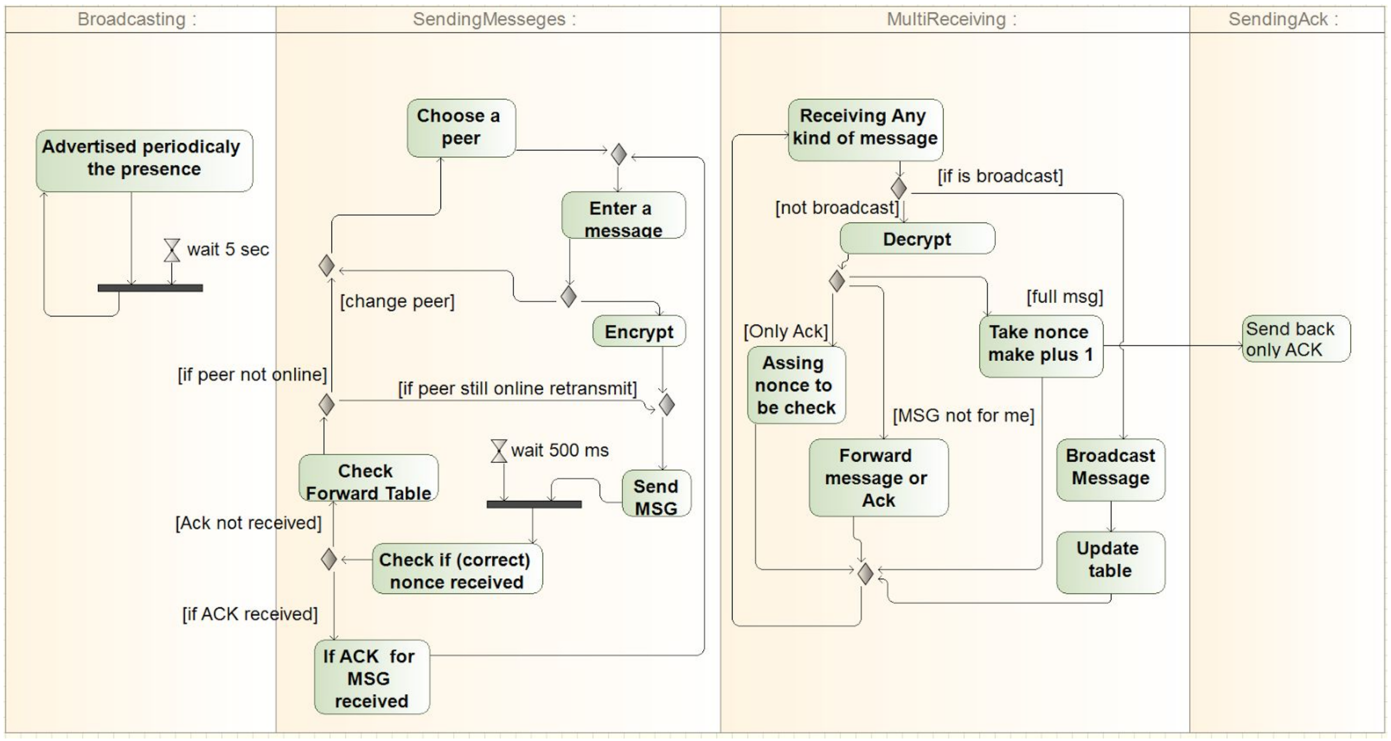
# Network Systems Final Project Report

**Yannis Linardos (s1866303) - Christodoulos Tziampazis (s1829963) - Bilge Tekes (s1797131)**

## ***System architecture and diagram***

**Note that the following diagram is not a normal activity diagram, it only represents our system's architecture. As you can see there is no initial or final nodes, thus we only show the structure of the 4 more important thread classes we use( the main class Run which run all classes is not included) . Everything will be explained in a more analytical way below.*

In order to start the application, the class Run should be executed, in which each user must indicate its last digit of its Ip address. Our system does not include packet fragmentation so one may send maximum 56 KB of data. This will not cause any problems to the end user as the application is designed to only transfer text messages, not pictures/videos/audio files.



## ***Description of implemented system functionalities***

**Note that all classes are marked with red color.*

### **Classes explanation**

**Main class: Run** = This is the class that contains the executable 'main' method. In the constructor we create the GUI and we create and start the three threads that follow. In the 'main' method we just call the constructor and the application starts.

**Thread SendingMessages** = Responsible for constructing and sending ONLY messages.Also,it retransmit messages and redirect(if needed) them based on the forwarding table.

**Thread MultiReceiving** = Responsible for receiving any kind of message(either advertising broadcast or a whole message or an Ack message).

**Thread SendingAck** = Responsible for constructing and sending ONLY Ack messages. Also it can redirect the Ack based on the forwarding table

**Thread Broadcasting** = Responsible for broadcasting periodically (every 5 sec) the forward table of each node.

**ForwardTable** = The data structure in which we save the forward tables. It also contains methods for adding, removing and replacing entries and for encoding and decoding the table into bytes so that it can be broadcasted.

**Route** = A simple class which only contains two fields, the InetAddress of the 'nextHop' to whom the node must forward the packet and the 'cost' in distance to the final destination.
**Encryption** = It contains a method to encrypt and a method to decrypt Strings (both static). It applies a simple XOR encryption with a key we randomly generated.

**Note that for Message Format Explanation we only explain the 3 possible messages and not the actual variables we use in the code. However, in the below descriptions of system functionalities variables are also indicated with red color.*

## Message Format Explanation

**Message** = A message is consist of Source(Sender), Destination(Receiver), Nonce(Random Number), Message(Text). The message is splitted with a semicolon ' ; ' into 4 parts individually and the form looks as follows : e.g [ 1;2;2225;HelloWorld ].
**AckMessage** = The acknowledgment message is consist with Source(Sender), Destination(but this time the destination is the actual sender of the Message) and the Acknowledged Nonce(nonce +1). This message is also splitted with semicolon ';' in 3 parts individually and looks like this : e.g [ 2;1;2226 ].
**Broadcast Message** = The broadcasted messages start with "@@@@@" so that we can differentiate them from regular messages, followed by ";;;". Then, we append the forward table entries in the following form: destination;nextHop;cost. Different entries are separated by ";;;". E.g. : [@@@@@;;;1;1;1;;;2;1;2;;;]

## Description of implemented system functionalities

### Ad-hoc multi-hop routing

The multi-hop routing is based on a distance vector algorithm. According to the algorithm, each node has a forward table in which it saves its distance to every other node in the network as well as to whom of its immediate neighbours ('nextHop') it should forward a packet in order to reach its final destination. In our case, we consider the distance between neighbours to be 1, so in the end the distance in the forward table implies the number of intermediate nodes between the source and the destination.
The forward table is implemented by the class 'ForwardTable class'. This is achieved by using a Map with keySet the InetAddress of the destination and values the Route to the destination. Every 5 seconds, each node broadcasts its table to its immediate neighbours using the thread 'Broadcasting Thread class' which encodes the forward table into bytes and sends the DatagramPacket to the multicast address 228.0.0.0, that everyone is listening.
In addition, each node receives the forward tables of its immediate neighbours. In the thread 'MultiReceiving Thread class', packets that contain the forward tables of the neighbours are taken care of by the method 'dealWithBroadcast'. This method compares the received forward table with its saved one and makes appropriate changes. If its current saved table did not contain a node of the received table, then the new node is added to the table with 'nextHop' the neighbor that sent this table and the 'cost' incremented by 1. If the node was contained in the table, then it checks if the new 'Route' has smaller cost and updates the entry accordingly.
In addition, each node contains a Map 'neighbours' which saves as keySet its neighbours and for values the last time it received broadcasted message from them. Every neighbour is automatically added to the forward table with 'nextHop' the neighbour itself and 'cost' 1. If a node does not receive a broadcasted message from a neighbour for 10 seconds, then it removes it from its Map 'neighbours' and the forward table.
To prevent the count to infinity problem, we created a 'MAXHOPS' integer field. When the 'cost' to a destination becomes 'MAXHOPS', we consider it unreachable. In our case, for three nodes the worst case scenario dictates to set 'MAXHOPS' to 2, but for four nodes it would be three.

**Reliable text messaging**

Reliable chat communication is based on acknowledgments. We decide that it will be better to use 'nonce' , a random integer number from 0 to 5000 that will be chosen randomly for each message except for retransmissions. If a user decides to send a message, a random number is generated and is assigned to variable 'num1'. However , the next random, nonce, can not be the same since we add a check that ensures the new generated nonce is not equal to the previous one.

On the sender side, after the nonce is generated, in the 'SendingMessages Thread class' a 'String msg' is assigned with the message format as explained above(Message) and a DatagramPacket variable called 'frstpacket' is created. Note that each message is encrypted before we assign it to 'frstpacket'(see Security) .In 'frstpacket' the destination is set by using the forwarding table to find the 'nextHop'. After that, the sender increases the nonce by 1 and saves it into the HashMap<Integer , DatagramPacket> called 'allpktsend' as a key and the DatagramPacket as the value(use of HashMap explained below).

On the receiver side there are 4 cases:

**1.** When 'MultiReceiving Thread class' receives a whole 'Message',which is 4 parts long (see 'Message'): First, it decrypts the message and splits it into parts as explained above. Then, it displays the message on the GUI and at the same time it increases the nonce number by 1. To acknowledge the message we create an instance of 'SendingAck Thread class' inside 'MultiReceiving Thread class' and we pass the nonce to its constructor so that the class can use it to create an 'AckMessage'(see above). The process is the same as when we send a message except that we construct only an 'AckMessage' without a text.

**2.** When a 'Message' arrives and is not destined for us then we simply check if the destination of the message is in our forward table , if it is we forward the message to the 'nextHop'.

**3.** When we have already sent a message and we expect to receive an 'AckMessage', which is 3 parts long (see 'AckMessage'): In that case we use a static variable 'ackIdent' and by using setters and getters we pass the value of the acknowledged nonce that we received to the 'SendingMessages Thread class' so that we can ensure that message was successfully transmitted to its destination. In that case the HashMap 'allpktsend' is used. After sending a message we wait for 500 ms to receive the ack, after that timeout the 'ackIdent' is used to loop through the HashMap and check if the ack received is indeed the nonce we saved. If that is not the case then it retransmits the same packet until the application realise that the peer is unreachable by checking in each loop the routing table.

**4.** When we receive a Broadcast Message: it is explained above in multi-hop routing.

**Graphical User Interface**

The GUI is rather simple. It consists of a TextArea that displays the sent and received messages, a TextField in which the user can type his/her message, a JComboBox with which the user can choose with whom to chat (it only contains the nodes that are reachable according to the forward table), a send button and a clear button that clears the TextArea.

**Security**

All messages (apart from the broadcasted ones) are encrypted before sending and decrypted upon received. We use a hardcoded key that we generated randomly to XOR the bytes before sending them and then the receiver will XOR it again when it receives it.

## ***System Testing and Results***

### Test for Forward Table Broadcasting Functionality

**Test Scenarios and expected results:** To test whether our implementation of the distance vector algorithm works, we created three test scenarios with increasing complexity.

- Test with two nodes in the network: We started the application on one computer and a few moments later another node did the same. We expected that the nodes will add each other to their forward tables. After a while, we disconnected one node and the expected behavior was that the other node will remove the disconnected node from its table

● Test with three nodes in the network: We repeated the previous setting with three nodes instead of two. When each node was connected, the other nodes should create an entry in their tables and the new node should add the two existing ones in its table. When one node disconnects, we should see the count to infinity problem manifest between the other two nodes until they reach a maximum value that was set in advance. Then, the other nodes consider the disconnected node unreachable.

● Test with three nodes that are not all connected with each other: We have two nodes connected as in the first setting. Then, a new node starts the application in a distance such that it can immediately reach only one of the other two nodes. Then, we should see first that the "node in the middle" adds the new node in its table and in the next transmission, the third node should it to its table too using as 'nextHop' the "node in the middle". When one of the two nodes in the edges disconnects, the count to infinity problem manifests itself until it reaches the set maximum value.

**Results:** The count to infinity problem caused most difficulties in this section but we solved it as described. Another problem that cause frustration was what data structure to use and how to encode the data. In the end we created our own data structure as described in the class 'ForwardTable' and we encoded by first converting it into a string and separating the rows of the table with ';;;'. After some trial and error we achieved to get the expected results.

## Test for Reliable Data Transmission Functionality

**Test Scenarios and expected results:** To test our implementation for reliable data and possible retransmission we test our code in 2 cases.

● Test with 2 nodes: With 2 nodes we try to exchange some messages while the other node is offline. The results we wanted to have was to receive the message when the offline node come back online before the limit of 10 seconds( which remove the node from the routing table ).

● Test with 3 nodes: With 3 nodes we try to test whether the above case with 2 nodes is working when we have 3 nodes which node 1 and node 3 cannot reach directly each other by having the node 2 to redirect their messages. The results again is the same as with 2 nodes.

**Results:** The results we achieve were the desirable as we received all the message in both test without any loss. A common problem we had was caused by the fact that we separated the packet with semicolons (;). This caused problems when a semicolon was included in the text but we solved it using loops and string manipulation. We also encountered the really improbable situation that the randomly generated Ack number was the same for two consecutive packets and we solved by checking.

## Overall System Testing

**Test Scenarios and expected results:** To test the overall system we made two test scenarios.

- Test with 2 nodes: When two nodes start the application in a network, they should find each other and they should appear in each other's GUI JComboBox so that the user can choose to text them. After choosing the destination, a user can type a message in the TextField and press the "send" button. The message should then appear in the TextArea of the source in the form: "To" + destination + ": " + message. It should also appear in the TextArea of the destination in the form: "From" + source + ": " + message. If a node disconnects, its address should disappear from the other's node JComboBox.
- Test with 3 nodes in the DEMO setting: After starting the application, the middle node should be able to see both edge nodes on its JComboBox while the edge nodes should only be able to see the middle node. 5 seconds later, after the second transmission of the middle node's forward table, the edge nodes should be able to see each other in their respective JComboBoxes. Then, each user should be able to choose an address from his/her GUI and type a message to the TextField. After pressing "send", the destination should see the message in his/her TextArea. This should be possible for all possible pairs of nodes.

**Results:** Most problems were solved while checking each functionality separately. When we integrated the parts, we only encountered minor problems that we solved easily.