Robert Weischedel

3/27/16

CS 4480

PA2 – A Assignment Report

Design

Since we were given much of the source code for this project, the majority of the work was done in StudentNetworkSimulator class, and with only one line added to the project class to print the statistics of the desired transmission. In the StudentNetworkSimulator class there were seven methods that needed to be implemented, they were aOutput(), aInput(), aTimerinterrupt(), aInit(), bInput, bInit(), and viewStatistics(). It should be noted that for each of the methods that required a checksum value to be created or evaluated that I did so by combining all the checksum data into one string and then iterating through the string, parsing each character into its ASCII value and then adding up all those values to be the checksum value for that particular packet.

aOutput()

The aOutput() method takes the messages sent from layer 5 and prepares them to be sent to layer 3. In my implementation I first keep track of the amount of messages that are sent from layer 5 that need to be sent to layer 3. I do this not only to keep track of the total messages that have been sent from layer 5, but to also keep track of how many packets are still on the wire in transit at the time the program ends. Next I check if the message has been acked or not, if it has been that I go ahead and create the message packet and send it I do this by first creating a checksum value for the packet, and then by using that value and all the other relevant data to create a message packet. But before the packet is sent I set the acked value to false, since this new message has not yet been acknowledged by B. Then I save a copy of the message just in case the packet is lost or corrupted in transit, so that I can be resent again according to the specifications of the Alternating-Bit Protocol. And finally, I increment the messages sent variable and then I start the time out timer and then send on the message to layer 3. If the next a message is sent, and the acked value is false, then a message will appear in the console window stating that the message was ignored because the previous message was still in transit.

aInput()

This method takes messages from the wire and brings them up to level 3 for processing in order to be sent back up to layer 5. In my implementation I first re-create and verify the checksum value to ensure that it matches the packets checksum value. If they match, and if the sequence numbers match as well the packet is “accepted” by first stopping the timer and then by resenting the sequence value of A to 0 or 1, based on the previous sequence value in accordance to the rules of the Alternating-Bit Protocol. And then finally the acked count is incremented, so I can keep statistics of packets that have completed the round trip. If the checksum doesn’t match, then the corrupted packet count is incremented.

aTimerInterrupt()

This method is called when the timer times out and the packet still hasn’t made it to B. Just like the Alternating-Bit Protocol requirements, this method will resend the message that was just lost in transit. My implementation performs this task by first re-creating the checksum value of the lost message, and using that value and all other necessary information to create a new message packet. Then I restart the transmission timer, increment the resent messages variable, and then send the packet to layer 3 once more. I keep track of the resent messages variable in order to keep track of total statistics of the entire transmission.

aInit()

This method was the second easiest method that was implemented. This method is one of the two starter methods that are called only once, and begin preparations for communication from A to B. This method takes care of the initial set up of variables in A. It sets the acked value to true, and the sequence valie of A, the total message count, the total messages sent, the acks count, the corrupted messages count, and the total resent messages count all to zero.

bInput()

This method takes the packet from layer 3 and ensures that it is correct and then forwards that information onto layer 5. It also creates an ack message in order for A to know that the message was received. My implementation does this by first re-creating the checksum value of the packets information and ensuring that it is the same as the packets checksum value. If it is, then I check and ensure if the packets sequence number is also correct, and if it is also correct then I reset the expected sequence number of B to the opposite of the packets sequence number, according to the Alternating-Bit Protocol. I also print out a message to state if the message data was not what was expected, this was primarily just used for testing purposes. And then finally I send the message up to layer 5 and then create a new empty message packet with the ack for A to affirm that the message was received by B.

bInit()

This method is easily the simplest method that was implemented, and it is similar to that of aInit() because it is called only once in order to initialize the setup of communication from A to B. In my implementation I simply just set the expected sequence value of B to zero, in preparation of the first package, which will have a sequence value of 0.

viewStatistics()

This method is used to print out the statistics of the current transmission session. It consists of a series of print statements that prints session information to the user from the console window. All the implementation is done in the StudentNetworkSimulator class, but the method is called from the project class.

Testing

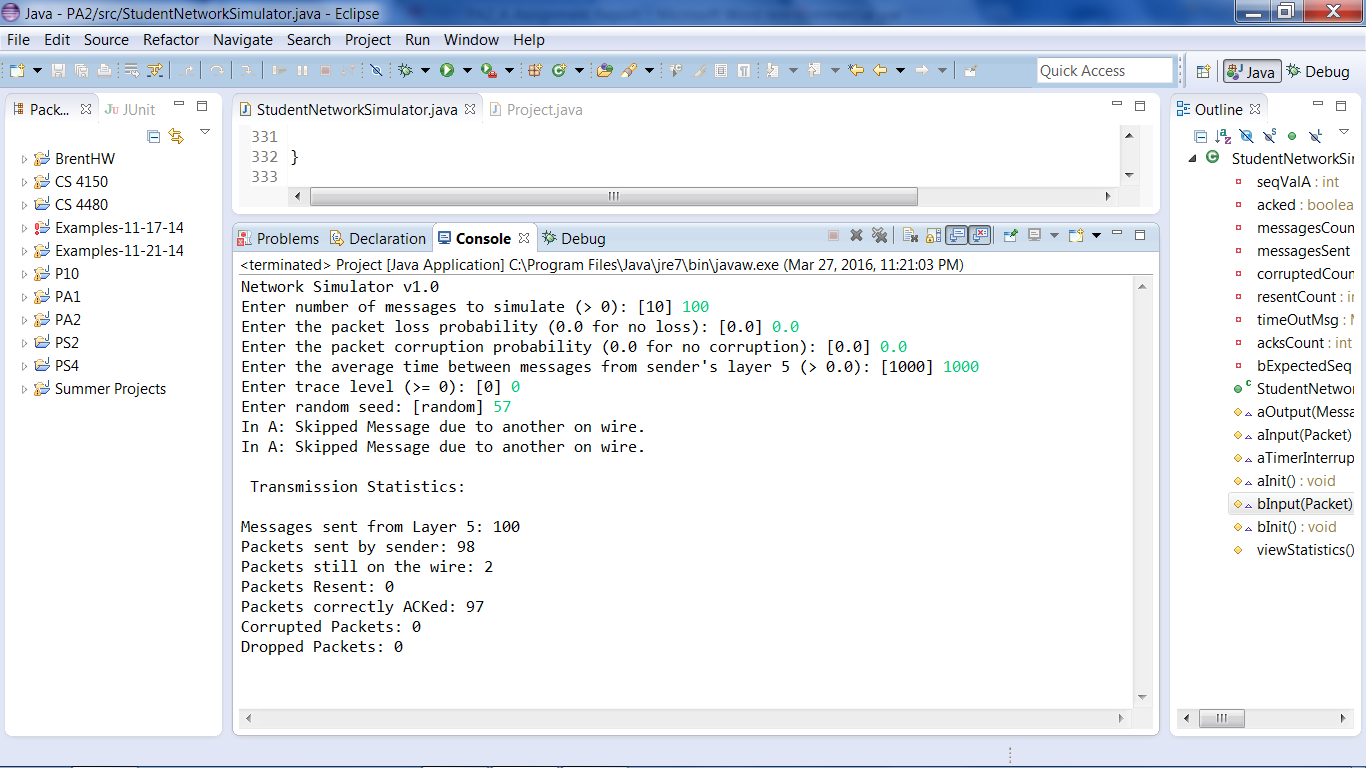
In order to assure that the program was working correctly I performed eight different tests on two different message size values. These six tests would consist of changing both the corruption and lost message values to 0.0 for the first test, 0.02 for the second test, 0.05 for the third test, and 0.10 for the fourth test. The fifth and sixth test will consist of a corruption and lost packet value being different. For the fifth test the corruption test would be 0.05 and the lost value would be 0.10, and the sixth test would be the same as the fifth test except with the values switched. The final two tests will consist of the either the corruption or the loss value be 0.1 while the other would be 0.0. It should be noted that all the other options will be run at the same consistent values in order to ensure test accuracy. The random seed value will be 57, the average packet time will be 1000 and the trace level will be 0. My first series of tests will consist of performing tests on a message sending size of 100 messages, and my second series of test will consist of performing tests on a message sending size of 1000 messages.

After performing all the tests I determined that the program was working as expected. The values that were returned were all roughly what the expected output should have been. The corrupted packet value tended to be a bit low or high by about a half of a percent or so. While the dropped packet value tended to be truer with less deviation. But all tests were successful in proving that within reason that the program performs as it should.

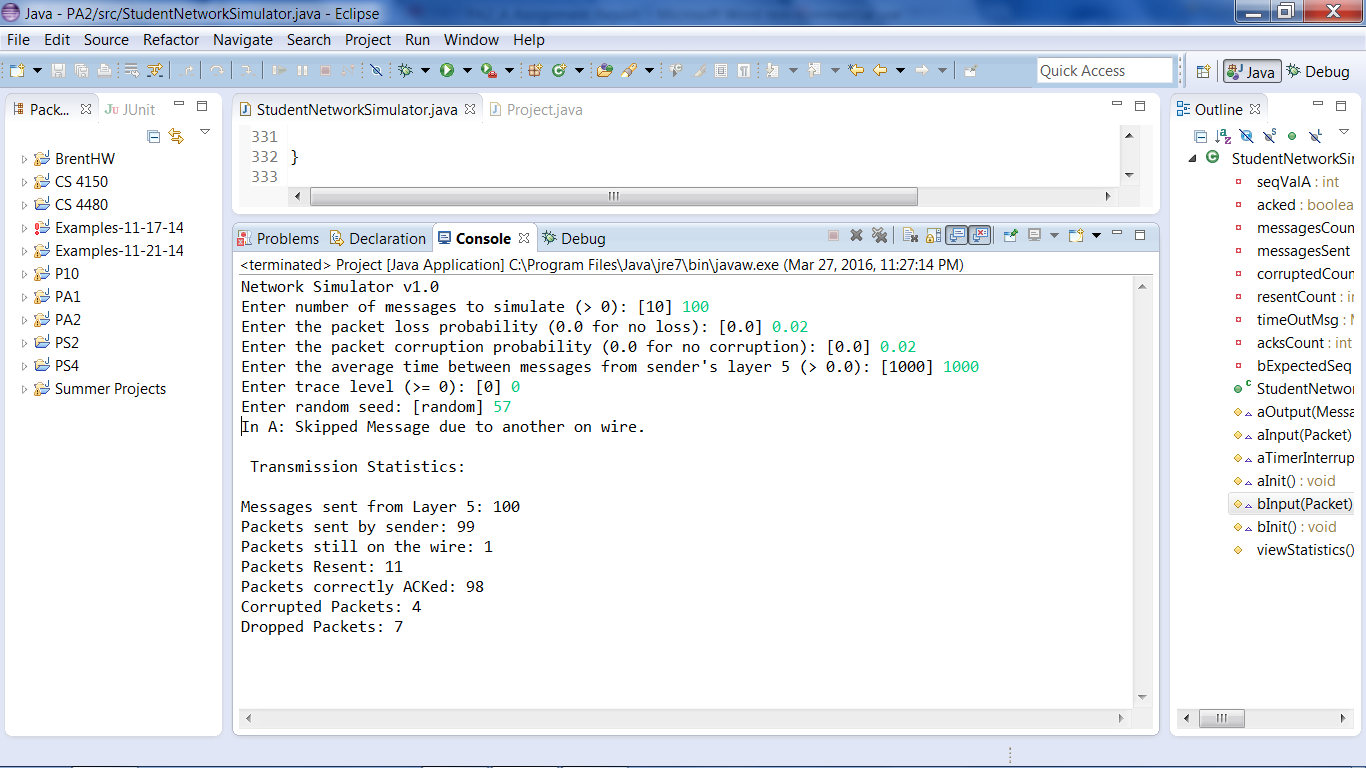
Output

100 Messages Sent

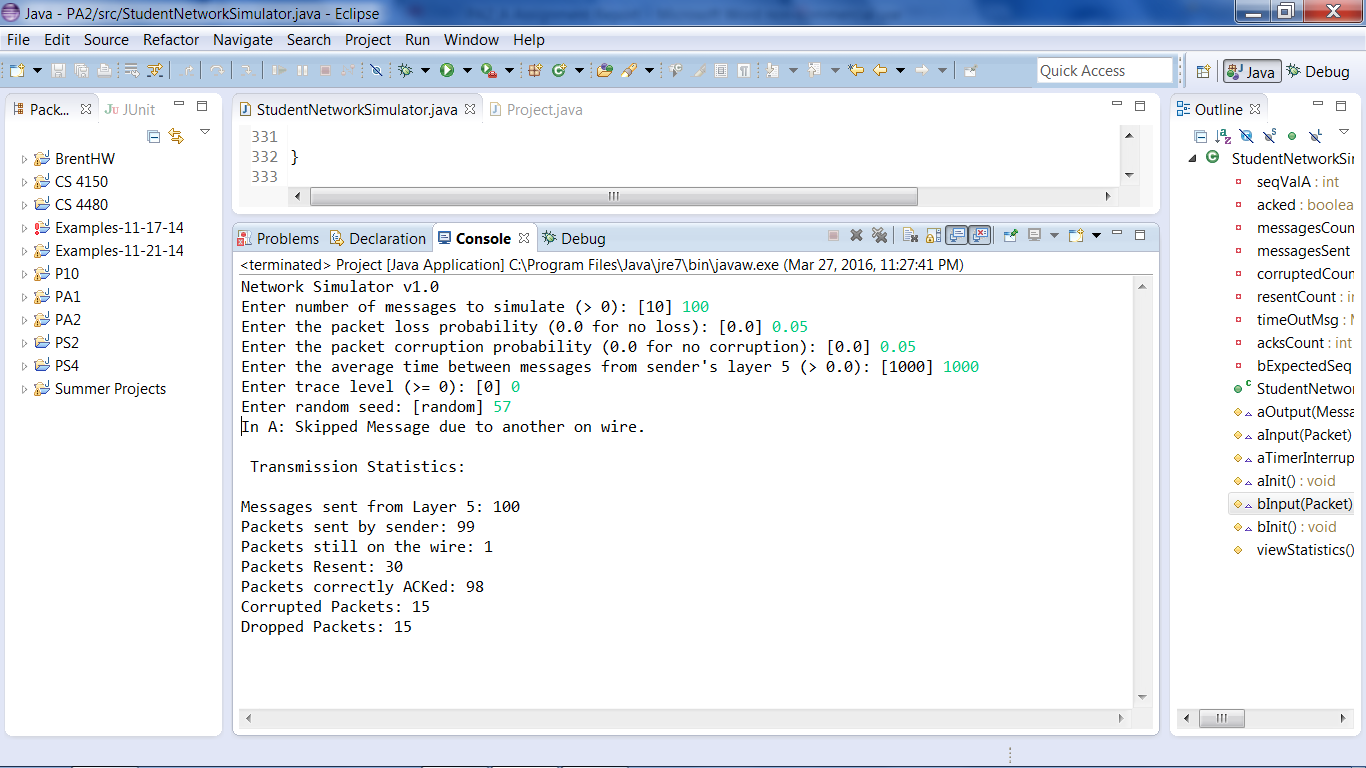
Corruption – 0.0, Loss – 0.0



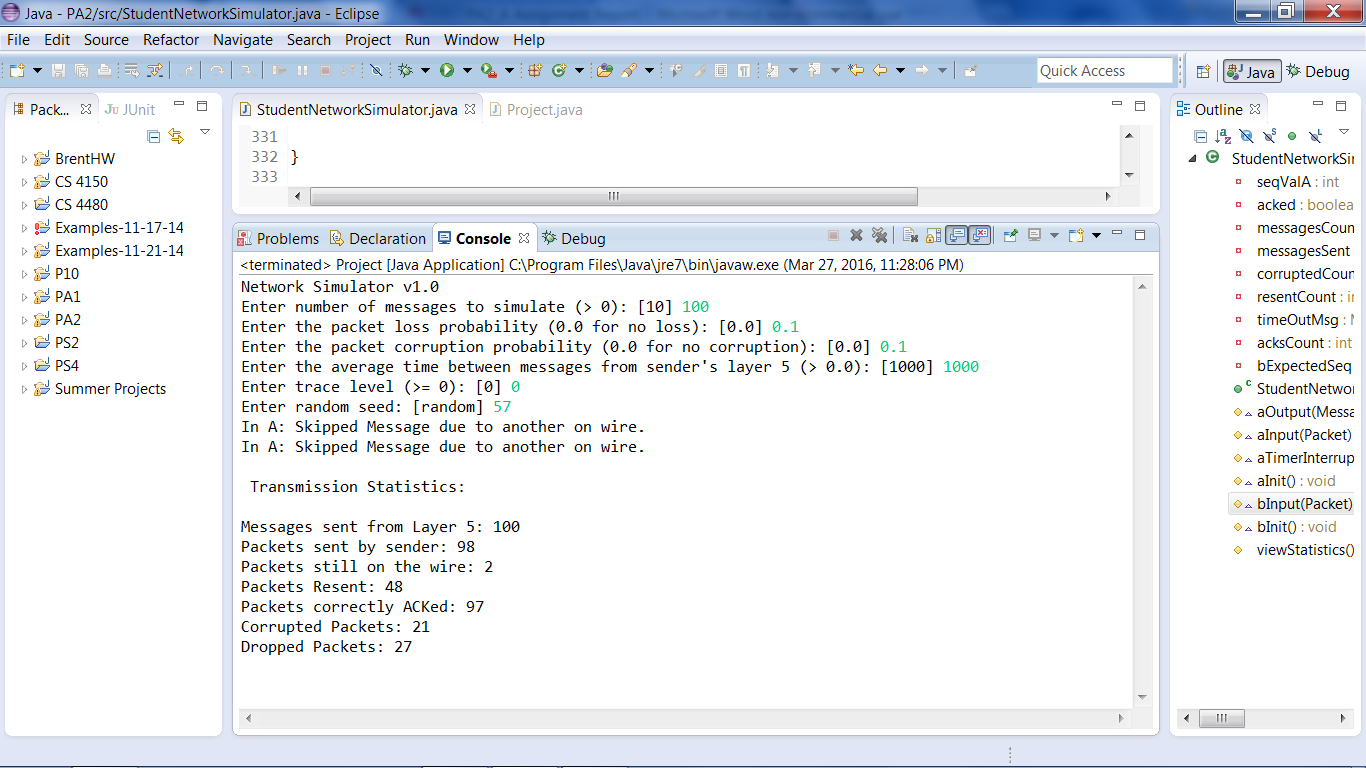
Corruption – 0.02, Loss – 0.02



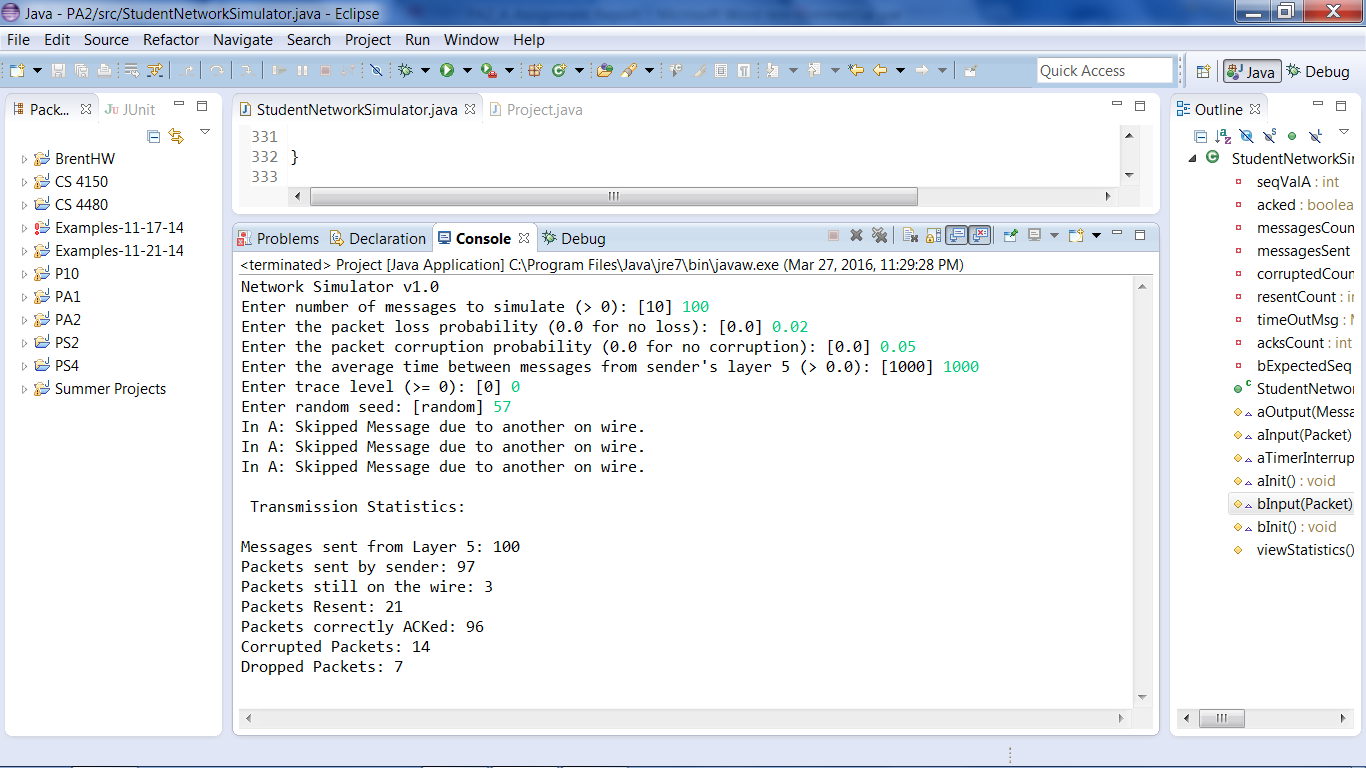
Corruption – 0.05, Loss – 0.05



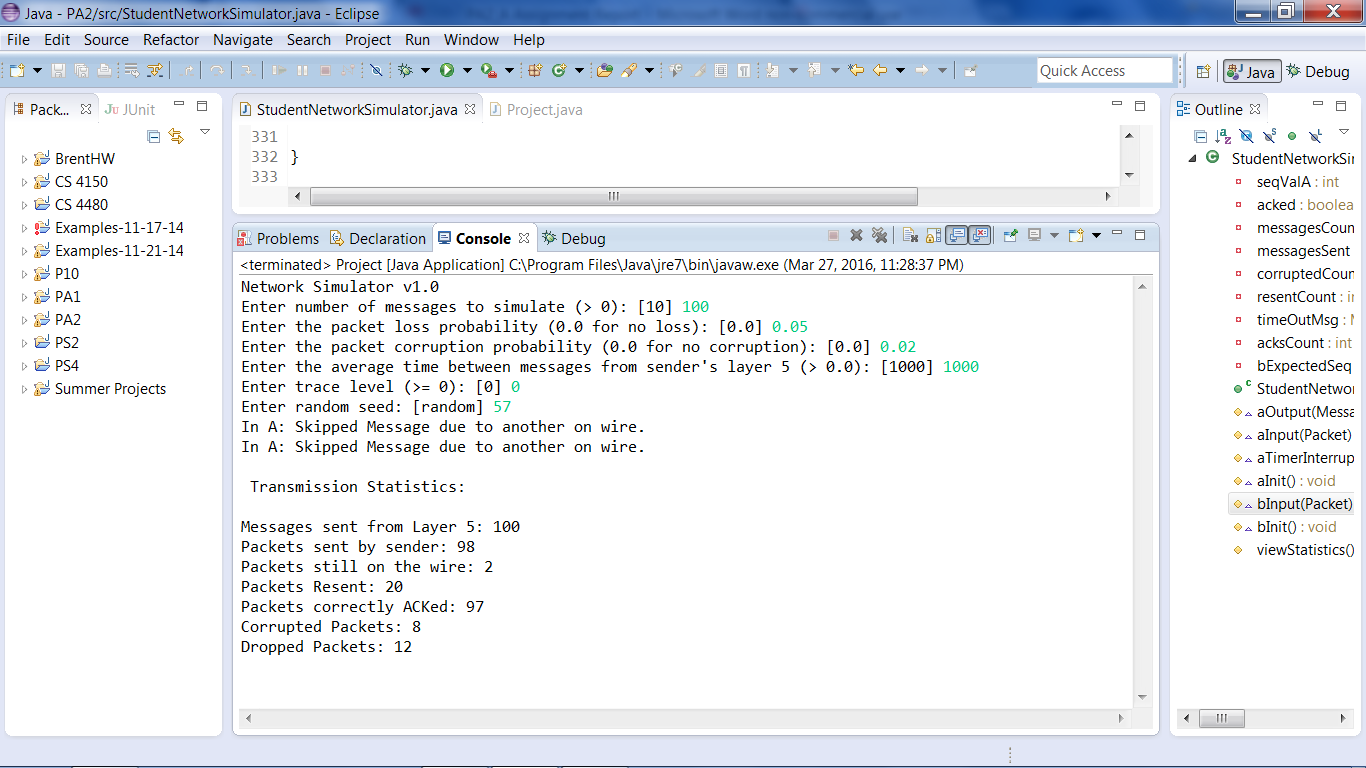
Corruption – 0.1, Loss – 0.1



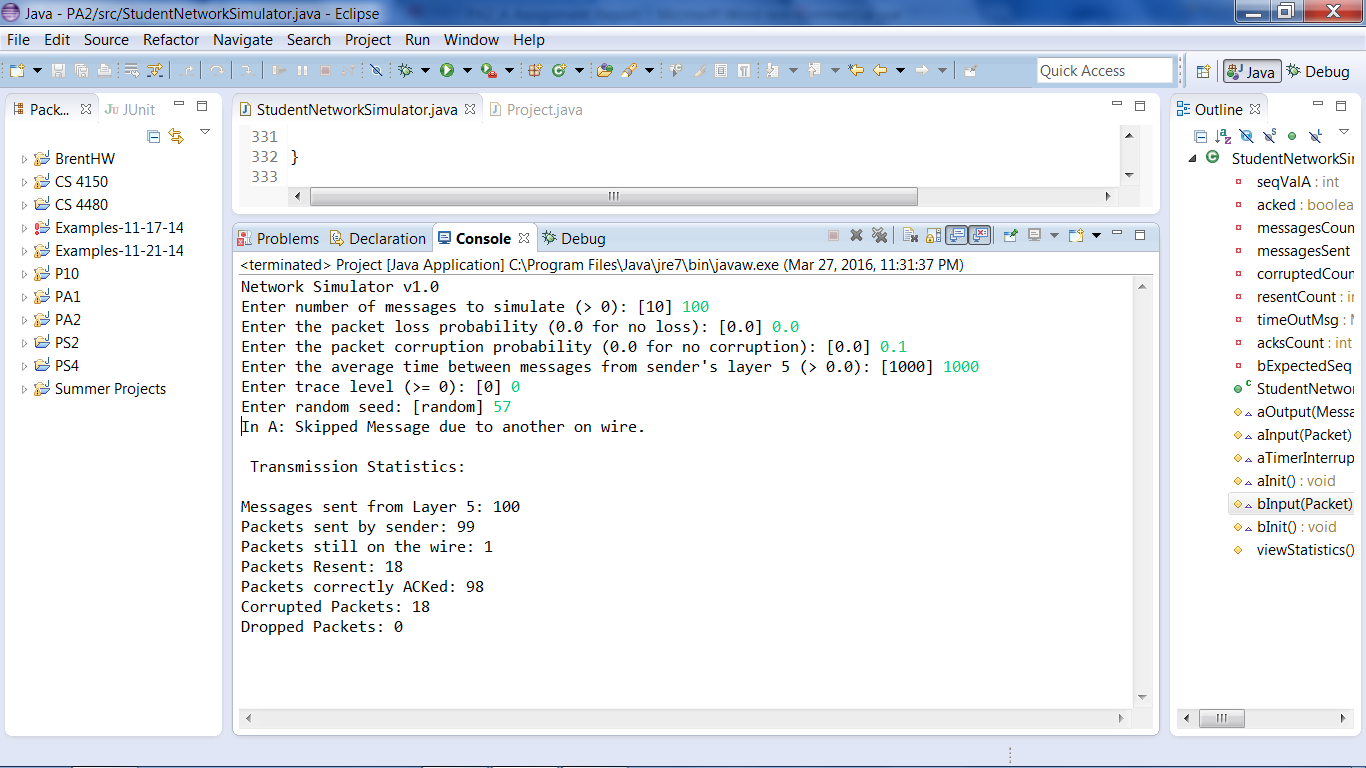
Corruption – 0.05, Loss – 0.02



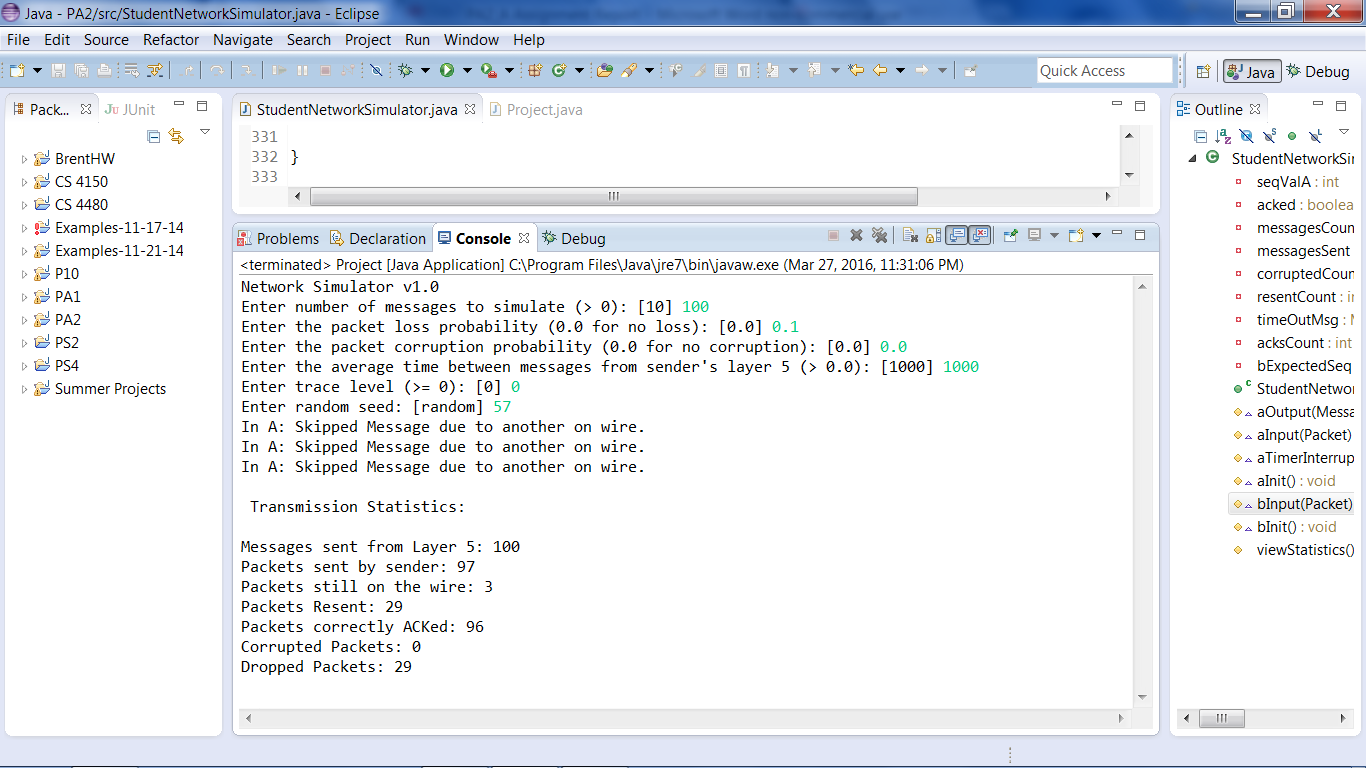
Corruption – 0.02, Loss – 0.05



Corruption – 0.10, Loss – 0.0

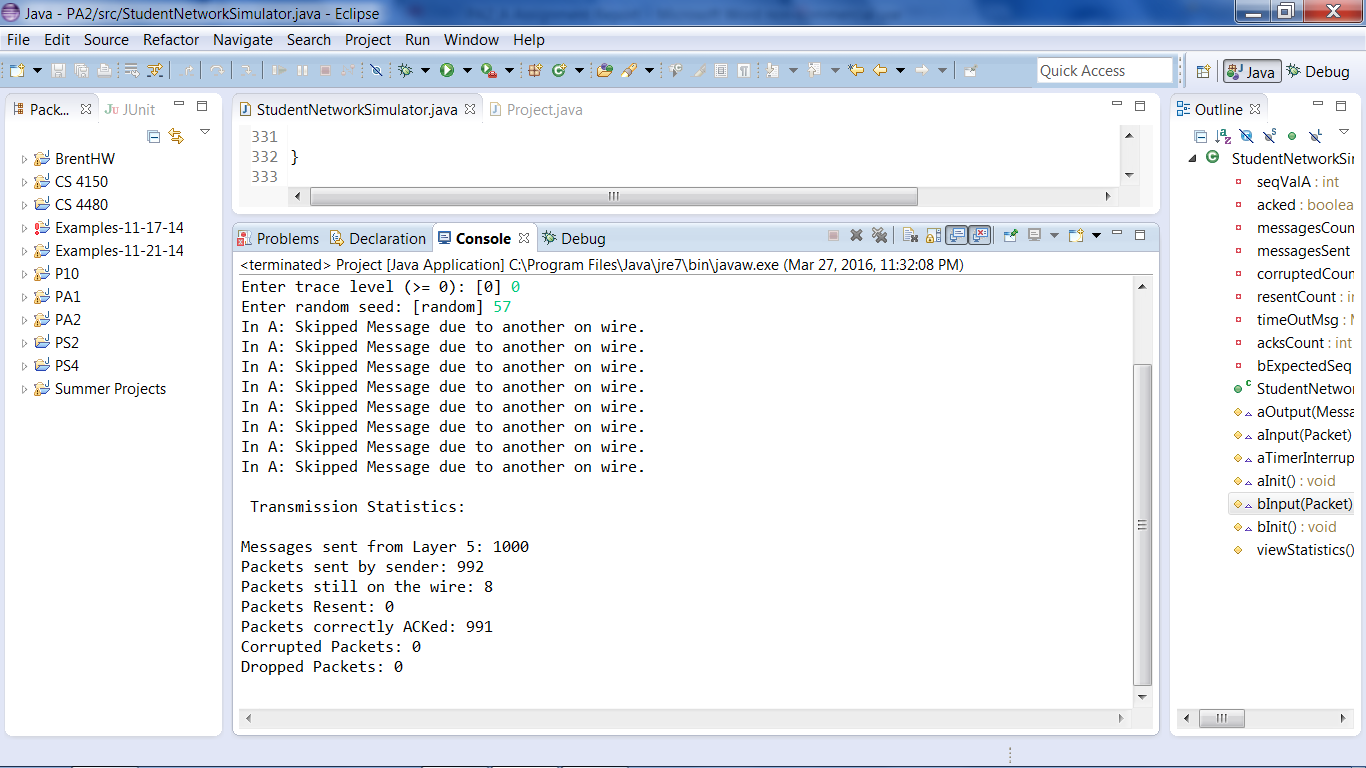


Corruption – 0.0, Loss – 0.10

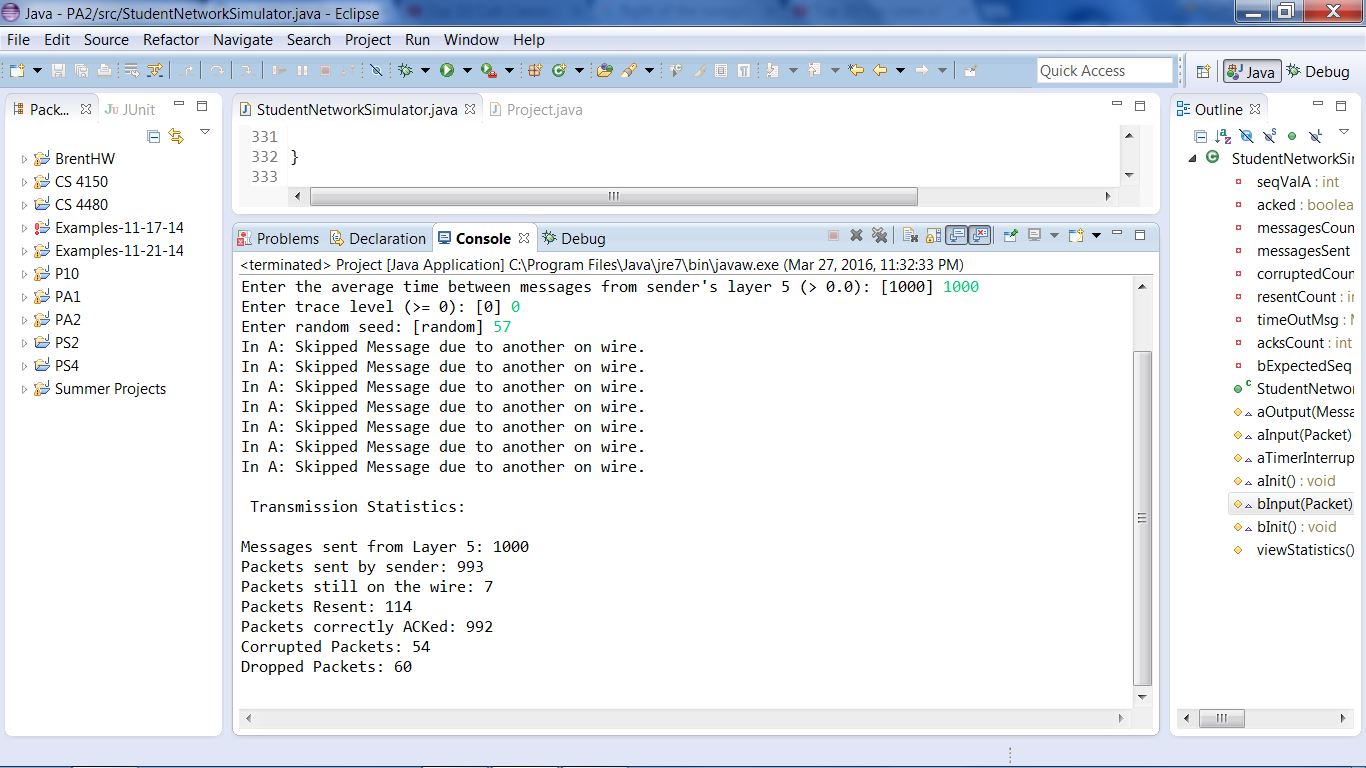


1000 Messages Sent

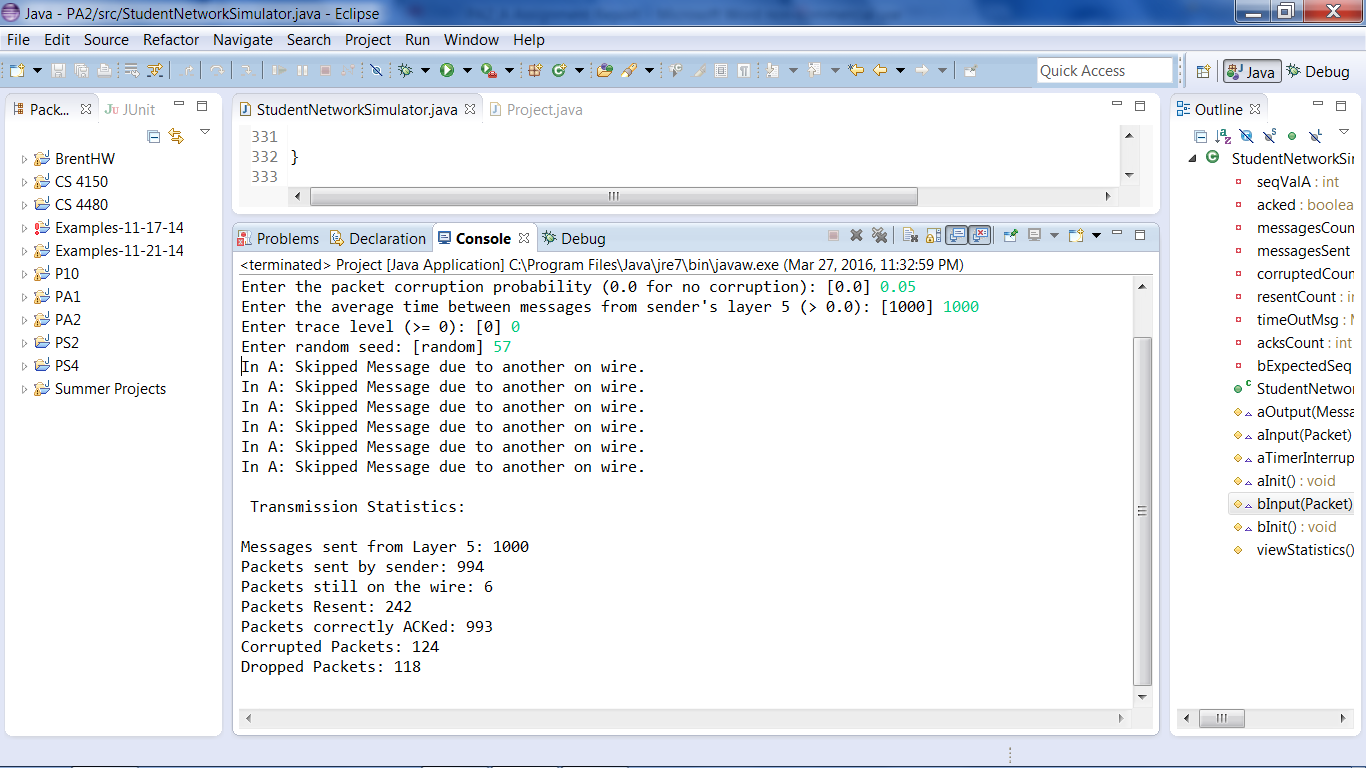
Corruption – 0.0, Loss – 0.0



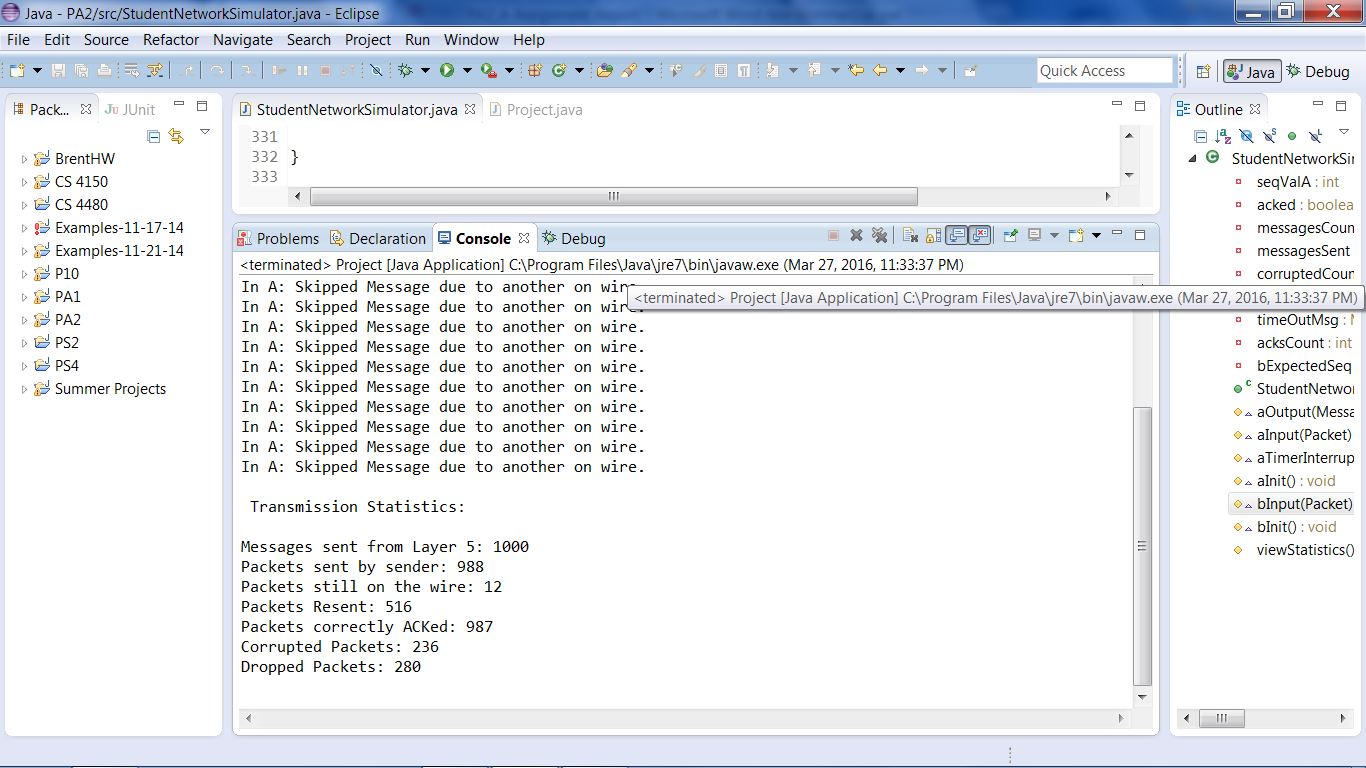
Corruption – 0.02, Loss – 0.02



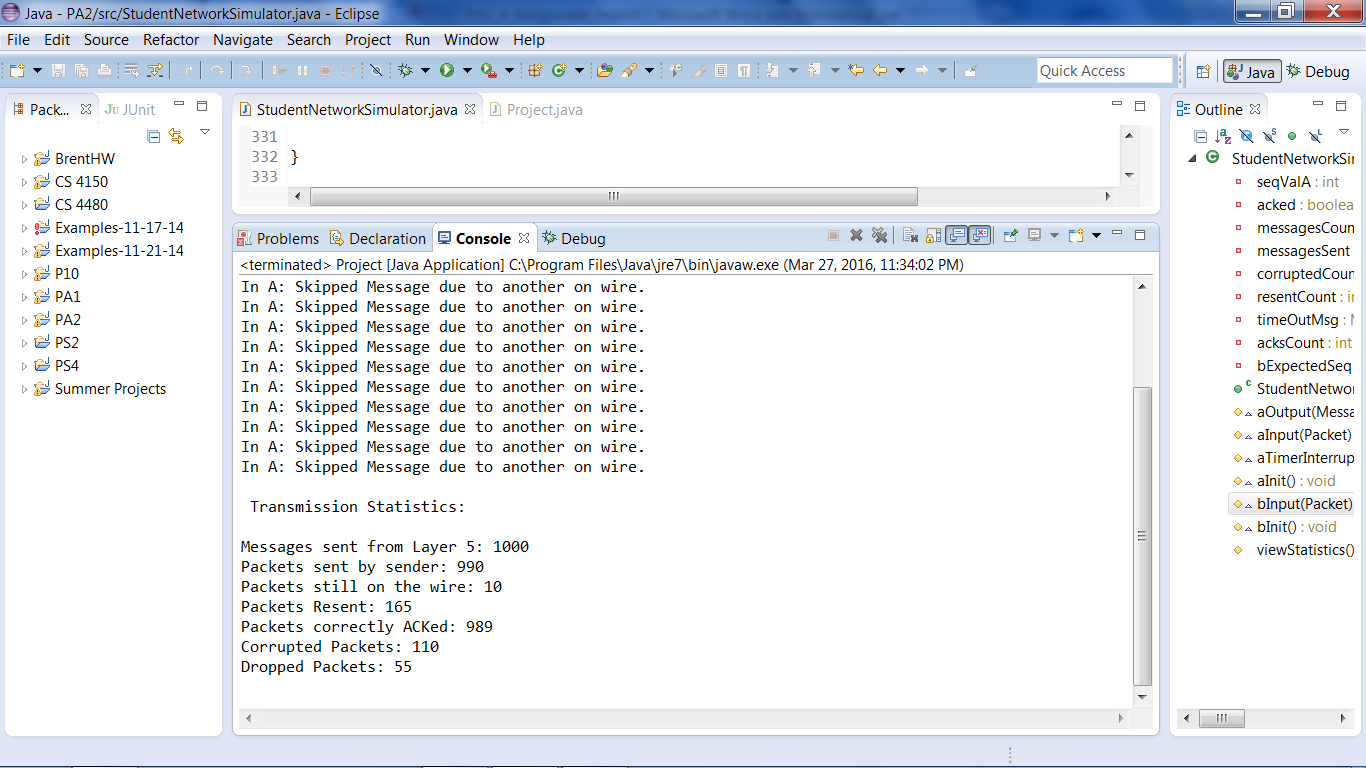
Corruption – 0.05, Loss – 0.05



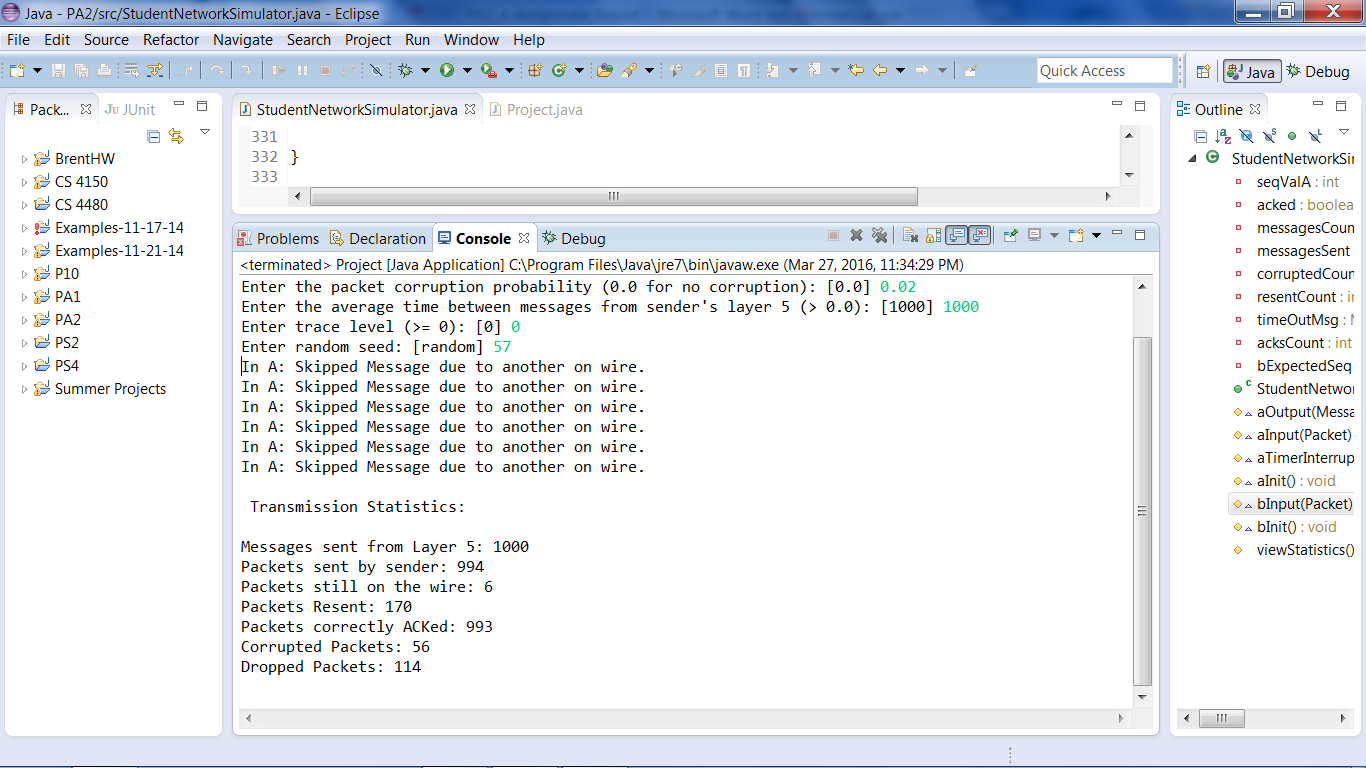
Corruption – 0.1, Loss – 0.1



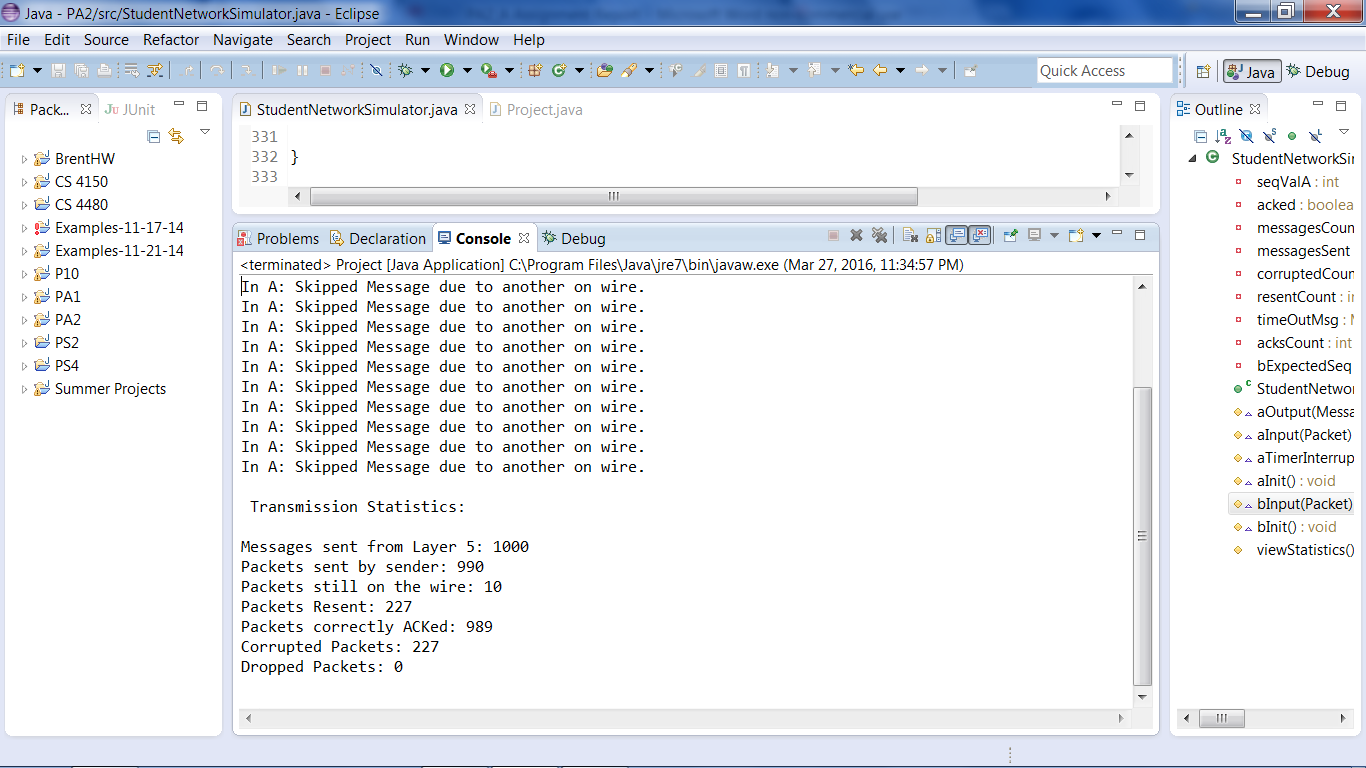
Corruption – 0.05, Loss – 0.02



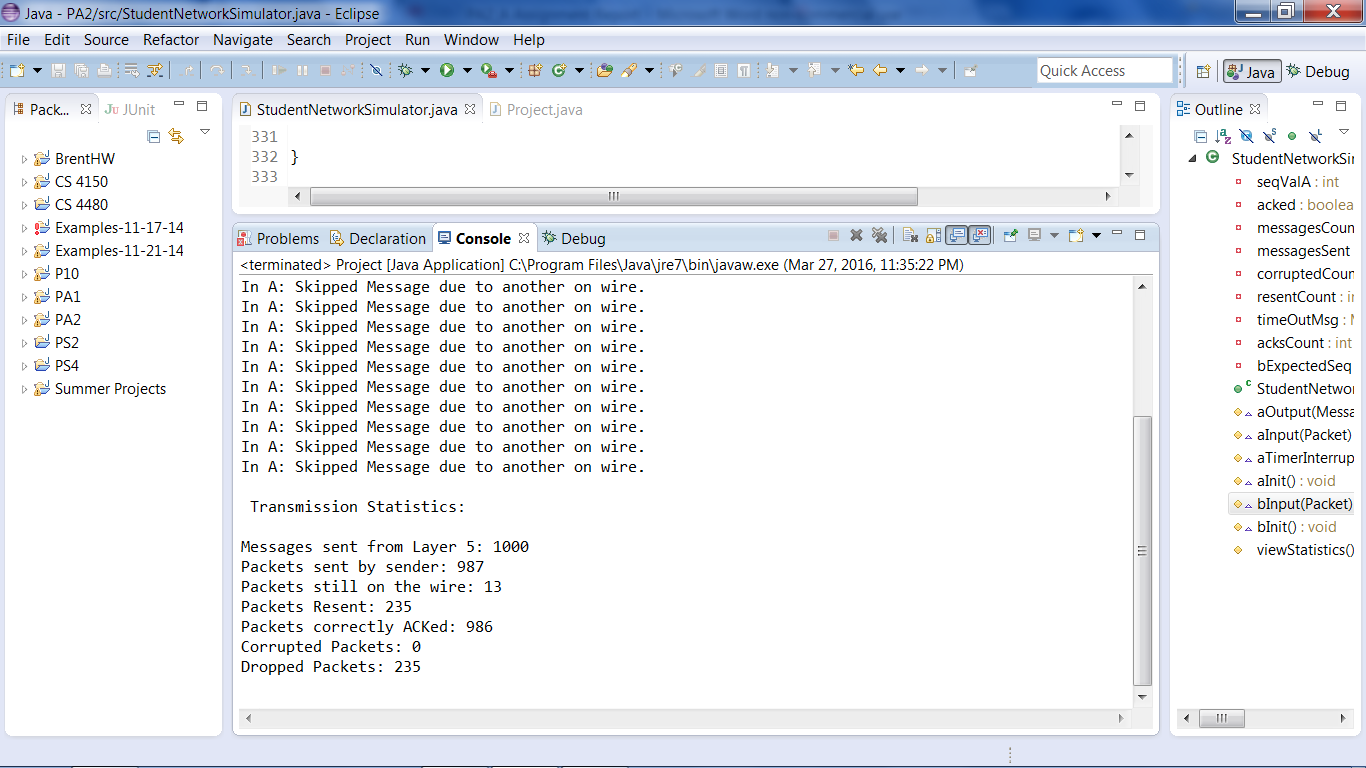
Corruption – 0.02, Loss – 0.05



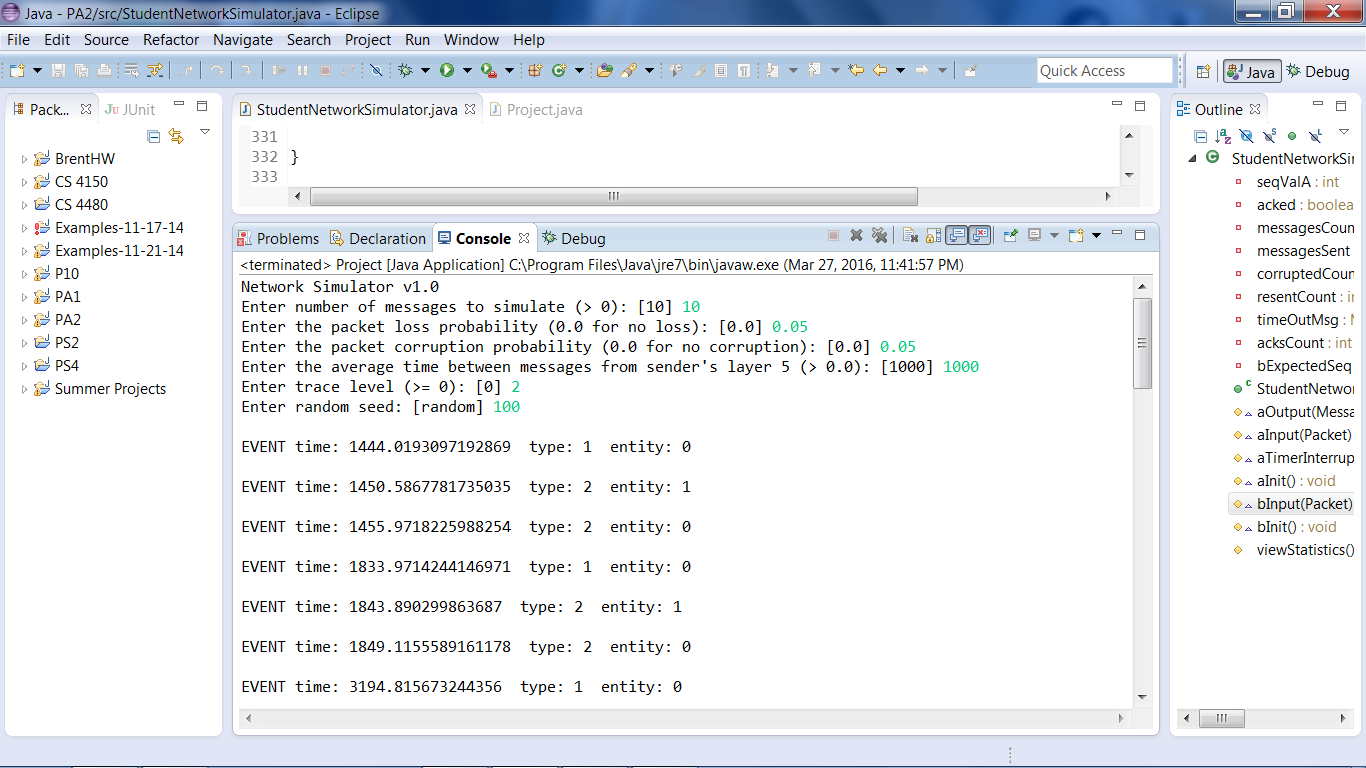
Corruption – 0.1, Loss – 0.0

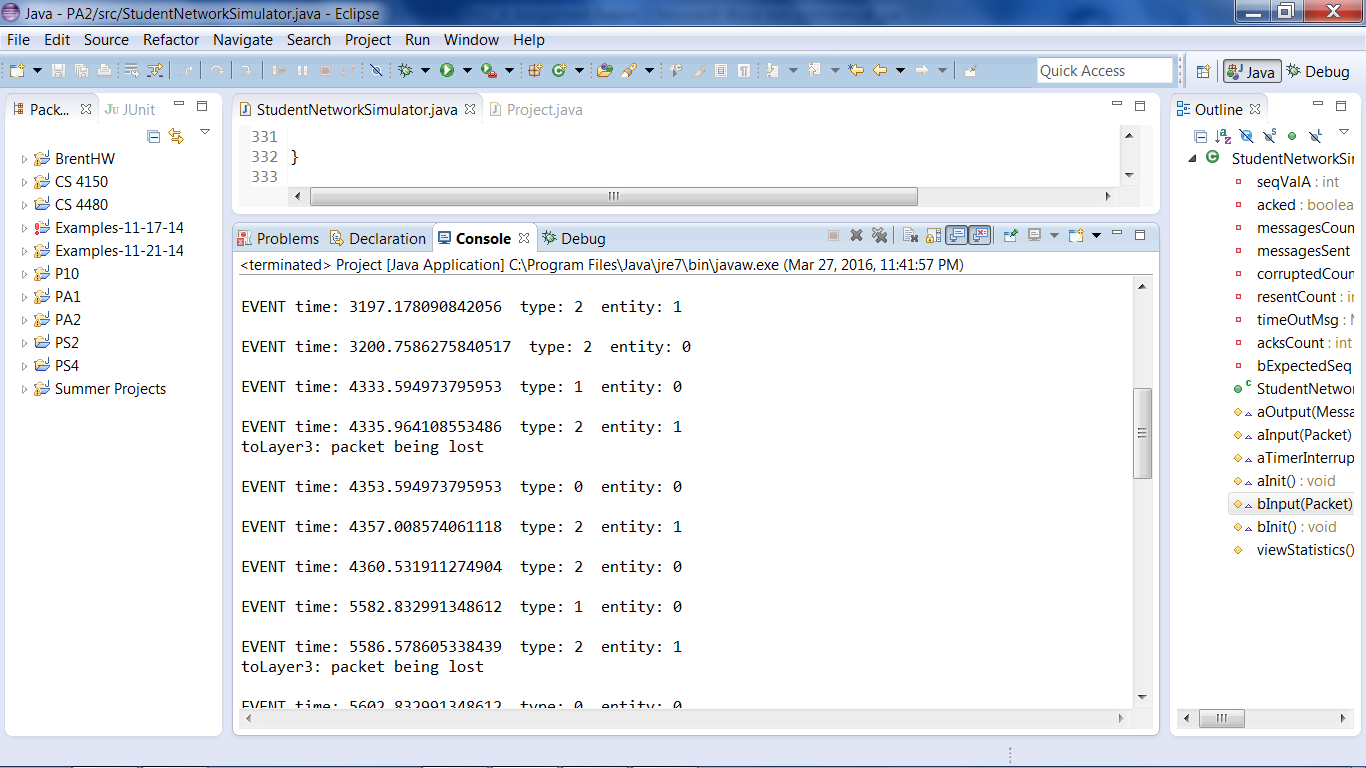


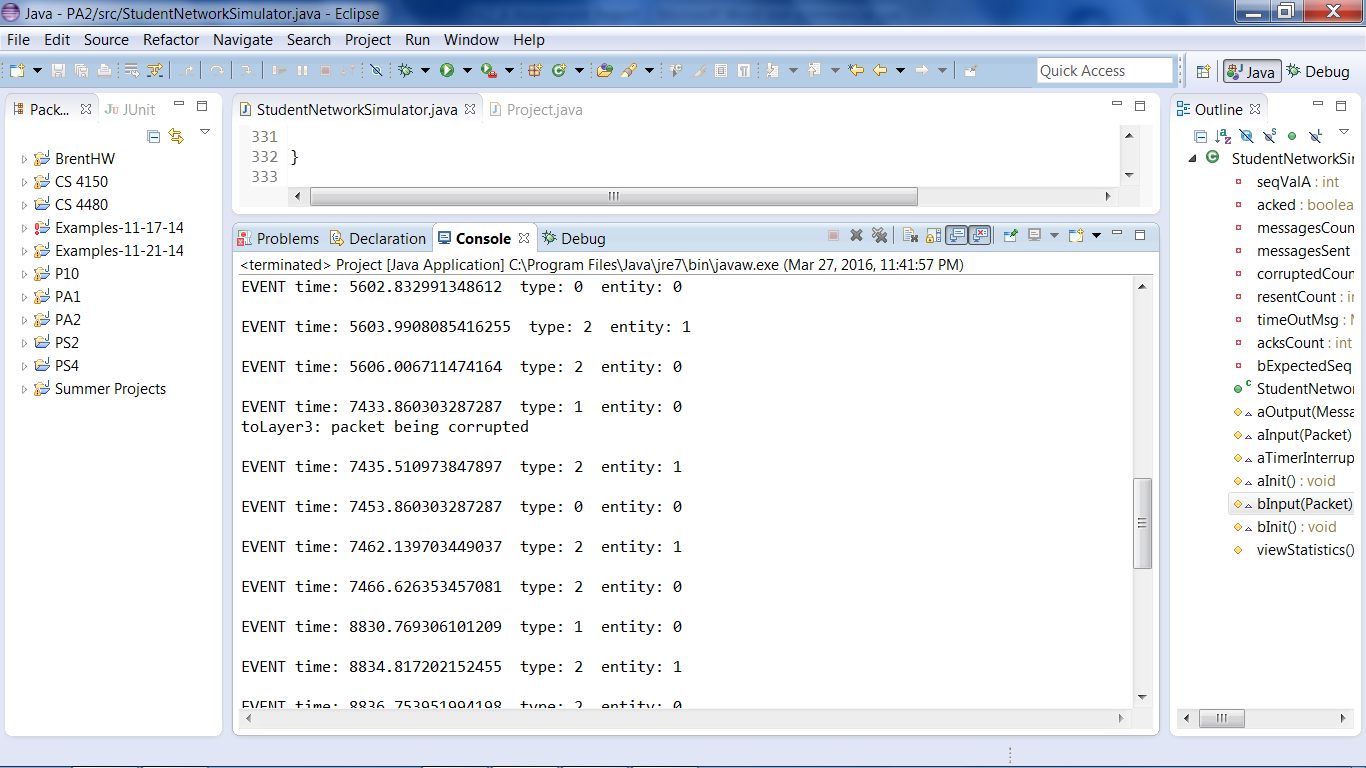
Corruption – 0.0, Loss – 0.1

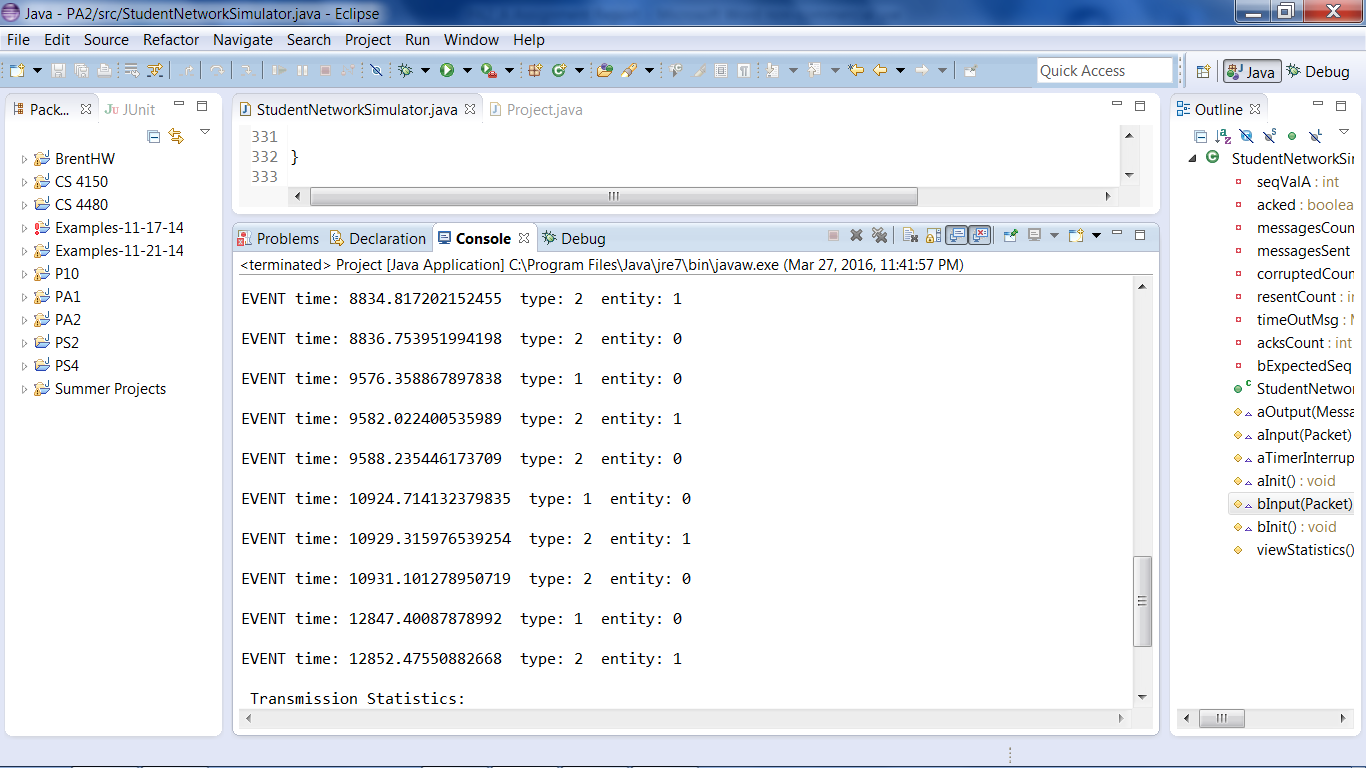


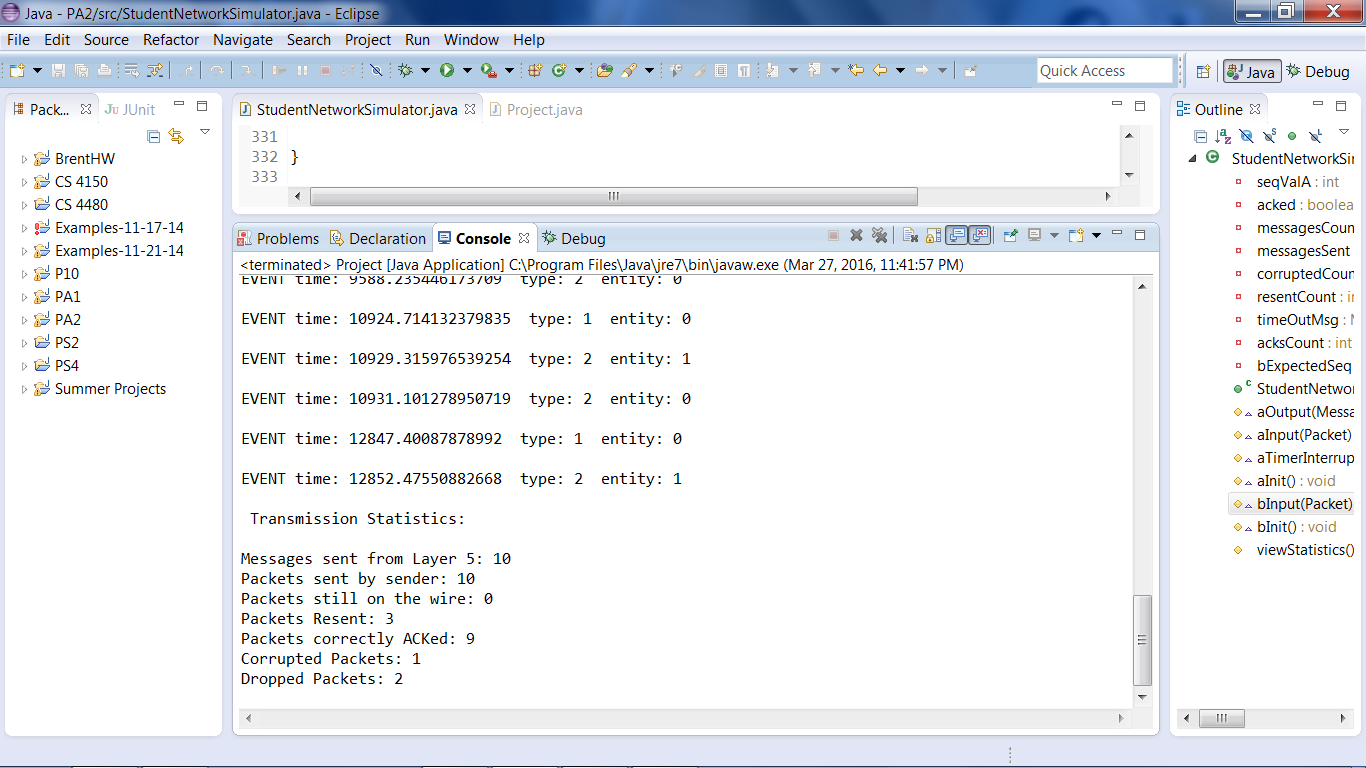
Requested Test From Specs Sheet











According to the experiment you wanted us to perform in section 3, in the output sub-section, the program returned corruption values that were a bit higher than expected. I was personally expecting both the corrupted packets and the dropped packets to both be 1 (because the values of both them would be less than 1, but you can’t have a packet that is half lost or half corrupted so they would round to 1), but the corrupted packet value is 2, which although it is higher I believe this is within expectable range.