

COMP/ELEC 416: Computer Networks

Project#3: Part 2

ModivSim: Modified Distance Vector Routing
Simulator

Ege Berk Süzgen, 53714

Emir Atışay, 53590

Onur Şahin, 64232

27.05.2020

Table of Contents

Introduction:	3
Project Setup:	3
Project Execution Scenario:	4
Detailed Explanation:	5
Modivsim Interaction and Architecture:	10
Performance Results:.....	13
Static Link Cost Scenario:.....	13
Dynamic Link Cost Scenario:.....	13
Flow Routing Scenario:	14
External Libraries:	18
Conclusion:	18

Introduction:

In this project it is asked to build a network structure from the very beginning and simulate it according to a modified distance vector routing algorithm. The input of the program is the core of the network structure which is going to be built. Each node (router) in the topology is retrieved from the input with their required fields such as link cost and link bandwidth. Both of those two fields are required to determine the router interaction, in other words links between each node are identified. After the network topology is prepared properly, there is the simulation part which routers start to inform their neighbors according to their instantiated fields. Those operations will end when all nodes stop updating their distance table in the other words, when the convergence is supplied.

Project Setup:

This project is implemented with Java and IntelliJ IDE is chosen to perform that implementation. It's ease of use and the simple debug interface are some of the reasons to choose IntelliJ as the main IDE. Before running the code, an external library which is SwingX should be imported in the libraries section of the project structure. Details of that library will be given in the "External Libraries" part of the report.

In order to create the desired network topology, the input part of the project should be proper. The project already has its input part, under the input package located in the src directory of the project. In the input package, there is the input.txt file which is the input of the program. Based upon that sample, the input of the program can be changed without changing the node link cost and link bandwidth structure. Followingly, there is also flow.txt, that contains the experiment flows that will take place after the system converges. To have different experiments, one can manipulate the input.txt and flow.txt while not violating the convention.

For the input.txt, the example convention is <Node Number>,<Neighbor Number, Link to Neighbor's Cost, Link Bandwidth>....<Neighbor Number, Link to Neighbor's Cost, Link Bandwidth>.

For the flow.txt, the example convention is <Flow Label>,<Source, Destination, Flow Size>.

Project Execution Scenario:

1. Fill the input.txt and flow.txt simulation scenarios from the input folder (Those parts are filled according to the sample topology of the instruction for first use).
2. Execute the Application.java file.
 - a. Application asks user for the period duration.
 - b. Application instantiates the ModivSim and begins to the simulation.
 - c. ModivSim reads the inputs and flows
 - d. According to the inputs, ModivSim generates the defined topology.
 - e. For the routers, ModivSim generates Node class which are also threads.
 - f. In a periodic manner, ModivSim triggers the update function of the Nodes.
 - g. Nodes with an update message trigger other node to receive the message.
 - h. Nodes check their distance table whether it converged.
 - i. ModivSim observes every node in the topology to indicate whether it is converged or not.
 - i. If the system is not converged, update periods continue to take place. (Step e)
 - ii. If the system converged, ModivSim began to send the packets to their source.
 1. Node Threads checks in every iteration if they obtain a task from ModivSim.
 2. Packets are sent or saved to the waiting queue according to their forwarding table.
 - a. If a packet is sent, ModivSim informed and used link is occupied for time calculated from the flow size and bottleneck bandwidth.
 - b. If a packet is in the queue, it waits for the link to become not occupied.

Detailed Explanation:

There are two parts which are going to be explained in a detailed way in this section which are Node and ModivSim implementation. Because the other parts such as popup classes are not complicated as Node and Modivsim and they are fully based on their specific purposes. So, there are not many functions with complex algorithms.

At first, the Node class will be examined without mentioning functions and fields which are already known by the project manual. The node class is the constructor of a router in a network topology. At a glance, Node class seems to have a high number of fields. A big majority of those fields are used for specific cases like a convergence checker called “converged” or for instance; some functions such as receiveUpdate() needs some information before being called for simplicity. The “neighborIDList” field is created for this usage. There are a bunch of fields which are used in the flow section of the project. There is “waitingQueue”. It is used if there is no path available for the flow, it enters the waiting queue and starts to wait and there is an available path for the flow. Also, there is the “tasks” field to hold the assigned flows. It also have getTask() function, to assign a packet from the simulator or the direct neighbors. In the run function, node checks whether there is a task, if there is any, it calls for flow function. For the link bandwidth operations, there is an additional field called “linkBandwidthOccupation” a hashtable holds a boolean for links whether they are occupied or not. The “popup” field is used to instantiate each Node’s each popup at the screen which shows the distance table and distance vector for the node. At last there are fields for the dynamic link cost scenario: “dynamicLink”, “rndBool” and “dynamicLinkTo”. They are used in the sendUpdate() function. With the “dynamicLink” field the link is checked whether it is dynamic or not in the very beginning of the sendUpdate() function. If it is dynamic the links are set randomly according to the result of the “rndBool” field and replace it with the previous costs in the distance table. The first function is the constructor of the Node class. It sets primary fields and accomplished starting operations such as files the distance table before any updates with the link cost input which is taken from the user. Then there is the essential run function because each node is implemented as a Thread and they are not just running in anyway. They have some running conditions which are briefly explained with the “tasks” filed. There has to be at least one task in the waiting queue for the thread to start. One of the most complicated functions in the Node class is the synchronized

“flow()” function. It is implemented as synchronized because there is the opportunity of race condition to be occurred via Node Threads. So, the synchronized feature of Java allows to avoid the race condition without implementing semaphores or a mutex. In the flow function, given flow is parsed. Then, if the task is not in the queue, its traceroute enlarges with the current node. Followingly, the current node checks whether the destination of the node is its. If it is the destination, first it signals the ModivSim that the packet is arrived to the desired place. After observing the bottleneck bandwidth, it calculates the total time of the transmission and sets a timer to release the used links after that transmission time. However, if it is not the destination node, it checks for its forwarding table and two best alternative nodes. If non occupied links is chosen from those alternatives, node signals its bandwidth to the next router and mark the following link as occupied. If no such link is found, the packet is added to the waitingQueue of the node. To mention the base functions such as sendUpdate(), it first checks for the dynamic links to assign it a random value if there is any. Followingly it calculates its distance vector, by another function called. If the distance vector do not match with the previous one, the one in the global scope, it updates its vector and distribute it via calling the receiveUpdate() of the others. In the receiveUpdate(), Bellman-Ford algorithm is implemented by updating the distance table by the given distance vector of the neighbor.

At last there are three function to deal with NodePopup operations. Those functions are refreshPopup() which refills the opened popup with the new distance table values, visualizeDistanceTable() function is used to update the JPanel in the NodePopup and visualizeVectorDistance() is also updates the JPanel in the NodePopup. The following figure demonstrates how those functions work. The first column is the first JPanel which does not require a function to update itself, it only takes the current round while being executed. The second one is not a JPanel, it is a JPanel. The reason for using JPanel is explained in the further part of the report. It shows the visually modified version of the distance table as it is demanded in the project manual. This modification is made by visualizeDistanceTable() function. And the most bottom JPanel is the visually modified distance vector created by the visualizeVectorDistance() function.

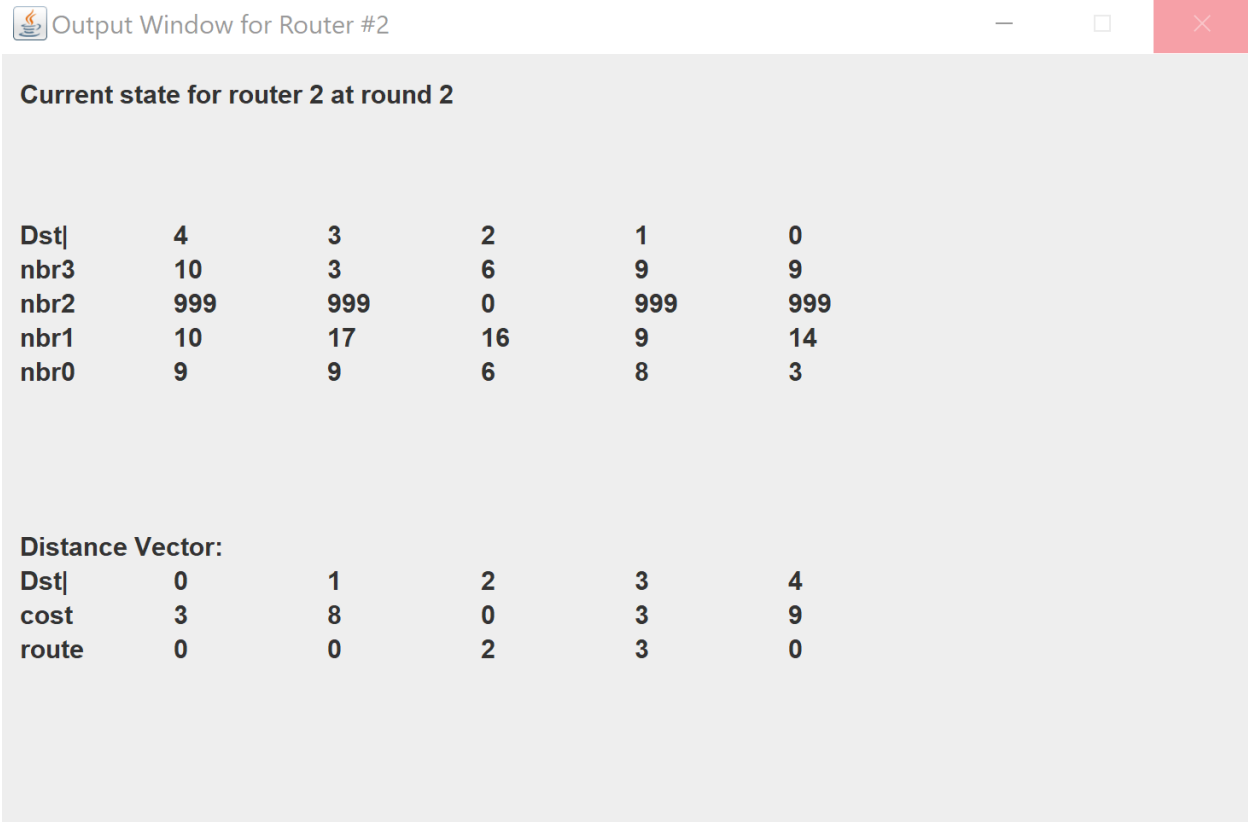


Figure 1

The Message and Packet classes are kind of utility classes. Message class represents the possible messages between the routes about their distance tables. In that regard, the Message class holds the source, destination, distance vector of the sender and the bandwidth of the used channel. Followingly, the Packet class represents the flow details between the routers. To have clear sight about the flows, it has flow name, flow info and flowroute. Flow info is the enlarged version of the given input form the flow.txt, where it holds the information about the bottleneck bandwidth. Every node in the traceroute updates it if its links is narrower and the destination of the packet then examines that bandwidth to distribute the transmission time to the other nodes. Followingly, all the receiver nodes add themselves to the flow route. Again, it helps to have better visualization and examination of the transmission.

Last and foremost, ModivSim is the brain of the system. This class is implemented based on the singleton principle. ModivSim begins with analyzing the inputs, the topologic information and the packets wanted to be tested on given topology. After parsing the input files, ModivSim

creates an instance for every node defined and holds them in a list of nodes which is `nodeList`. And holds the flows in the `LinkedHashMap` `flowS`. In addition, it has `ModivsimPopup` to prompt the wanted activities in the system. It also has utility functions like `returnNeighbor()` to ensure that the routers do not directly communicate with each other. If a node wants to communicate with its neighbor, in receiving messages or forwarding a packet flow, it queries the instance of the `ModivSim`. Followingly, the main function of the `ModivSim` is run, which coordinates the simulation. After the input parsing, node threads are started, and Timer thread is instantiated to have the periodic approach. In every period, if the system is not converged, nodes in the `nodeList` are triggered for their `sendUpdate()` function. After that process, the round is incremented. However, if the system is converged, packets from the `flowS` begin to be distributed. To have the packet convention mentioned above, flows are parsed again. To send them for once, a Boolean `sent` is used. After all, the system is again checked for convergence. The convergence check is based on the attribute of the nodes. Every node checks them as converged, if they observe no difference on their distance table in their updates. As it is wanted, no termination clause is implemented.

The below figure illustrates a sample `ModivsimPopup` output. It is not the same as it is demanded in the project manual. It is decided to print more than that in order to understand the flow better. There are 2 types of different output logs. The first two are when a router sends an update to another one, it informs the system with their Node ID's and the receiver also does the same operation in a reversed manner. Another log is the update information of a node, it indicates which node updated its distance table. The last type of the log is the inner information about a message which is sent from a node to another one. It dramatically facilitates the tracing, if it is needed.


```
C:\Users\HP\IdeaProjects\Comp416-Project3.2\src\input
0
1 has received message from 0
1 has updated its distance table
2 has received message from 0
2 has updated its distance table
0 has sent message to 1
  Content of the message is the new distance vector of 0: {2=3, 1=5, 0=0}
1 has received message from 0
0 has sent message to 2
  Content of the message is the new distance vector of 0: {2=3, 1=5, 0=0}
2 has received message from 0
0 has received message from 1
0 has updated its distance table
2 has received message from 1
2 has updated its distance table
4 has received message from 1
4 has updated its distance table
1 has sent message to 0
  Content of the message is the new distance vector of 1: {4=1, 2=8, 1=0, 0=5}
0 has received message from 1
1 has sent message to 2
  Content of the message is the new distance vector of 1: {4=1, 2=8, 1=0, 0=5}
2 has received message from 1
1 has sent message to 4
  Content of the message is the new distance vector of 1: {4=1, 2=8, 1=0, 0=5}
4 has received message from 1
0 has received message from 2
0 has updated its distance table
1 has received message from 2
1 has updated its distance table
3 has received message from 2
3 has updated its distance table
2 has sent message to 0
  Content of the message is the new distance vector of 2: {4=10, 3=3, 2=0, 1=8, 0=3}
0 has received message from 2
2 has sent message to 1
  Content of the message is the new distance vector of 2: {4=10, 3=3, 2=0, 1=8, 0=3}
1 has received message from 2
2 has sent message to 3
```

Figure 2

Modivsim Interaction and Architecture:

The following figure illustrates the ModivSim class's interaction with other classes with a potential sample scenario. When the Application is started, a ModivSim instance is created. For a better design, the ModivSim class is implemented as a singleton. The instance of the ModivSim class calls its run function to take the input and start each Node Thread according to the input. After that, nodes call their sendUpdate() functions, also their receiveUpdate() functions are triggered. So, their distances are modified which needs to be reported each state. After those operation, Modivsim instance calls its fillPopup() function to display the current situation of the network topology.

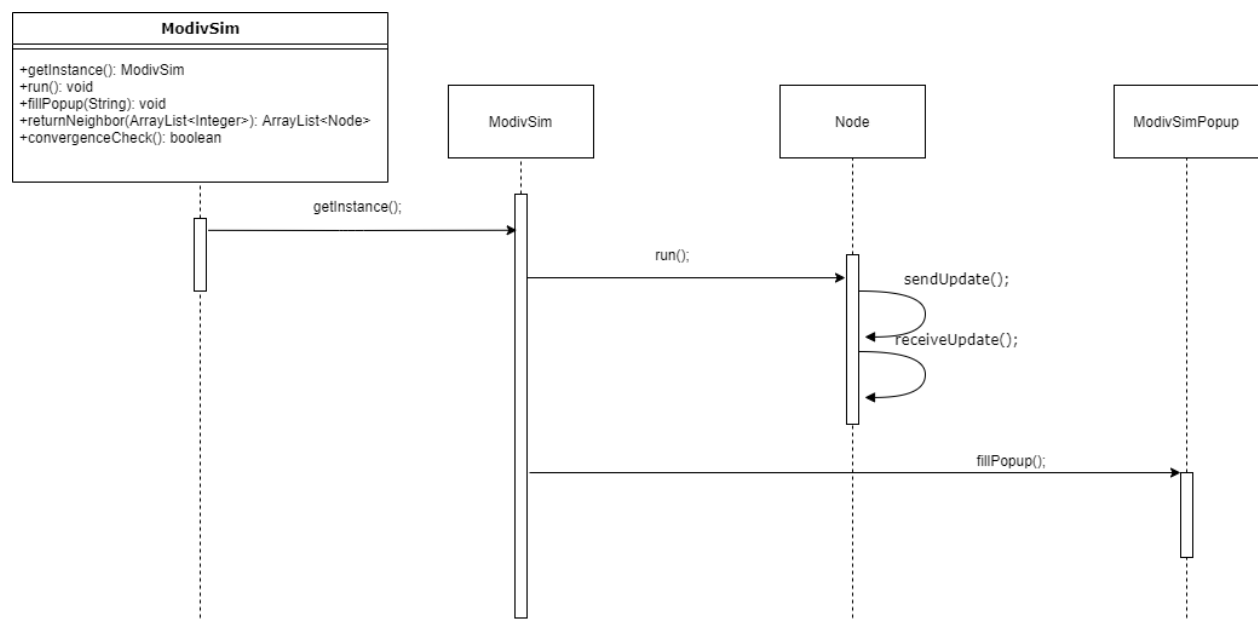


Figure 3

In the following figure, the sample packet flow is shown in the sequence diagram. In the run function of the ModivSim, Node's getTask() function is called, and the given packet is assigned to that node. Followingly, node begins to distribute the packets as they have a task to do and calls flow function for that purpose. In the flow function, again getTask() is used, this time

for the next node. To have coordination, neighbor nodes are obtained from the ModivSim instance. The main motivation behind this is to prohibit the interactions of the non-neighbor nodes. At the end, ModivSim fills its window by the fillPopup() function of its.

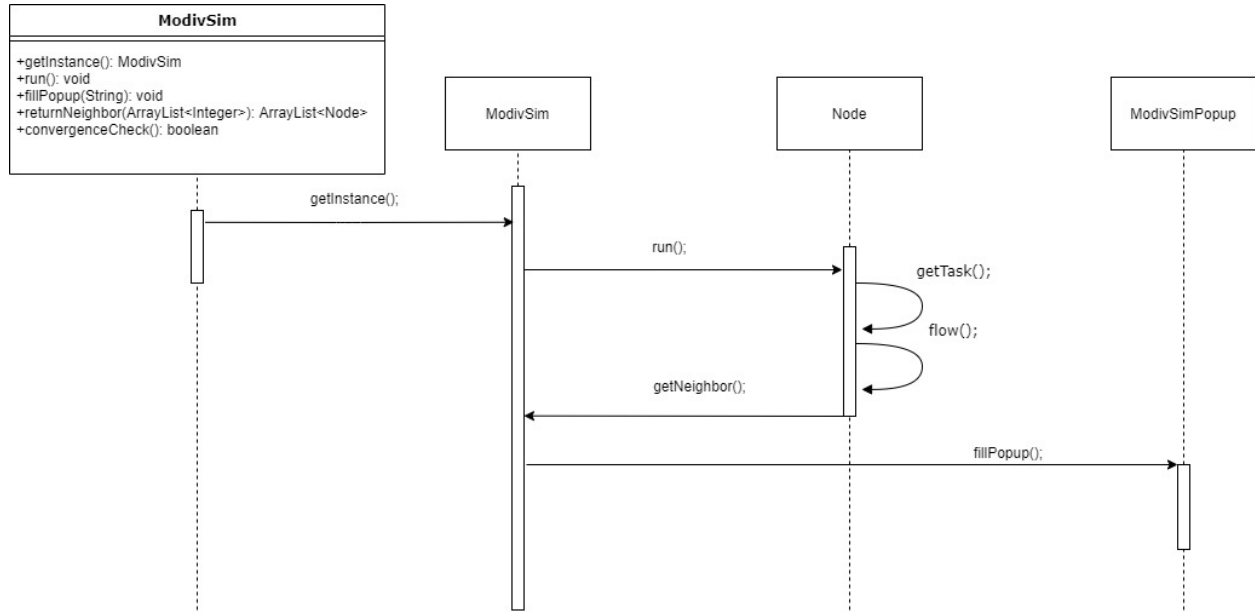


Figure 4

The following figure (5) represents the class diagram of the project. Each field and function of every class is shown in the figure. This visualization facilitates the project's understanding process by showing the relation between them. As it can be seen, different numbers of nodes can be assigned to the ModivSim. And Node can have multiple packets. However, Windows and ModivSim, Node are implemented in one-to-one manner.

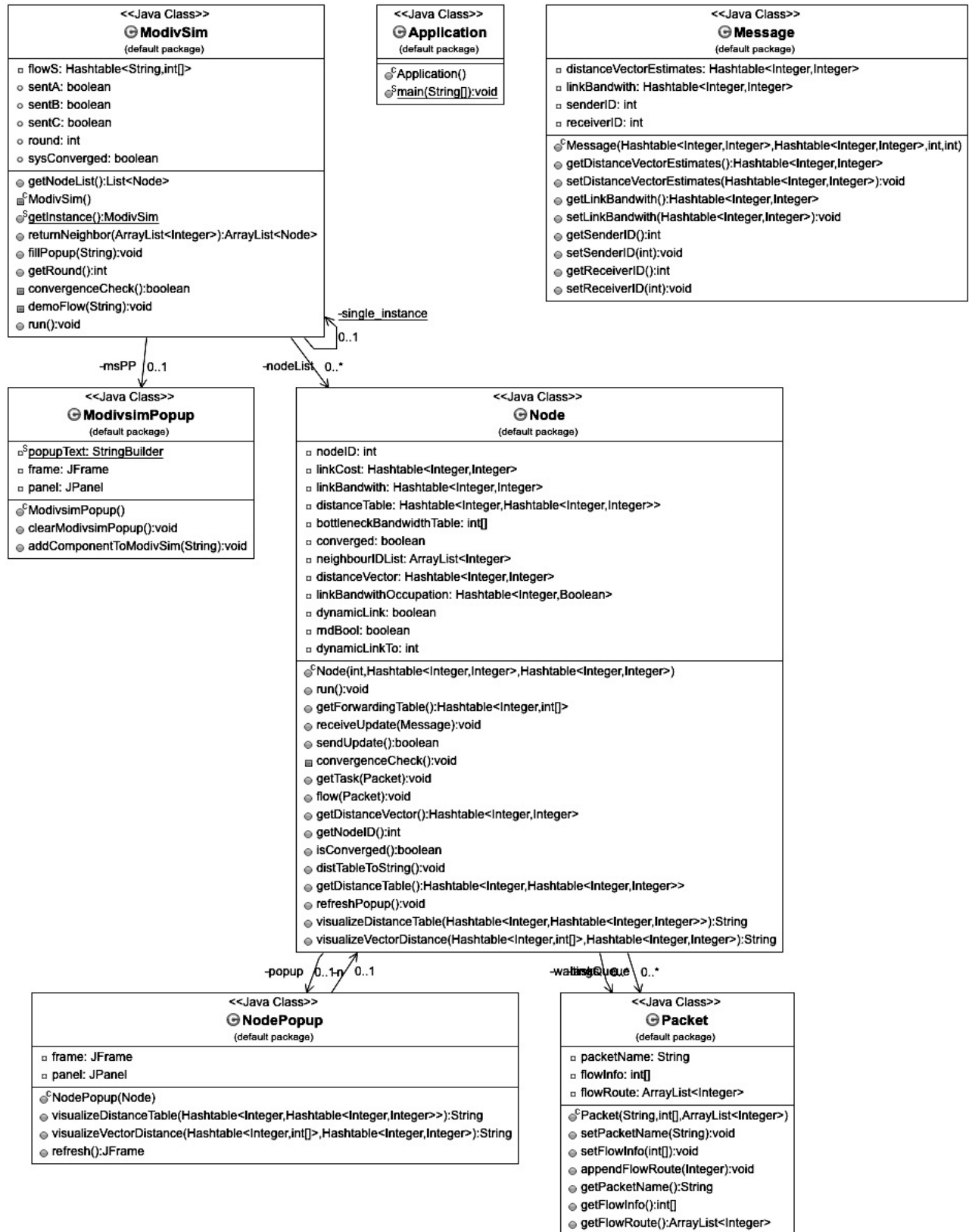


Figure 5

Performance Results:

The performance of the experiment can be analyzed in three steps: Static Link Cost Scenario, Dynamic Link Cost Scenario and Flow Routing. Those results will be interpreted by comparison of the theoretical and practical outcomes.

Static Link Cost Scenario:

Static Link Cost is the main component of the simulation. To have other scenarios adopted, static link should be observed. Basically, in static link, all the costs are predefined and given to the simulation through the input.txt. Sample topology of the instructions is used in this process. Followingly, it is observed that it took 3 rounds to converge. Simulating the rounds by hand, it is checked that the number of rounds are correct. Followingly, if compare the paths with the given paths of the instructions, the results are double checked by means of number of rounds to converge, the finalized distance tables, distance vectors and forwarding tables.

Dynamic Link Cost Scenario:

Addition to the predefined cost of the links, an unstable link cost is implemented in that scenario. The main mechanism is as follows: if a link is defined as dynamic, by giving it a value of 'x', random value between 1-10 in the first run. Followingly, an entropy is implemented as a Boolean. With that entropy, that dynamic cost is either randomized again or not. That re-randomization part is essential since it makes convergence unpredictable. Through the experiment, different numbers of dynamic links are tested and the number of rounds to the convergence is noted. Following graph represents the relation of the rounds and dynamic link number.

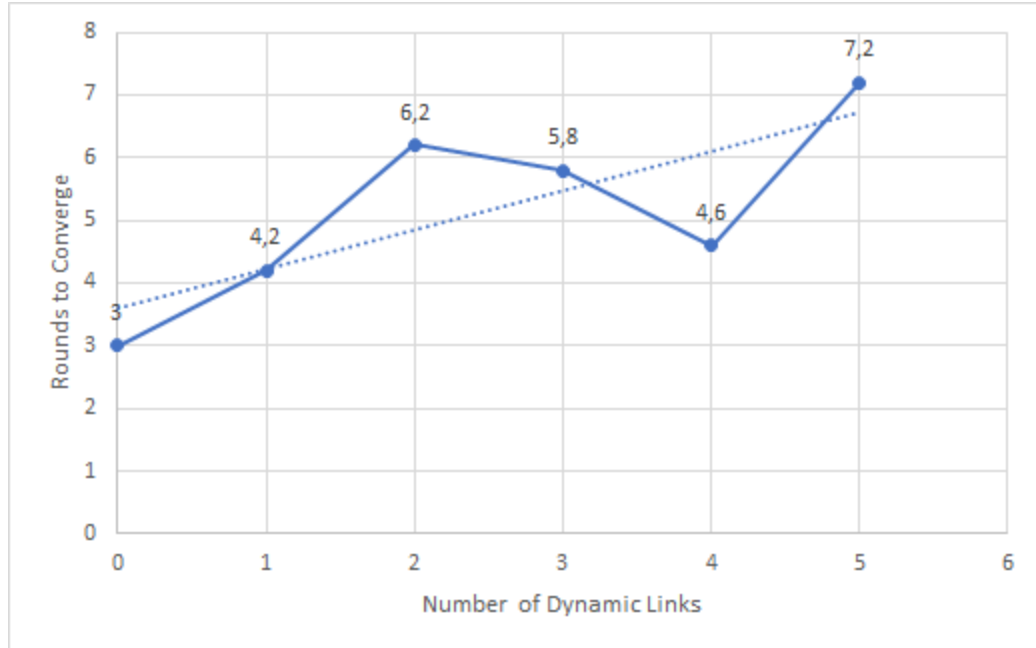


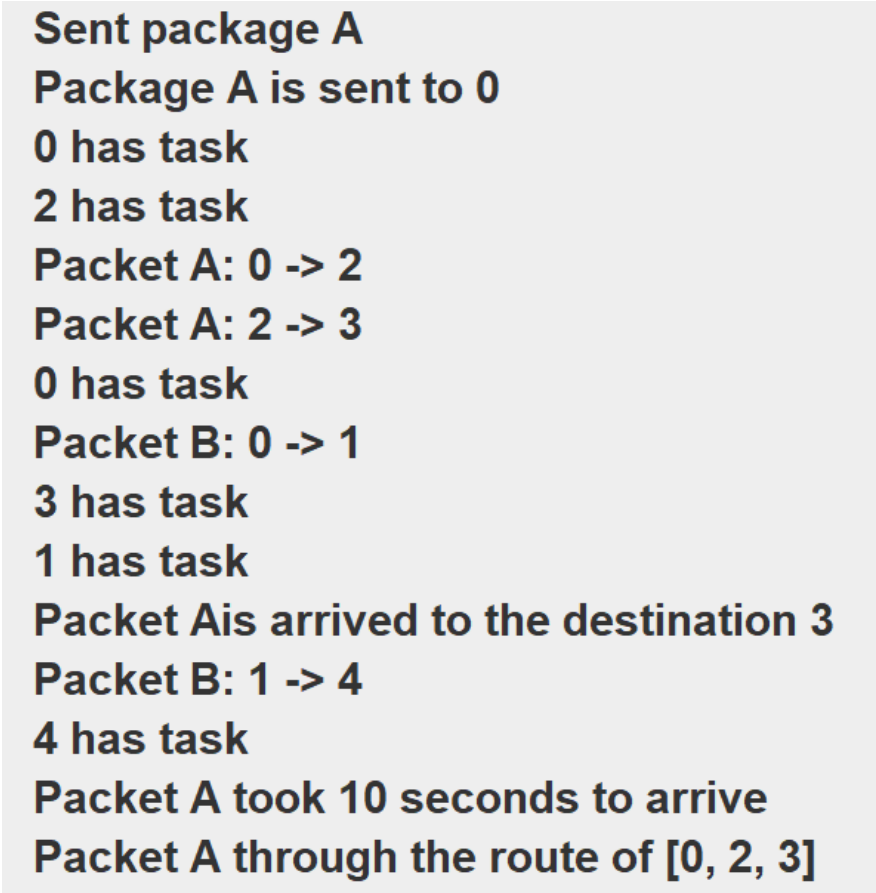
Figure 6

From the figure above, the relation between dynamic links and rounds taken for convergence is shown. To have more reliable results and the randomization of those costs, 5 experiments have been done for each number of links. However, unlikely to the expectations, the relationship seemed nonlinear. One can see that the number of rounds decreased from 2 to 4 dynamic links. In the first run, it can be said that that is because of the randomness of the initialization of the costs. In addition to that, place of the dynamic links are also important. It is observed that, assigning dynamic cost to link that have least predefined cost results in dramatic increase in the number of rounds. That is because, the least cost links are the main subject in the Distance Vector Routing algorithm.

Flow Routing Scenario:

The predecessor scenarios were channelizing to the convergence in the Distance Vector Algorithm implementation. However, in the Flow Routing Scenario, the packet flows are tested in the converged version of the Static Link Cost. The sample flows of the instructions are used in this part of the experiment.

The first flow was A,0,3,100 means that packet A, with source 0, size 100 Mb wanted to be sent to the node 3. For this scenario, the packet used the route of 0->2->3 and according to the bottleneck bandwidth, it took 10 seconds. While comparing it with the theoretical results, it can be said that forwarding tables are worked properly and traceroute, and time is consistent with the instructions. From the figure below, one can see the sample snapshot from the Simulator window which shows packet A's traceroute and related information such as hops, time elapsed to arrive to the destination.



```
Sent package A  
Package A is sent to 0  
0 has task  
2 has task  
Packet A: 0 -> 2  
Packet A: 2 -> 3  
0 has task  
Packet B: 0 -> 1  
3 has task  
1 has task  
Packet A is arrived to the destination 3  
Packet B: 1 -> 4  
4 has task  
Packet A took 10 seconds to arrive  
Packet A through the route of [0, 2, 3]
```

Figure 7

The second flow was B,0,3,200 means that packet B, with source 0, size 200 Mb wanted to be sent to the node 3. Within this scenario, the second-best alternative approach of the forwarding table is tested. While having an occupation mechanism for the links, packet B used the route of 0->1->4->3 and it took 40 seconds, since the 0->2 link is occupied by the link. Again, this experiment proved that the distance vectors are

calculated properly and forwarding tables are used efficiently. One can see that packet B used another route since the link between the 0 and 2 is occupied by the packet A from the below figure.

Packet B: 0 -> 1
3 has task
1 has task
Packet A is arrived to the destination 3
Packet B: 1 -> 4
4 has task
Packet A took 10 seconds to arrive
Packet A through the route of [0, 2, 3]
3 has task
Packet B: 4 -> 3
Packet B is arrived to the destination 3
Packet B took 40 seconds to arrive
Packet B through the route of [0, 1, 4, 3]

Figure 8

In the last part of the flow scenarios, flow defined as C,1,2,100 is sent to the source 1. After transmitting packet to the node 0, the packet is sent to the waiting queue since both of the alternatives are occupied. While observing the first desired measure in this part, link release after the packet transfer was also tested. From the outputs, it is seen that after 10 seconds, which transmission time of packet A, packet C is extracted from the waiting queue and sent to node 2. The link operations are tested in this part and second desired measure, the releasement of link, is seen as correctly implemented.

As it can be seen from the above figure, packet C hop to router 0 from router 1. Followingly, packet pushed to the queue. System is designed to prompt if there is any packet in the queue.

```
1 has task  
Packet C: 1 -> 0  
0 has task  
0 has task  
C is in the queue in router 0
```

Figure 9

Followingly, the simulator also prompts the occupation of the links and signals if there is any releasement. After the releasement, the flow of the C can be seen from the figure below.

```
Link between 0 and 1is not occupied now  
Link between 0 and 2is not occupied now  
Link between 1 and 4is not occupied now  
C is in the queue in router 0  
Link between 2 and 3is not occupied now  
Packet C: 0 -> 2  
Link between 4 and 3is not occupied now  
2 has task  
Packet Cis arrived to the destination 2  
Packet C took 10 seconds to arrive  
Packet C through the route of [1, 0, 2]
```

Figure 10

External Libraries:

As it is expected to visualize each update state of the nodes, an additional library should be used. At the first stage, Java's own GUI library is chosen for this purpose. However, it was inadequate. JLabel class was used to display each component in the pop up JFrame, but it is not possible to pass to the next line in the JLabel object, it is restricted with a usage of single line. So, the need for a new class was born and this class was not in the Java Swing library. The custom version of the Java Swing library is examined which is Java SwingX. It brought the feature, which is demanded, multiline JLabel. So, SwingX library is used as an external Java library.

Conclusion:

In this experiment, a modified version of network simulation based on distance vector routing algorithm is designed. Within the experiment, a decentralized approach of routing, Distance Vector Routing algorithm, is implemented to determine the routing. Periodic updates, additional information about the bandwidth of links and second-best alternative in the forwarding table is modification that is done on the simulator. Throughout the experiment, how convergence is done is observed and followingly, packet flows are tested in the predefined topology. By having nodes as threads, autonomous of flow events and reason behind the decentralization signified. Followingly, a dynamic link also tested to see the race to the convergence. In the last part of the experiment, the packet flow, simple packet transfer and transfer effort on the occupied links are observed. As a result of the experiment, network simulation was done by internalizing the main functionalities of Distance Vector Algorithm while having additional features on it.

Task Distribution:

Emir Atısay: The implementation of Distance Vector Routing Algorithm within ModivSim class, Implementation of flow routing.

Ege Berk Süzgen: Reading and processing the inputs, implementing Node and Message classes, Designing of additional classes.

Onur Şahin: Test of ModivSim, graph and reporting.