Homework 1: Programming Component

Using Dijkstra's Shortest Paths to Route Packets in a Network Overlay

VERSION 1.1

DUE DATE: Wednesday February 14th, 2024 @ 8:00 pm

The objective of this assignment is to get you familiar with coding in a distributed setting where you need to manage the underlying communications between nodes. Upon completion of this assignment you will have a set of reusable classes that you will be able to draw upon. As part of this assignment you will be: (1) constructing a logical overlay over a distributed set of nodes, and then (2) computing shortest paths using Dijkstra's algorithm to route packets in the system.

The overlay will contain at least 10 messaging nodes, and each messaging node will be connected to $\mathbf{C_R}$ (default of 4) other messaging nodes. Each link that connects two messaging nodes within the overlay has a weight associated with it. Links are **bidirectional** *i.e.* if messaging node **A** established a connection to messaging node **B**, then messaging node **B** must use that link to communicate with **A**.

Once the overlay has been setup, messaging nodes in the system will select a node at random and send that node (also known as the sink node) a message. Rather than send this message directly to the sink node, the source node will use the overlay for communications. This is done by computing the shortest route (based on the weights assigned during overlay construction) between the source node and the sink node. Depending on the overlay and link weights, there may be zero or more intermediate messaging nodes that packets between a particular source and sink must pass through. Such intermediate nodes are said to relay the packets. The assignment requires you to verify **correctness** of packet exchanges between the source and sinks by ensuring that: (1) the number of messages that you send and receive within the system match, and (2) these messages have not been corrupted in transit to the intended recipient. Message exchanges and connection setups/terminations happen continuously in the system.

All communications in this assignment are based on **TCP**. The assignment must be implemented in **Java** and you cannot use any external jar files. You must develop all functionality yourself. 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 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.

1.1 Registry:

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

A. Allows messaging nodes to register themselves. This is performed when a messaging node starts up for the first time.
B. Allows messaging nodes to deregister themselves. This is performed when a messaging node leaves the overlay.
C. Enables the construction of the overlay by orchestrating connections that a messaging node initiates with other messaging nodes in the system. Based on its knowledge of the messaging nodes (through function A) the registry informs messaging nodes about the other messaging nodes that they should connect to.
D. Assign and publish weights to the links connecting any two messaging nodes in the overlay. The weights these links take will range from 1-10.

The registry maintains information about the registered messaging nodes in a registry; you can use any data structure for managing this registry 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 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 ports over which it listens for communications i.e. the port numbers should not be hard-coded or specified at the command line. `TCPServerSocket` is used to accept incoming TCP communications.

Once the initialization is complete, the node should send a registration request to the registry.

2 Interactions between the components

This section will describe the interactions between the registry and the messaging nodes. This section includes the prescribed wire-formats. You have freedom to construct your wire-formats but you must include the fields that have been specified. A good programming practice is to have a separate class for each message type so that you can isolate faults better. The `Message Type`s that have been specified could be part of an interface, say `csx55.overlay.wireformats.Protocol` and have values specified there. This way you are not hard-coding values in different portions of your code.

Use of Java serialization is not allowed. Your classes for the message types should not implement the `java.io.Serializable` interface.

2.1 Registration:

Upon starting up, each messaging node should register its IP address, and port number with the registry. It should be possible for your system to register messaging nodes that are running on the same host but are listening to communications on different ports. There should be 3 fields in this registration request:

```
Message Type (int): REGISTER_REQUEST
IP address (String)
Port number (int)
```

When a registry receives this request, it checks to see if the node had previously registered and ensures 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).

The contents of the response message are depicted below. The success or failure of the registration request should be indicated in the status field of the response message.

```
Message Type (int): REGISTER_RESPONSE
Status Code (byte): SUCCESS or FAILURE
Additional Info (String):
```

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 messaging node-registry maintained at the registry.

2.2 Deregistration

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

```
Message Type: DEREGISTER_REQUEST
Node IP address:
Node Port number:
```

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.

2.3 Peer messaging nodes list

Once the **`setup-overlay`** command (see section 3) is specified at the registry it must perform a series of actions that lead to the creation of the overlay via messaging nodes initiating connections with each other. Messaging nodes await instructions from the registry regarding the other messaging nodes that they must establish connections to.

The registry must ensure two properties. First, it must ensure that the number of links to/from (the links are bidirectional) every messaging node in the overlay is identical; this is configurable metric (with a default value of 4) and is specified as part of the **`setup-overlay`** 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 connection requirement for the overlay is $\mathbf{C_R}$, each messaging node will have $\mathbf{C_R}$ links to other messaging nodes in the overlay. The registry selects these $\mathbf{C_R}$ messaging nodes that constitute the *peer-messaging nodes list* for a messaging node randomly. However, a check should be performed to ensure that the peer-messaging nodes list for a messaging node does not include the targeted messaging node i.e. a messaging node should not have to connect to itself. The registry keeps track of the connections that are being created; for example, if messaging node A is asked to connect to messaging node B, the connection counts for both A and B are incremented. The registry must ensure that connection counts are met and not breached.

The registry sends a different list of messaging nodes to each messaging node in the overlay. To avoid duplicate connections being established between messaging nodes, only one messaging node in a link should be instructed to create the connection. For instance, if there is a link between nodes A and B, only node A should be instructed to establish a link with node B or vice versa. The number of peer messaging nodes included in messages to different messaging nodes may vary from $\mathbf{C_R}$ through 0. If a messaging node's connection limit was reached through previous messages sent to other messaging

nodes in the overlay, a message still needs to be sent to that messaging node. The peer-list message will have the following format

```
Message Type: MESSAGING_NODES_LIST
Number of peer messaging nodes: X
Messaging node1 Info
Messaging node2 Info
…..
Messaging nodeX Info
```

If a messaging node does not need to establish any connections, set the number of peer messaging nodes to 0. The information corresponding to a messaging node includes the following: `messaging node_hostname:portnum`. Upon receiving the `MESSAGING_NODES_LIST` message a messaging node should initiate connections to the specified messaging nodes. After establishing connections, a messaging node should print the message "All connections are established. Number of connections: x" to the console to help with testing and evaluating your implementation.

2.4 Assign overlay link weights

The registry is also responsible for assigning weights to connections in the overlay. The weight for each link is an integer between 1-10 and is randomly computed by the registry. This information will be encoded in the message as follows.

```
Message Type: Link_Weights
Number of links: L
Linkinfo1
Linkinfo2
...
LinkinfoL
```

A `Linkinfo` connecting messaging nodes A and B contains the following fields: `hostnameA:portnumA hostnameB:portnumB weight`

A single message should be constructed with all link weights and sent to all registered messaging nodes. A messaging node should process this message and store its information to generate routing paths for messages as explained in the following section. Further, it should acknowledge the receipt and processing of this message by printing the message "Link weights received and processed. Ready to send messages." to the console.

2.5 Initiate sending messages

The registry informs nodes in the overlay when they should start sending messages to each other. It does so via the `TASK_INITIATE` control message.

```
Message Type: TASK_INITIATE
Rounds: X
```

2.6 Send message

Data can be fed into the network from any messaging node within the network. 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*** messages 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 node to the sink node, the source node computes the shortest path to the sink node using Dijkstra's shortest path algorithm. This path is then used as a *routing plan* that will be included in the packet. The routing plan indicates how the packet must be routed; for example, A may have a direct connection to B, but depending on the link weights, the routing plan may call for the packet to sent as A → C → E → D → B. Since the link weights are fixed

once assigned, you may cache the routing plans once computed as an optimization. Also a message includes a payload which is a random integer as explained in section 4.

A key requirements 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.

2.7 Inform registry of task completion

Once a node has completed its task of sending a certain number of messages in *rounds* (described in section 4), it informs the registry of its task completion using the `TASK_COMPLETE` message. This message will have the following format:

```
Message Type: TASK_COMPLETE
Node IP address:
Node Port number:
```

2.8 Retrieve traffic summaries from nodes

Once the registry has received `TASK_COMPLETE` messages from all the registered nodes it will issue a `PULL_TRAFFIC_SUMMARY` message. This message is sent to all the registered nodes in the system. This message will have the following format. To allow all messages that are already in transit to reach their destination nodes, you should wait for some time (e.g., 15 seconds) after receiving all `TASK_COMPLETE` messages before issuing a `PULL_TRAFFIC_SUMMARY` message.

```
Message Type: PULL_TRAFFIC_SUMMARY
```

2.9 Sending traffic summaries from the nodes to the registry

Upon receipt of the `PULL_TRAFFIC_SUMMARY` message from the registry, the node will create a response that includes summaries of the traffic that it has participated in. The summary will include information about messages that were sent and received. This message will have the following format.

```
Message Type: TRAFFIC_SUMMARY
Node IP address:
Node Port number:
Number of messages sent
Summation of sent messages
Number of messages received
Summation of received messages
Number of messages relayed
```

Once the `TRAFFIC_SUMMARY` message is sent to the registry, the node must reset the counters associated with traffic relating to the messages it has sent and received so far e.g number of messages sent, summation of sent messages, etc.

3 Specifying commands and interacting with the processes

Both the registry and the messaging node should run as *foreground* processes and allow support for commands to be specified while the processes are running. The commands that should be supported are specific to the two components.

3.1 Registry

`list-messaging-nodes`
This should result in information about the messaging nodes (hostname, and port-number) being listed. Information for each messaging node should be listed on a separate line.

`list-weights`
This should list information about links comprising the overlay. Each link's information should be on a separate line and include information about the nodes that it connects to and the weight of that link. For example, **`carrot.cs.colostate.edu:2000 broccoli.cs.colostate.edu:5001 8`**, indicates that the link is between two messaging nodes (**`carrot.cs.colostate.edu:2000`**) and (**`broccoli.cs.colostate.edu:5001`**) with a link weight of 8.

`setup-overlay number-of-connections`
This should result in the registry setting up the overlay. It does so by sending messaging nodes messages containing information about the messaging nodes that it should connect to. The registry tracks the connection counts for each messaging node and will send the `MESSAGING_NODES_LIST` message (see Section 2.3) to every messaging node. A sample specification of this command is **`setup-overlay 4`** that will result in the creation of an overlay where each messaging node is connected to exactly 4 other messaging nodes in the overlay. You should handle the error condition where the number of messaging nodes is less than the connection limit that is specified.

NOTE: You are not required to deal with the case where a messaging node is added or removed after the overlay has been set up. You must however deal with the case where a messaging node registers and deregisters from the registry before the overlay is set up.

`send-overlay-link-weights`
This should result in a `Link_Weights` message being sent to all registered nodes in the overlay. This command is issued once after the **`setup-overlay`** command has been issued. This also allows all nodes in the system to be aware of not just all the nodes in the system, but also the complete set of links in the system.

The following figure depicts an example overlay with 8 messaging nodes with the connection requirement (C_R) set to 4. For instance, node A is connected with nodes B, C, G and H using bidirectional links with weights of 1, 4, 7 and 6 respectively.

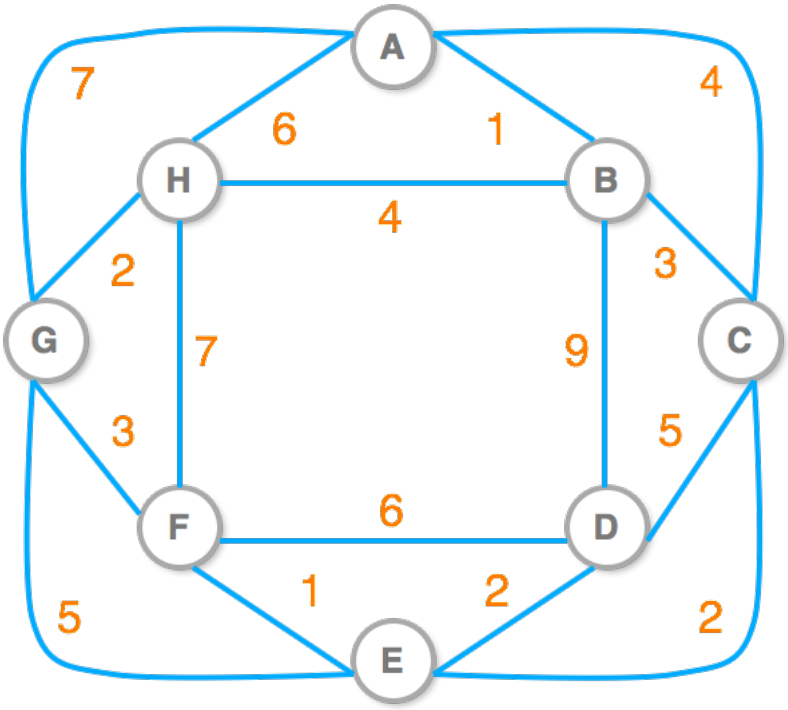

Figure 1: Graphical depiction of an overlay and link weights.

`start number-of-rounds`
The **`start`** command results in nodes exchanging messages within the overlay. Each node in the overlay will be responding for sending **`number-of-rounds`** messages. An advantage of this is that you are able to debug your system with a smaller set of messages and verify correctness of your programs across a wide range of values. A detailed description is provided in section 4 below.

3.2 Messaging node

`print-shortest-path`
This should print the shortest paths that have been computed to all other the messaging nodes within the system. The listing should also indicate weights associated with the links.
e.g. carrot--8--broccoli--4---zucchini---2--brussels--1--onion

`exit-overlay`
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.

4 Setting

For the remainder of the discussion, we assume that the **`setup-overlay`** command has been specified followed by the **`send-overlay-link-weights`** command at the register. Also, nodes will not be added to the system from hereon.

When the **`start`** command is specified at the registry, the registry sends the `TASK_INITIATE` control message to all the registered nodes within the overlay. Upon receiving this information 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 messages to a randomly chosen node (excluding itself, of course) from the set of registered nodes based on the `Link_Weights` message. All communications in the system will be based on TCP. To send a message the source node computes a routing plan (encoding the shortest path) with zero or more intermediate nodes relaying the message en route to the destination sink node. The payload of each message is a random integer with values that range from `2147483647` to `-2147483648`. At the end of each round, the process is repeated by choosing another node at random. The number of rounds initiated by each node is determined by the specified **`number-of-rounds`**.

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. When setting up the overlay, a messaging node only opens one connection at a time to another node, it may receive multiple incoming connections as other nodes try to connect to it.

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 messages that were sent by that node and the `receiveTracker` maintains information about the number of messages that were received. Additionally, each node will track the number of messages that it relayed – i.e., messages for which it was neither the source nor the sink. Consider the case where there are 10 nodes in the system as depicted in Figure 1. Since every node initiates 5000 rounds, each of which contains 5 messages, the number of messages sent by every node is 25,000. With 10 nodes in the system, the total number of messages would be 250,000. Since a sending node chooses the target node for each round at random, the number of messages received by different receivers would be different. However, because each round has 5 messages, the total number of messages received at a receiver would a multiple of 5 and close to 25,000 (i.e. it could 24000, 24595, 26905, …).

The number of messages that a node relays will depend on the overlay topology, link weights, and shortest paths that it belongs to. This is tracked using the variable `relayTracker`.

To track the messages 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.

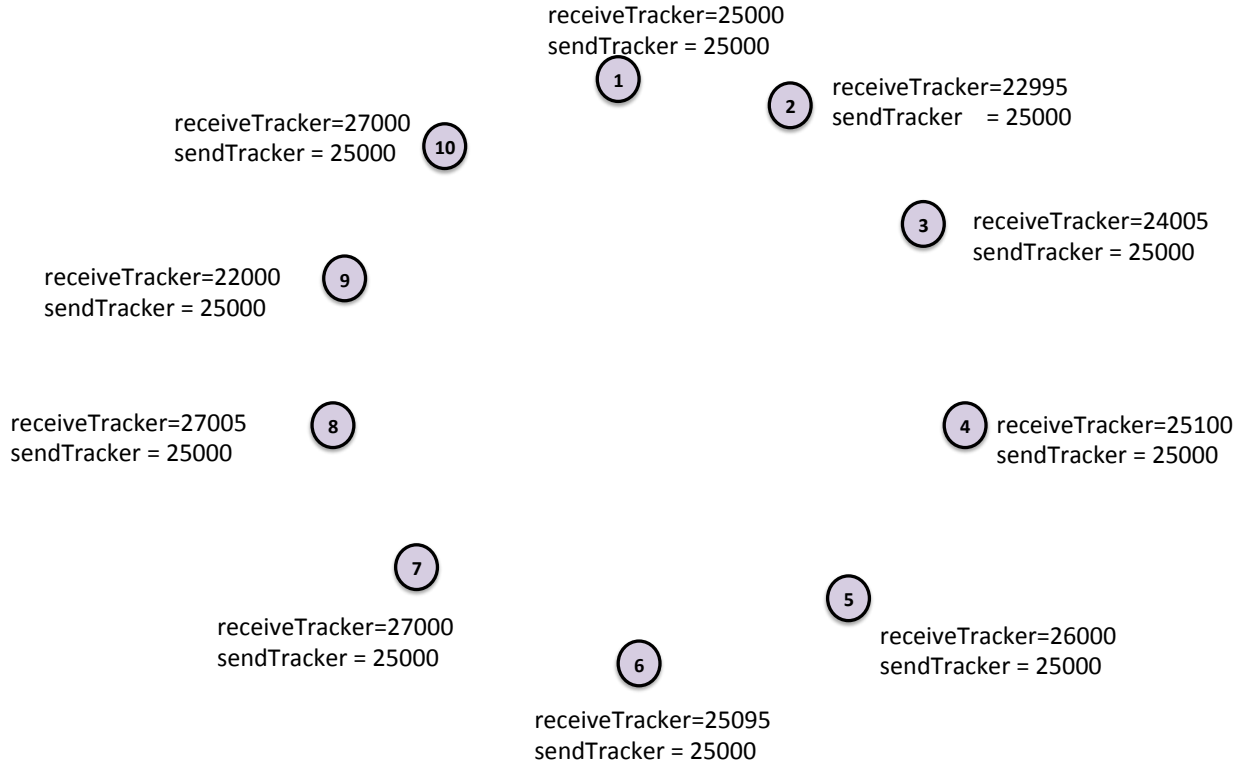

Figure 2: Depiction of a possible distribution of the number of messages sent and received within a set of 10 nodes.

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 node has completed its rounds, it will send a TASK_COMPLETE message to the registry. When the registry receives a TASK_COMPLETE message from each of the N registered nodes in the system, it issues a PULL_TRAFFIC_SUMMARY message to all the nodes.

Upon receipt of the PULL_TRAFFIC_SUMMARY message, a node will prepare to send information about the messages that it has sent and received. This includes: (1) the number of messages that were sent by that node, (2) the summation of the sent messages, (3) the number of messages that were received by that node, and (4) the summation of the received messages. The node packages this information in the TRAFFIC_SUMMARY message and sends it to the registry. After a node generates the TRAFFIC_SUMMARY, it should reset the counters that it maintains. This will allow testing of the software for multiple runs.

Upon receipt of the TRAFFIC_SUMMARY from all the registered nodes, the registry will proceed to print out the table as depicted below. Each row must be printed on a separate line.

Example output at the registry:
The collated outputs from 10 nodes are depicted below. Note how the number of received messages may be slightly different than the number of sent messages. The summation of sent or received messages at a node may be negative. In this particular example the final summation across all nodes is positive, it may well be negative in your case and that is fine!

	Number of messages sent	Number of messages received	Summation of sent messages	Summation of received messages	Number of messages relayed
Node 1	25000	25440	-340,040,800,604.00	-144,703,367,090.00	40445
Node 2	25000	25395	277,777,554,744.00	192,844,494,434.00	55435
Node 3	25000	24535	-42,851,633,614.00	199,699,309,204.00	60770
Node 4	25000	25130	184,871,797,810.00	91,406,191,639.00	30535
Node 5	25000	24245	-106,636,042,422.00	-180,588,270,287.00	10140
Node 6	25000	25120	24,251,523,172.00	398,033,468,762.00	78545
Node 7	25000	25205	145,053,292,085.00	-377,484,205,221.00	45675
Node 8	25000	24280	-235,398,166,411.00	51,922,993,583.00	8765
Node 9	25000	24985	-70,572,398,997.00	-100,564,359,421.00	15655
Node 10	25000	25665	328,837,533,087.00	34,726,403,247.00	16560
Sum	**250000**	**250000**	**165,292,658,850.00**	**165,292,658,850.00**	

5 Command line arguments for the two components

Your classes should be organized in a package called **`csx55.overlay`**. The command-line arguments and the order in which they should be specified for the Messaging node and the Registry are listed below

```
java csx55.overlay.node.Registry portnum

java csx55.overlay.node.MessagingNode registry-host registry-port
```

6 Grading

Homework 1 accounts for 15 points towards your final course grade.

Registry Breakdown: 8 points

1 point:	The registry is functional with support for registration and de-registration of nodes.
3 points:	Setting up of the overlay while ensuring that there are no partitions and satisfying the connection limit requirement Successful initiation of the message exchange process at *all* nodes. A node will not start sending messages until it receives the list of messaging nodes from the repository.
1 point:	Propagating the link weights
2 points	Traffic summaries are collated and printed out as depicted in the example table. This feature will assist in testing the program as well as during grading.
1 points	Responding to commands specified while interacting with the process

Messaging node Breakdown: 7 points

1 point	Establishing connections based on the `MESSAGING_NODES_LIST`
3 points	Routing data packets successfully within the overlay without duplication and complete reachability.
1 points	The mechanism for task completion and retrieval of traffic summaries works correctly
2 points	Message totals for send and receive match.

7 Deductions

There will be a **15-point deduction** if any of the restrictions below are violated.

1. The data that you will be sending will be byte[]. None of your classes can implement the `java.io.Serializable` interface.
2. No GUIs should be built under any circumstances. These are auxiliary paths and the deduction is in place to ensure that none of you attempt to do this.

8 Milestones:

You have 4 weeks to complete this assignment. The weekly milestones below correspond to what you should be able to complete at the end of every week.

Milestone 1: You should be able to have two nodes talking to each other i.e. you are able to exchange messages between two servers.

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 and link weights. You should also be able to issue all commands at the foreground processes.

Milestone 3: You should be able to compute the shortest at each messaging node, generate and enforce routing plans to route messages fed into the overlay. You should be able to track the summation counts for the messages and the contents of these messages.

Milestone 4: Iron out any wrinkles that may preclude you from getting the correct (i.e. not corrupted) outputs at all times.

9 What to Submit

Use **CANVAS** to submit a single .tar file that contains:
- all the Java files related to the assignment (please document your code)
- the `build.gradle` file you use to build your assignment
- a README.txt file containing a manifest of your files and any information you feel the TAs needs to grade your program.

Software versioning: Java 11 and gradle version 8.3

This environment is provided on CS lab machines using module load in Bash:
module load courses/cs455
module load courses/cs555

Filename Convention: The class names for your messaging node and registry should be as specified in Section 5. You may call your support classes anything you like. All classes should reside in a package called **`csx55.overlay`**. The archive file should be named as <FirstName>_<LastName>_HW1.tar. For example, if you are Cameron Doe then the tar file should be named Cameron-Doe-HW1.tar.

10 Version Change History

This section will reflect the change history for the assignment. It will list the version number, the date it was released, and the changes that were made to the preceding version. Changes to the first public release are made to clarify the assignment; the spirit or the crux of the assignment will not change.

Version	Date	Comments
1.0	1/17/2024	First public release of the assignment.
1.1	1/22/2024	Added information about setting up the environment using modules (See section 9) and updated CR lower-bound to 0 on page-3 in the peer messaging nodes list (section 2.3).