Homework 1

NETWORK COMMUNICATIONS OVER A LOGICAL OVERLAY
VERSION 1.0

DUE DATE: Wednesday, February 23rd, 2022 @ 5:00 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) sending messages over the overlay.

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 are allowed to use language libraries 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.

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 `cs455.overlay.wireformats.Protocol` and have values specified there. This way you are not hard-coding values in different portions of your code.

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).

If there are no errors in the request, the registry should assign an identifier for the node. As discussed in section (2.3) the identifiers are used to construct an overlay. The identifier generation process should be random and result in a value between 0-1023. The identifier generation process should also avoid collisions in the identifier space. For example, if the registry previously generated an ID of 455 and in the (low probability) event that the random number generates "455" again, the registry must regenerate the ID till such time that it produces a unique nodeIdentifier.

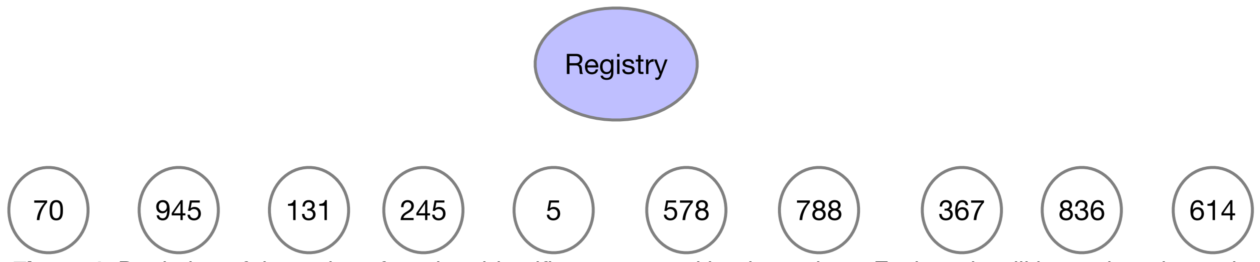

Figure 1: Depiction of the series of random identifiers generated by the registry. Each node will be assigned exactly one identifier and retains it for the duration that the process is running.

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
Node Identifier (int)
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. The messaging node does so by sending a control 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 Overlay Connections Directives

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 organizes the nodes in a ring structure similar to a clock: the nodes are sorted (in ascending order). Each node should be connected to two other nodes – a node with an identifier immediately higher than itself, and a node with an identifier immediately lower than itself. The registry which is aware of all the identifiers must account for wrap arounds i.e., the node with the highest identifier will be connected to the node with the 2nd highest identifier and the node with the lowest identifier. Figure 2 depicts the end-result of setup-overlay command for the set of nodes that were assigned identifiers in Figure 1.

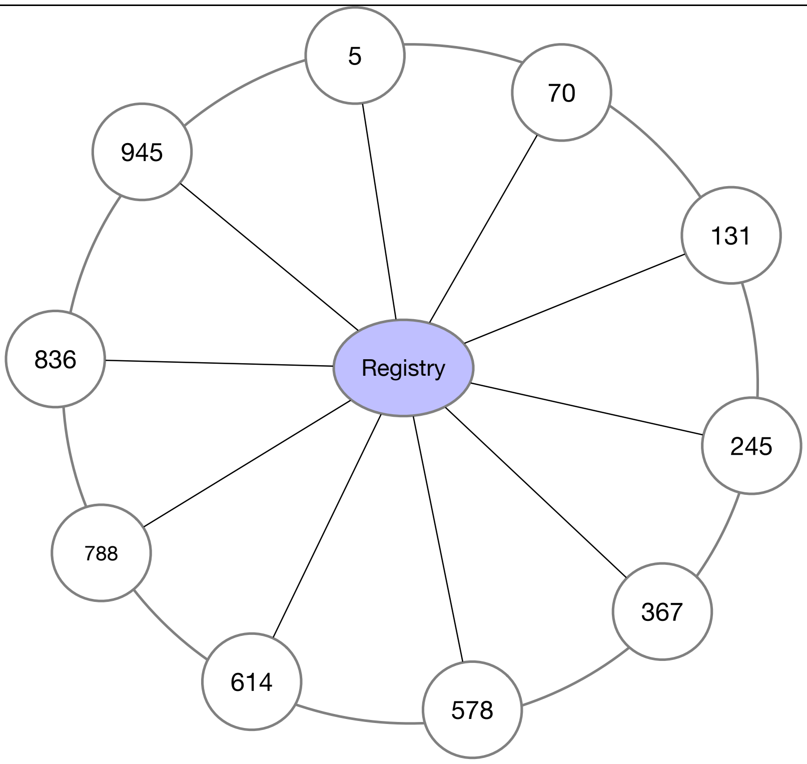

Figure 2: Once the setup overlay command is specified, the registry ensures that the nodes are organized in a ring structure. Note how the identifiers are sorted with each node being connected to nodes with identifiers greater than (or less than) itself. The solid lines represent connections initiated within the system.

The registry must ensure two properties. First, it must ensure that every messaging node is connected to at least two other nodes. Second, the registry must ensure that there is *no partition* within the overlay i.e., the overlay should not devolve into disjoint subgroups.

The registry sends a different set 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 peer-list message could have the following format

```
Message Type: CONNECTIONS_DIRECTIVE
Messaging node1 Info /* The first node to connect to */
Messaging node2 Info /* The second node to connect to */
```

The Info field encapsulates the host and port information of the messaging node in question.

2.4 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
Number (int)
```

Where `Number` represents the number of unique messages that a node must create and send to its clockwise neighbor.

2.5 Send messages

Each messaging node is required to create and send a specified number (up to 250,000) of different messages to their clockwise neighbor. Each message completes a circular trip around the ring topology via "relays" (message forwarding to the clockwise neighbor) facilitated by the other nodes. In particular, each node relays every received message; relaying is suppressed when a node encounters a message that it had originated.

Every message sent by a node must include two fields: the identifier of the node that created the message and a payload. The payload of each message is a random integer with values that range from `2147483647` to `-2147483648`. Note that both the `Node-Identifier` and `Payload` fields are fixed (and must not be changed) when received (or relayed) by other nodes. Each node completes a circular roundtrip around the ring topology created by the **`setup-overlay`** command (described in section 3).

The nodeIdentifer included in the originating message is used by the source to suppress any message that it had originally created. For example, if node-245 originated a message and this message is eventually received back at that node (after completing a clockwise trip around the ring topology) it should discard that message without any further processing. This ensures that each message does not make infinite trips around the topology.

```
Message Type: DATA_TRAFFIC
Node Identifier (int)
Payload (int)
```

2.6 Inform registry of task completion

Once a node has completed its task of sending a certain number of messages (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 Identifier (int)
Node IP address:
Node Port number:
```

2.7 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
```

2.8 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
```

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.

`setup-overlay`

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.

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.

`start number`

The **`start`** command results in nodes exchanging messages within the overlay. Each node in the overlay will be responding for sending the specified **`number`** of messages. An advantage of this parametrization to **`start`** 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. During grading, your system will be tested with the parameter **`number`** set to 250,000.

3.2 Messaging node

`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: Testing, Debugging, and Validation

For the remainder of the discussion, we assume that the **`setup-overlay`** command has been specified. 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. No nodes will be added (or removed) once the **`setup-overlay`** command has been specified.

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 sending messages to other nodes.

Each node is responsible for sending the specified **`number`** of messages: individual messages are sent in the clockwise direction.

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. The `sendTracker` only tracks the number of messages that were sent (not relayed) by a node. The `receiveTracker` tracks all messages that were received; this includes messages that a node created and eventually suppressed. For example, say node-945 created a message which makes a clockwise trip around the ring and arrives back at node-945; such a message is detected and suppressed but is also included in the receiver-side statistics at that node.

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" (not relayed) by a node, while the `receiveSummation` sums values of the payloads in all messages (excluding the messages that had originated at a given node) 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 node has completes sending the specified **`number`** of messages, 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 output depicted below represents the scenario where there are 10 nodes in the system, the overlay has been constructed by the registry, and the **`number`** of messages sent by a node is configured to be 250,000. The collated outputs from 10 nodes are depicted below. Note how the number of received messages at each node is 2,500,000 because each node receives *every* sent message. The summation of sent 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
Node 1	250,000	2,500,000	-340,040,800,604.00	165,292,658,850.00
Node 2	250,000	2,500,000	277,777,554,744.00	165,292,658,850.00
Node 3	250,000	2,500,000	-42,851,633,614.00	165,292,658,850.00
Node 4	250,000	2,500,000	184,871,797,810.00	165,292,658,850.00
Node 5	250,000	2,500,000	-106,636,042,422.00	165,292,658,850.00
Node 6	250,000	2,500,000	24,251,523,172.00	165,292,658,850.00
Node 7	250,000	2,500,000	145,053,292,085.00	165,292,658,850.00
Node 8	250,000	2,500,000	-235,398,166,411.00	165,292,658,850.00
Node 9	250,000	2,500,000	-70,572,398,997.00	165,292,658,850.00
Node 10	250,000	2,500,000	328,837,533,087.00	165,292,658,850.00
Sum	**2,500,000**	25,000,000	**165,292,658,850.00**	**1,652,926,588,500.00**

5 Command line arguments for the two components

Your classes should be organized in a package called **`cs455.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 cs455.overlay.node.Registry portnum

java cs455.overlay.node.MessagingNode registry-host registry-port
```

6 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. You should also be able to issue all commands at the foreground processes.

Milestone 3: You should be able to send, relay, and suppress messages. 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.

7 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.

E-mailing the codes to the Professor, GTA, or the class accounts will result in an automatic 1 point deduction.

Filename Convention: The class names for your client and server should be as specified in Section 5. You may call your support classes anything you like. All classes should reside in a package called **`cs455.scaling`**. The archive file should be named as TeamName-HW2.tar. For example, if your assigned team name is Alpha then the tar file should be named Alpha-HW2.tar.

8 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	01/20/2022	First public release of the assignment.