Network System - Integration Project Report - Group #54

Group members:

- Konstantin Dichev, s2316714
- Kristian Nedelchew, s2343800

1. System overview

Our system works based on finding the neighbors – waiting for the network to converge. Then sending a packet supported by fragmentation if it exceeds the allowed 32-byte allocation in the ByteBuffer. The application checks the neighbors based on an EchoThread that we implemented. The Echo Thread constantly sends messages to the neighbors. That way, we discover all the nodes that can be reached and each node stores information about its neighbors in a table where each of the entries has TTL property. After that, the network converges and once it has converged even if a node is not in the range of the origin of the packet it will still receive it given that it is in the range of a node that is adjacent to the origin. This is the way the multi-hop we implemented works. We also started implementing RDT mechanism. We were close to having it completed but we experienced bugs associated with the synchronization in concurrent programming that led to issues that were difficult to follow and debug. Although our code made complete sense, we were getting weird effects from the program. The idea behind our code was that a node can acknowledge newly received packets to the sending node and transmit them to the following nodes in one go. Therefore, we use the network bandwidth efficiently.

2. Description of the system functionalities and/or protocols

2.1. Medium access control

The medium access control mechanism we implemented is based on putting the thread to sleep when the link is busy and keeping the thread asleep while the link is busy. We use a random roll of a relatively large number for the value of the Thread.sleep() to make sure that the threads do not transmit at the same time. That way, we avoid collisions as much as possible (no 100% collision detection is possible, or at least known). Additionally, we implemented a method to detect collision for the very distant nodes. (We noticed that in the line topology the first and the last node can start transmitting at the same time because they cannot define that the node at the other end has started transmitting)

2.2. Reliable broadcasting

a node sends a packet it makes a record about it and determines all of the neighbors that are expected to acknowledge this packet.
the arrival of a packet that has not been received before, the nodes send an acknowledgment, and if necessary, further transmit it and obviously determines if it needs to receive acknowledgments for it. If after some time none of the expected acknowledgments are received it re-transmits them. However, we could not finish this part of the program because we had no time to fix the errors associated with the synchronization of all the threads we had running. (after conducting multiple tests we came to the conclusion that the issue is very likely to be associated with concurrency).

2.3. Multi-hop forwarding

In order to achieve the multi-hop forwarding we needed knowledge of all the possible neighbors in the given topology. We figured out that adding a thread that constantly sends echo packets to track the neighbors was a clever way of dealing with that matter. We had an issue with the HashMaps that we introduced as our way of storing data. After a long session with a TA, we decided that it would be better to use the BlockingQueues that were provided by the framework because that way we needn't worry about the concurrency confusion. Every node keeps track of all of the received messages and prints out only those that were not previously received.

2.4. Fragmentation

implemented a fragmentation mechanism for the packets in order to comply with the buffer allocation of 32 bytes. We had to take into account a header that we needed for the other functionalities that consisted of 5 bytes (data.get(0) – the transmitter of the message, data.get(1) – the type of the message(if it is an echo, a normal, an ack message or both normal and ack), data.get(2) – the seq number, data.get(3) – the origin of the message, data.get(4) – the length of the message). Furthermore, we added a way of preventing packets from arriving out of order at the end host.

3. Testing

During the implementation, we conducted multiple system tests by constantly adding System.out.println(…) in methods that had high complexity. That helped us see what exactly happens in one iteration and which conditions are met. For example, the run() method of the messageHandler thread is of high complexity. Followed by the isReceived() method we needed to check what exactly happened in order to figure out the bugs. We were stuck on the messageHandler issues for a long time because initially, we did not take into account the problems that we might face with the introduction of additional threads to the protocol. After all, it was difficult to synchronize all the parts.
Furthermore, we ended up with a couple of versions of our code. To check which one is resistant to exceptions and bugs, we had to run them through all the possible topologies and send long messages multiple times for each topology.