregistration` that maps a node Id to its address, a string suitable for `net.Dial`.
NoIds	A `fixed32` integer defining the number peers in the network.
Ids	A `repeated sfixed32` list of all node ids without their IP addresses.

Table 4: `message NodeRegistry` format

Note that the node registry message includes IP addresses only for nodes within a particular node's routing table. Upon receipt of the manifest from the registry, each messaging node should initiate connections to the nodes that comprise its routing table.

2.4 Node overlay setup

Upon receipt of the `message NodeRegistry` from the registry, each messaging node should initiate connections to the nodes that comprise its routing table. Every messaging node must report to the registry on the status of setting up connections to nodes that are part of its routing table.

The format of the `message NodeRegistryResponse` is similar to the previous response messages in Table 2.

[5]A previous version used a map of node id to addresses. Maps do not preserve iteration order. While it could be reconstructed from the Ids field, we instead use a list of Deregistration messages, which are pairs of the Ids and Addresses. It would have been cleaner to introduce a new message type, however.

Packets	A `fixed32` integer that defines the number of packages to be sent.

Table 5: `message InitiateTask` format

2.5 Initiate sending messages

The registry informs nodes in the overlay when they should start sending messages to each other. It does so with sending a `message InitiateTask` control message. This message also includes the number of packets that must be sent by each messaging node. See Table 5 for a meaning of the message.

2.6 Send data packets

Data packets can be fed into the overlay from any messaging node within the system. Packets are sent from a source to a sink; it is possible that there might be zero or more intermediate nodes in the system that *relay* packets en route to the sink. Every node tracks the number of messages that it has relayed during communications within the overlay.

When a packet is ready to be sent from a source to the sink, the source node consults its routing table to identify the best node that it should send the packet to. There are two situations:

1. there is an entry for the sink in the routing table, or
2. the sink does not exist in the routing table and the messaging node must relay the packet to the closest node.

During routing, care must be taken to ensure that you don't change directionality i.e. your routing decisions should target only nodes that are clockwise successors. You must also ensure that you do not overshoot the sink-node you are trying to reach. Routing errors will result in a packet continuously looping through the overlay and consuming bandwidth.

A key requirement for the dissemination of packets within the overlay is that no messaging node should receive the same packet more than once. This should be achieved without having to rely on duplicate detection and suppression. The message format is described in Table 6.

The dissemination trace includes nodes (except the source and sink) that were involved in routing the particular packet. The dissemination traces will help you in your debugging and help you identify any bugs in your implementation.

2.7 Inform registry of task completion

Once a node has finished sending its number of messages (described in Section 4), it informs the registry of its task completion using the `message TaskFinished`.

This message has the format described in Table 7.

2.8 Retrieve traffic summaries from nodes

Once the registry has received a `message TaskFinished` message from all the registered nodes it will issue a `message RequestTrafficSummary`. This message is sent to all the registered nodes in the overlay. This message has no fields.

2.9 Sending traffic summaries from the nodes to the registry

Upon receipt of the `message RequestTrafficSummary` from the registry, the messaging node shall create a response that includes a summary of the traffic that it has participated in. The summary shall include information about messages that were sent, received, and relayed by the node. This message has the format in Table 8.

Once the `message TrafficSummary` message is sent to the registry, the node must reset the counters associated with traffic relating to the messages it has sent, relayed, and received so far: the number of messages sent, summation of sent messages, etcetera.

3 Interacting with processes

Both the registry and the messages should run as foreground processes. Support for the following commands helps in debugging the programs. The commands

Destination	A `sfixed32` designating the ID of the destination node
Source	A `sfixed32` designating the ID of the source node
Payload	A random `sfixed32` serving as our payload
Hops	A `fixed32` integer defining the number of hops this message has taken.
Trace	A `repeated sfixed32` list of nodes this message has been forwarded by.

Table 6: `message NodeData` format

Id	A `sfixed32` signed integer defining the ID of the node.
Address	A `string` holding the clients address in a format accepted by `net.Dial`.

Table 7: `message TaskFinished` format

Id	A `sfixed32` designating the node's ID
Sent	A `fixed32` designating the total number of packets sent (only the ones that were started/initiated by the node)
Received	A `fixed32` designating the total number of packets received (packets with this node as final destination)
Relayed	A `fixed32` designating the total number of packets relayed (received from a different node and forwarded)
TotalSent	A `sfixed64` designating the sum of packet data sent (only the ones that were started by the node)
TotalReceived	A `sfixed64` desginating the sum of packet data received (only packets that had this node as final destination)

Table 8: `message TrafficSummary` Format

to be supported are specific to the two components.

3.1 Registry

list List the hostname, port number, and node id of all currently registered messaging nodes.

setup *n* Setup the overlay with *n* entries.

route List the computed routing tables for each node in the overlay.

start *n* The start command makes the registry send the `message TaskInitiate` message to all nodes registered in the overlay. A command of `start 50` results in each messaging node sending 50 packets to nodes chosen at random. A node must not send a packet to itself.

3.2 Messaging Node

print This should print information to the console about the number of messages that have been sent, received, and relayed along with the sums for the messages that have been sent from and received at the node.

exit This allows a messaging node to exit the overlay. The messaging node should first send a deregistration message (see Section 2.2) to the registry and await a response before exiting and terminating the process.

3.3 Implementation Sketch

A sketch of such a command interpreter is shown in Listing 1. The `strings` package provides many functions for simple string analysis.

4 Setting

For the remainder of the discussion we assume that the `setup` command has been specified. Also, nodes will not be added to the system from hereon. Any errors during the overlay setup should be reported back to the registry.

The `start` command can only be issued after all nodes in the overlay report success in establishing connections to nodes that comprise its routing table. This is reported in the `message NodeRegistryResponse`. Only after all nodes report success in setting up connections should the registry print

```
Registry now ready to initiate tasks.
```

to the console.

When the `start` command is specified at the registry, the registry sends the `message InitiateTask` control message to all the registered nodes within the overlay. Upon receiving this message from the registry, a given node will start exchanging messages with other nodes.

Each node participates in a set of rounds. Each round involves a node sending a packet to a randomly chosen node (excluding itself, of course) from the set of all registered nodes advertised in the `message NodeRegistry`. All communications in the system will use TCP. To send a data packet the source node consults its routing table to make decisions about the link to send the packet over. During a packet's routing from the source to the sink there might be zero or more intermediate nodes relaying the packet en route to the destination sink node. The payload of each data packet is a random integer with values that range from 2147483647 to -2147483648. During each round, 1 packet is sent. At the end of each round, the process is repeated by choosing another node at random. The number of rounds that each node will participate in is specified in the `message InitiateTask` command. During grading this value will be set anywhere between 25,000 and 100,000 messages.

The number of nodes will be fixed at the start of the experiment. We will likely use around 10 nodes for the test environment during grading.

4.1 Tracking communications between nodes

Each node will maintain two integer variables that are initialized to zero: sendTracker and receiveTracker. The sendTracker represents the number of data packets that were sent by that node and the receiveTracker maintains information about the number of packets that were received. Additionally, each node will track the number of packets that it relayed i.e., packets for which it was neither the

```go
package main

import (
	"bufio"
	"os"
	"strings"
	"fmt"
)

func main() {
	reader := bufio.NewReader(os.Stdin)
	repl: for {
		cmd, err := reader.ReadString('\n')
		if err != nil {
				fmt.Println(err)
				break
		}
		cmd = strings.Trim(cmd, "\n")
		switch (cmd) {
		case "print":
				fmt.Println("print")
		case "exit":
				fmt.Println("Bye.")
				break repl
		default:
				fmt.Printf("command not understood: %s\n", cmd)
		}
	}
}
```

Listing 1: Interactive loop

source nor the sink. Consider the case where there are 10 nodes in the system and every node sends 25,000 packets. With 10 nodes in the system, the total number of data packets would be 250,000. Since a sending node chooses the target node at random, the number of packets received by different receivers would be different.

The number of packets that a node relays will depend on the overlay topology and the routing table supplied at each messaging node. The number of packets is tracked using the variable relayTracker.

To track the data packets that it has sent and received, each node will maintain two additional long variables that are initialized to zero: sendSummation and receiveSummation. The data type for these variables is a long to cope with overflow issues that will arise as part of the summing operations that will be performed. The variable sendSummation, continuously sums the values of the random numbers that are sent, while the receiveSummation sums values of the payloads that are received. The values of sendSummation and receiveSummation at a node can be positive or negative.

4.2 Correctness Verification

We will verify correctness by:

1. checking the number of messages that were sent and received, and
2. if these packets were corrupted for some reason.

The total number of messages that were sent and received by the set of all nodes must match i.e. the cumulative sum of the receiveTracker at each node must match the cumulative sum of the sendTracker variable at each node. We will check that these packets were not corrupted by verifying that: when we add up the values of sendSummation it will exactly match the added up values of receiveSummation.

4.3 Collecting and printing outputs

When a messaging node completes sending the required number of packets, it sends a `message TaskFinished` to the registry. When the registry receives these from each registered node in the system, it sends a `message RequestTrafficSummary` to all the messaging nodes.

Upon receipt of this message, a node will prepare to send information about the data packets that it has sent and received. This includes:

1. the number of packets that were sent by that node,
2. the summation of the sent packets,
3. the number of packets that were received by that node, and

	Packets			Sum Values	
	Sent	Received	Relayed	Sent	Received
Node 1	25,000	25,095	67,375	93,621,870,668	-12,128,394,816
Node 2	25,000	25,017	67,296	-246,903,482,532	108,212,257,345
Node 3	25,000	24,798	67,360	-335,356,792,441	65,776,216,500
Node 4	25,000	25,051	67,372	89,016,048,382	260,492,050,045
Node 5	25,000	24,912	67,439	-150,372,901,113	23,093,507,630
Node 6	25,000	24,762	67,586	61,645,416,274	-412,104,380,095
Node 7	25,000	24,808	67,668	94,867,862,942	-109,941,024,057
Node 8	25,000	24,954	67,657	34,693,116,464	53,878,071,043
Node 9	25,000	25,293	67,421	474,225,881,302	-262,102,181,715
Node 10	25,000	24,826	67,482	-190,281,122,109	84,925,294,329
Node 11	25,000	25,038	67,576	-212,203,755,192	70,357,084,958
Node 12	25,000	25,303	67,505	509,693,519,689	6,649,837,422
Node 13	25,000	24,963	67,398	10,153,753,796	122,025,341,787
Node 14	25,000	25,015	67,636	-79,055,293,468	-57,687,910,765
Node 15	25,000	25,165	67,515	-244,882,579,433	-32,584,226,382
Sum	375,000	375,000	1,012,286	-91,138,456,771	-91,138,456,771

Table 9: Collated Outputs

4. the summation of the received packets.

The node packages this information in the `message TrafficSummary` and sends it to the registry. After a node generates this message, it shall reset the counters that it maintains. This will allow testing the software for multiple runs.

Example output at the registry Upon receipt of the `message TrafficSummary` from all the registered nodes, the registry will proceed to print out the table in a comma-separated value format: Print the report for each node on a single line. Separate entries with commas. You should not print thousands separators.

The collated outputs from 15 nodes are depicted in Table 9[6]. Note how the number of received messages may be slightly different than the number of sent messages at each node. The sum of sent or received values at a node may be negative. In this particular example the final summation across all nodes is negative, it may well be positive and that is fine!

5 Command line arguments for the two components

Your code should be organized in a directory called "group-n", where n is the number of your submission group. The command line arguments for the messaging nodes and registry are listed below:

[6]CSV files can easily be rendered as tables, e.g. by importing them to Excel or using a LaTeXpackage like csvsimple

```
go run registry.go registry-port

go run messenger.go registry-host:registry-port
```

Note that we want to be able to run the system on the same host (localhost) and on different hosts.

6 Grading

This assignment counts for 20 % towards your final grade. The programming component accounts for 60 % with the explanation accounting 40 %. We will award 30 points in total, 15 points for the registry and 15 points for the messaging node.

Table 10: Breakdown of registry

Points	Feature
2	The registry is functional with support for registration and de-registration of nodes.
3	Setting up of the overlay with the correct routing tables while ensuring that there are no network partitions, i.e. all nodes can reach each other.
2	Successful retrieval of task summaries from the messaging nodes.
2	Traffic summaries are collated and printed out as depicted in the example table.
1	Describe your method of allocating Ids to nodes.
3	Describe the method you have implemented to avoid partitions.
2	Explain why your solution does not have deadlocks.

6.1 Your own work

Plagiarism results in 0 points for this assignment. See RU Library site about Plagiarism, RU Project Code of Conduct and Section 7: Procedures and appeals of RU's General rules on study and assessment for more information.

6.2 Deduction

There will be a 100 % penalty if any of the following restrictions are violated:

1. You specify your own message or wire format. You must use the pre-defined one.

2. If you use RPC or any other middleware to implement this assignment.
3. There is a spirit to this assignment that you must preserve. If your messaging nodes use any information other than the routing tables to route content (by creating custom messages between nodes) this deduction will apply. We will be using wireshark to ensure that your nodes are not circumventing the specified protocols.
4. If you import packages that are not permitted.
5. If you exchange code with your peers *without acknowledgement*.
6. If you use code from websites without citation and attribution. Exception: code examples from the go tutorial, this text, and the lectures are fine to use.

7 Milestones

You have three weeks to complete this assignment.[7] The milestones below correspond to what you should be able to complete after about 6 to 8 hours.

Milestone 1 You should be able to have two nodes exchanging messages as specified by the wire format. (Estimated work time: 8 h for the whole group)

[7]Shrideep Pallickara allows four weeks, but his students have to implement the Solution in Java and must implement a binary message format. You enjoy the luxury of writing the code in Go and do not have to implement (and debug!) the wire format.

Table 11: Breakdown of messaging node

Points	Feature
2	Establishes connection based on `message NodeRegistry`
3	Routes data packets successfully within the overlay without duplication and complete reachability
1	The mechanism for task completion and retrieval of summaries works correctly
3	Message totals of send and receive match. This includes bot the message count and the content summation counts.
2	Explain why your methods delivers data packets. How do you avoid that packets travel forever?
2	Explain how you avoid duplications in routing the packets.
2	Explain why your mechanism for task completion and retrieval are working correctly.

Milestone 2 You should be able to have 10 messaging node instances talk to the registry, and have the registry sending commands to orchestrate the setting up of the overlay. You should also be able to issue all commands at the foreground processes. (Estimated work time: 12 h for the whole group)

Milestone 3 You should be able to use the routing tables at each messaging node to send and relay messages fed into the overlay. You should be able to track the summation counts for the messages and the contents of these messages. (Estimated work time: 6 h for the whole group)

Milestone 4 Iron out any wrinkles that may preclude you from getting the correct (i.e. not corrupted) outputs at all times. (Estimated work time: 4 h for the whole group)

8 Submission

Your submission should be a single tar file that contains:

- All go files related to the assignment
- A *bash* script that sets up the network of nodes for testing
- A README file containing a description of each file and any information you feel the TA needs to grade your assignment.

Your reasoning can be part of the README file. It can be in a pdf file inside the tar file. It can also be in source code comments. It can be any combination of these. Please clearly specify where we find your documentation in the README file.

References

Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, February 2003. doi:10.1109/TNET.2002.808407.

Marten van Steen and Andrew S. Tanenbaum. *Distributed Systems*. CreateSpace Independent Publishing Platform, 3.01 edition, February 2017. URL `https://www.distributed-systems.net/`.