Homework 1: Programming Component

Routing Packets Within a Structured Peer-to-Peer (P2P) Network Overlay
VERSION 1.0

DUE DATE: Wednesday February 19th, 2020 @ 5:00 pm

Objectives & Overview

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 implementing routing schemes for packets in a structured peer-to-peer (P2P) overlay system. This assignment requires you to: (1) construct a logical overlay over a distributed set of nodes, and then (2) use partial information about nodes within the overlay to route packets. 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 will be imposing 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 verify **correctness** of packet exchanges between the source and sink 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 happen continually 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.

Some context for this assignment:
What you are building is 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. 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.

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:

A. Allows messaging nodes to register themselves. This is performed when a messaging node starts up for the first time.
B. 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.
C. Allows messaging nodes to deregister themselves. This is performed when a messaging node leaves the overlay.
D. 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 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.

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. Along with the semantics of these interactions, the prescribed wire-formats have also been included. 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.

Use of Java serialization is not allowed and the deductions for doing so are very steep; please see the deductions section for additional information. 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 to register messaging nodes that are running on the same host but are listening to communications on different ports. There should be 4 fields in this registration request:

```
byte: Message Type (OVERLAY_NODE_SENDS_REGISTRATION)
byte: length of following "IP address" field
byte[^^]: IP address; from InetAddress.getAddress()
int: Port number
```

When a 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 below. The success or failure of the registration request should be indicated in the status field of the response message.

```
byte: Message type (REGISTRY_REPORTS_REGISTRATION_STATUS)
int: Success status; Assigned ID if successful, -1 in case of a failure
byte: Length of following "Information string" field
byte[^^]: Information string; ASCII charset
```

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 message to the registry. This deregistration request includes the following fields

```
byte: Message Type (OVERLAY_NODE_SENDS_DEREGISTRATION)
byte: length of following "IP address" field
byte[^^]: IP address; from InetAddress.getAddress()
int: Port number
int: assigned Node ID
```

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 `REGISTRY_REPORTS_DEREGISTRATION_STATUS` that is similar to the `REGISTRY_REPORTS_REGISTRATION_STATUS` message.

2.3 Peer node manifest

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 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-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 routing table size requirement for the overlay is $\mathbf{N_R}$, each messaging node will have links to $\mathbf{N_R}$ other messaging nodes in the overlay. The registry selects these $\mathbf{N_R}$ 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 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 `REGISTRY_SENDS_NODE_MANIFEST` message. The contents of the manifest message are different for each messaging node (since the routing table at every messaging node would be different). The wire format is shown when $\mathbf{N_R}$=3, if $\mathbf{N_R}$=4 there will also be an entry for a node $2^{(N_R-1)}$ hops away.

```
byte: Message type; REGISTRY_SENDS_NODE_MANIFEST
byte: routing table size  NR
int: Node ID of node 1 hop away
byte: length of following "IP address" field
byte[^^]: IP address of node 1 hop away; from InetAddress.getAddress()
int: Port number of node 1 hop away
int: Node ID of node 2 hops away
byte: length of following "IP address" field
byte[^^]: IP address of node 2 hops away; from InetAddress.getAddress()
int: Port number of node 2 hops away
int: Node ID of node 4 hops away
byte: length of following "IP address" field
byte[^^]: IP address of node 4 hops away; from InetAddress.getAddress()
int: Port number of node 4 hops away

byte: Number of node IDs in the system
int[^^]: List of all node IDs in the system [Note no IPs are included]
```

Note that the manifest 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 `REGISTRY_SENDS_NODE_MANIFEST` 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.

```
byte: Message type (NODE_REPORTS_OVERLAY_SETUP_STATUS)
int: Success status; Assigned ID if successful, -1 in case of a failure
byte: Length of following "Information string" field
byte[^^]: Information string; ASCII charset
```

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 `REGISTRY_REQUESTS_TASK_INITIATE` control message. This message also includes the number of packets that must be sent by each messaging node.

```
byte: Message type; REGISTRY_REQUESTS_TASK_INITIATE
int: Number of data packets to send
```

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.

```
byte: Message type; OVERLAY_NODE_SENDS_DATA
int: Destination ID
int: Source ID

int: Payload

int: Dissemination trace field length (number of hops)
int[^^]: Dissemination trace comprising nodeIDs that the packet traversed
through
```

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 completed its task of sending a certain number of messages (described in section 4), it informs the registry of its task completion using the OVERLAY_NODE_REPORTS_TASK_FINISHED message. This message should have the following format:

```
byte: Message type; OVERLAY_NODE_REPORTS_TASK_FINISHED
byte: length of following "IP address" field
byte[^^]: Node IP address:
int:  Node Port number:
int:  nodeID
```

2.8 Retrieve traffic summaries from nodes

Once the registry has received OVERLAY_NODE_REPORTS_TASK_FINISHED messages from all the registered nodes it will issue a REGISTRY_REQUESTS_TRAFFIC_SUMMARY message. This message is sent to all the registered nodes in the overlay. This message will have the following format.

```
byte: Message Type; REGISTRY_REQUESTS_TRAFFIC_SUMMARY
```

2.9 Sending traffic summaries from the nodes to the registry

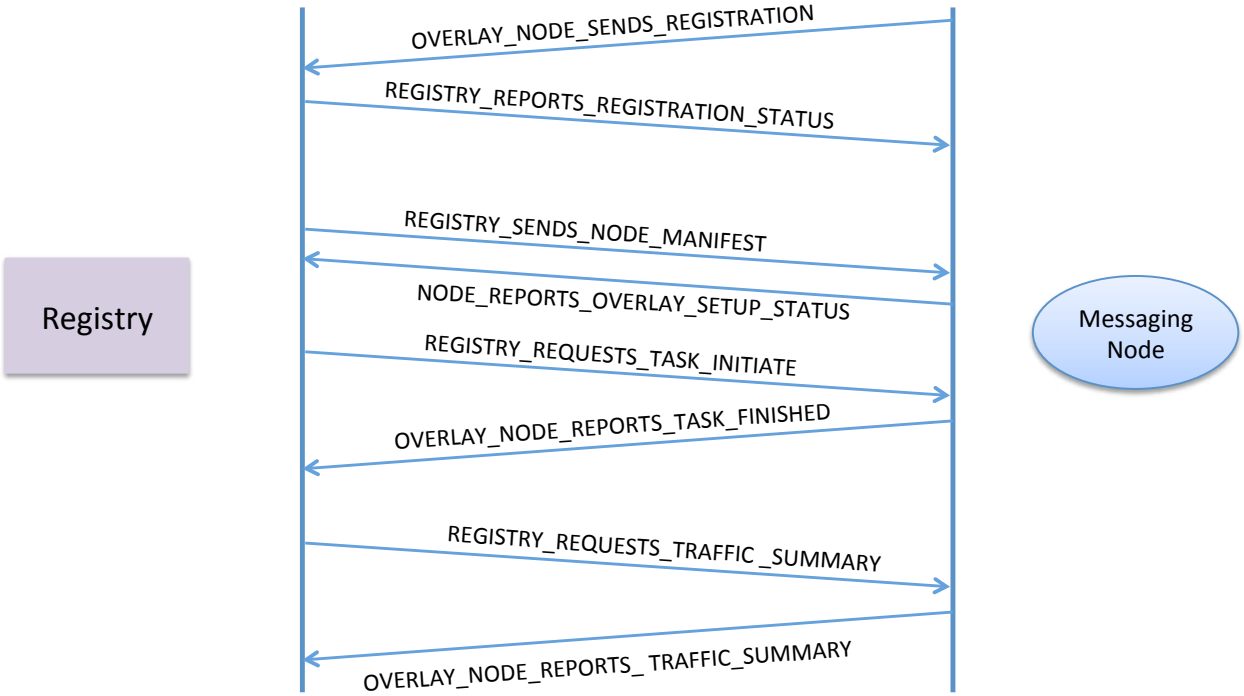
Upon receipt of the REGISTRY_REQUESTS_TRAFFIC_SUMMARY message from the registry, the messaging node will create a response that includes summaries of the traffic that it has participated in. The summary will include information about messages that were sent, received, and relayed by the node. This message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

```
byte: Message type; OVERLAY_NODE_REPORTS_TRAFFIC_SUMMARY
int: Assigned node ID
int: Total number of packets sent
       (only the ones that were started/initiated by the node)
int: Total number of packets relayed
       (received from a different node and forwarded)
long: Sum of packet data sent
       (only the ones that were started by the node)
int: Total number of packets received
       (packets with this node as final destination)
long: Sum of packet data received
       (only packets that had this node as final destination)
```

Once the OVERLAY_NODE_REPORTS_TRAFFIC_SUMMARY 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.

2.10 Summary of Messages Exchanged between the registry and node

The figure below depicts the exchange of messages between the registry and a particular messaging node in the system.



2.11 Values for the control messages

Please use the following values for your message types.

OVERLAY_NODE_SENDS_REGISTRATION	2
REGISTRY_REPORTS_REGISTRATION_STATUS	3
OVERLAY_NODE_SENDS_DEREGISTRATION	4
REGISTRY_REPORTS_DEREGISTRATION_STATUS	5
REGISTRY_SENDS_NODE_MANIFEST	6
NODE_REPORTS_OVERLAY_SETUP_STATUS	7
REGISTRY_REQUESTS_TASK_INITIATE	8
OVERLAY_NODE_SENDS_DATA	9
OVERLAY_NODE_REPORTS_TASK_FINISHED	10
REGISTRY_REQUESTS_TRAFFIC_SUMMARY	11
OVERLAY_NODE_REPORTS_TRAFFIC_SUMMARY	12

3 Specifying commands and interacting with the processes

Both the registry and the messaging nodes 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, port-number, and node ID) being listed. Information for each messaging node should be listed on a separate line.

`setup-overlay number-of-routing-table-entries (e.g. setup-overlay 3)`
This should result in the registry setting up the overlay. It does so by sending every messaging node the `REGISTRY_SENDS_NODE_MANIFEST` message that contains information about the routing table specific to that node and also information about other nodes in the system.

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.

`list-routing-tables`
This should list information about the computed routing tables for each node in the overlay. Each messaging node's information should be well separated (i.e., have 3-4 blank lines between node listings) and should include the node's IP address, portnum, and logical-ID. This is useful for debugging.

`start number-of-messages (e.g. start 25000)`
The **`start`** command results in the registry sending the `REGISTRY_REQUESTS_TASK_INITIATE` to all nodes within the overlay. A command of **`start 25000`** results in each messaging node sending `25000` packets to nodes chosen at random (of course, a node should not send a packet to itself). A detailed description of the sequence of actions that this triggers is provided in section 4.

3.2 Messaging node

`print-counters-and-diagnostics`
This should print information (to the console using System.out) 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-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. 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 `NODE_REPORTS_OVERLAY_SETUP_STATUS` message. Only after all nodes report success in setting up connections should the registry print out information on the console saying: "`Registry now ready to initiate tasks.`"

When the **`start`** command is specified at the registry, the registry sends the `REGISTRY_REQUESTS_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 a packet to a randomly chosen node (excluding itself, of course) from the set of all registered nodes advertised in the `REGISTRY_SENDS_NODE_MANIFEST`. All communications in the system will be based on 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 `REGISTRY_REQUESTS_TASK_INITIATE` 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 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.

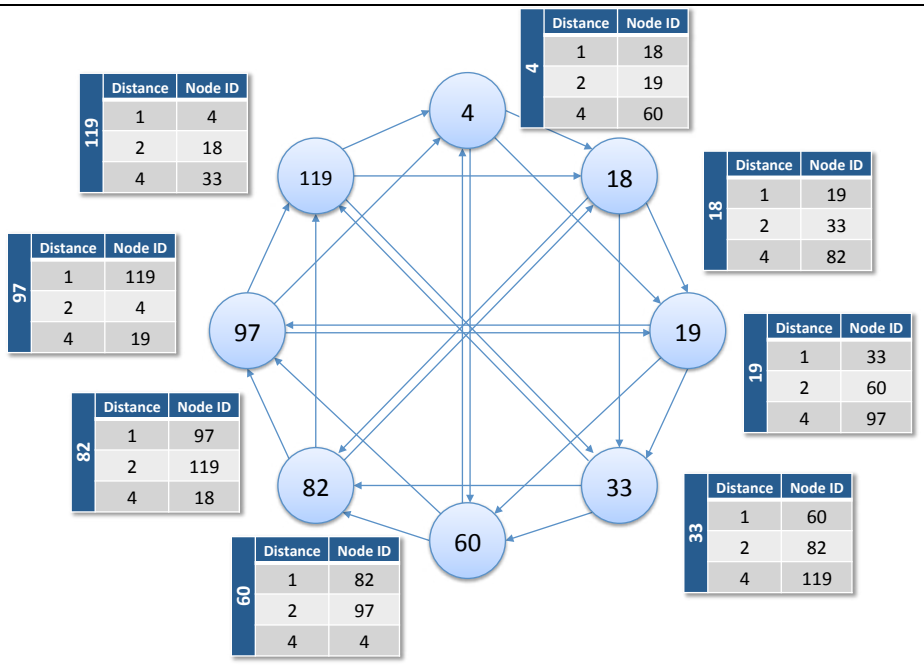


Figure 1: Depiction of a possible overlay and the routing table at each node.

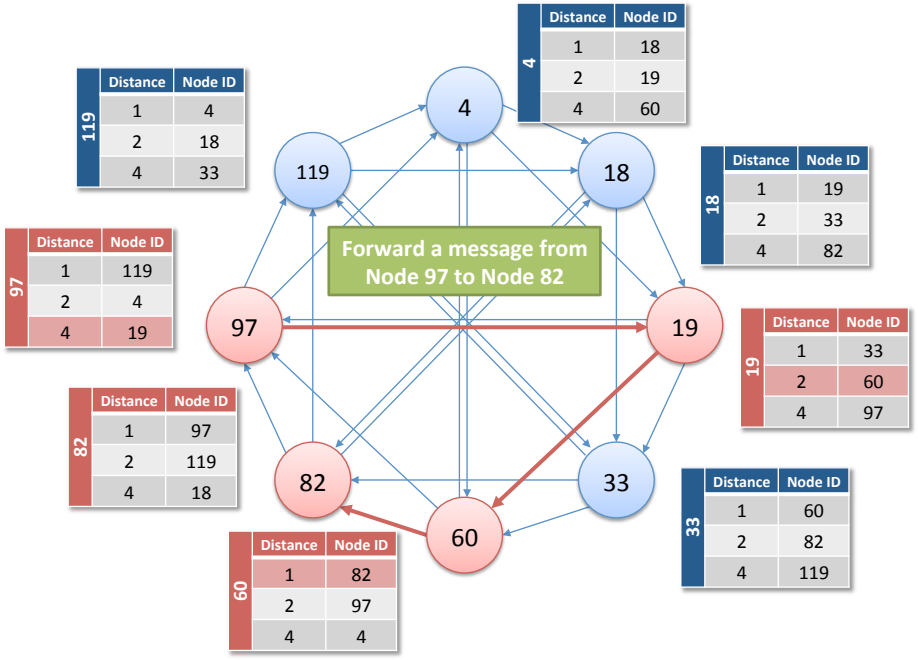


Figure 2: Depiction of how packets are routed (from 97 to 82)

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 will send a `OVERLAY_NODE_REPORTS_TASK_FINISHED` message to the registry. When the registry receives an `OVERLAY_NODE_REPORTS_TASK_FINISHED` message from each of the N registered nodes in the system, it issues a `REGISTRY_REQUESTS_TRAFFIC_SUMMARY` message to all the nodes.

Upon receipt of the `REGISTRY_REQUESTS_TRAFFIC_SUMMARY` 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 (4) the summation of the received packets. The node packages this information in the `OVERLAY_NODE_REPORTS_TRAFFIC_SUMMARY` message and sends it to the registry. After a node generates the `OVERLAY_NODE_REPORTS_TRAFFIC_SUMMARY`, it should reset the counters that it maintains. This will allow testing the software for multiple runs.

Example output at the registry: Upon receipt of the `OVERLAY_NODE_REPORTS_TRAFFIC_SUMMARY` from all the registered nodes, the registry will proceed to print out the table as depicted below with each row on a separate line. The collated outputs from 15 nodes are depicted below. Note how the number of received messages may be slightly different than the number of sent messages at each node. The summation of sent or received messages 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!

	Packets Sent	Packets Received	Packets Relayed	Sum Values Sent	Sum Values 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**

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 Grading

Homework 1 accounts for 15 points towards your final course grade. The programming component accounts for 80% of these points with the written element (to be posted later) accounting for the remaining 20%. This programming assignment will be graded for 12 points. The point distribution for this assignment is listed below.

Registry Breakdown: 6 points

1 point:	The registry is functional with support for registration and de-registration of nodes.
3 points:	Setting up of the overlay with the correct routing tables while ensuring that there are no network partitions i.e. all nodes can be reached. Successful initiation of the message exchange process at *all* nodes. A node will not start sending messages until the overlay setup has completed successfully.
1 point:	Successful retrieval of task summaries from the messaging nodes
1 point:	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.

Messaging node Breakdown: 6 points

1 point	Establishing connections based on the `REGISTRY_SENDS_NODE_MANIFEST`
2 points	Routing data packets successfully within the overlay without duplication and complete reachability.
1 point	The mechanism for task completion and retrieval of traffic summaries works correctly
2 points:	Message totals for send and receive match. This includes both the message counts and the content summation counts.

7 Deductions

There will be a **12-point deduction** (i.e. a 100% penalty) 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.
3. If you use distributed objects (e.g. RMI, CORBA, etc.) to implement this assignment.

There will be a **18-point deduction** (i.e. a 150% penalty)

1. 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.
2. Exchanging code with your peers.

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

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.

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 the CS455 checkin program 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 description of each file and any information you feel the GTA needs to grade your program.

The folder set aside for this assignment's submission using checkin is **HW1-PC**

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 **`cs455.overlay`**. The archive file should be named as <LastName>_<FirstName>_ASG<x>.tar. For example, if you are John Doe then the tar file should be named Doe_John_ASG1.tar.

10 Change History

Version	Date	Change
1.0	1/20/2020	First public release of the assignment.