There are two main components to the application that allow it to function. There is the sender and the receiver. The primary function of the receiver is to ensure that the messages are sent to the receiver ideally in order. The sender opens the “alice.txt”, which it reads one line at a time. Once it has a line the sender checks to make sure that the window has room for the information. Then the segment information such as seqnum, length, checksum and payload are set correctly. The application then establishes the packet and datagram to send it; however, the segment needs to be serialized before sending. After the object is serialized, it is sent and a timer starts to make sure that in case of a time out the message is resent. In the event that the window is full the sender will send a message informing the user the message couldn’t be sent, but the sender will retry. The sender also receives the acknowledgement number from the receiver and sets the sendbase value. In addition, the timer must be reset because there are no more acknowledgements to wait on. The other major component is the receiver, which compliments the sender.

The receiver segment continuously loops waiting for input from the sender. After receiving the packet the receiver must deserialize the object into an explicit object. If the object was passed successfully it is retrieved and the checksum is checked to ensure integrity. The main function the user will notice is the receiver prints the line of text to the console. The receiver must now return an acknowledgement number to the sender, so it is aware the message was received. After the acknowledgment number is incremented by one for the next time the sender sends a message.

Once the end of the file is reached the connection between the two segments is no longer necessary. The user should be able to see all of the lines from Alice in Wonderland in their receiver console. This should have effectively replicated a TCP like connection using UDP.

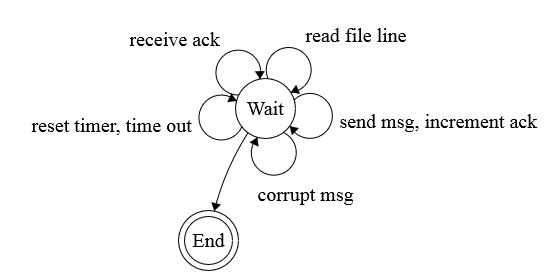
**Additional requirements:**

The sender and receiver will have a 5% failure rate as indicated by the assignment. 5% of the time the sender should not receiver an acknowledgement. In the instance that the timer times out then the message should be resent. Otherwise the sender will receive the next acknowledgment number and know the previous message was also received. On the other hand, the receiver will simulate the checksum being wrong. This results in an acknowledgement never being sent and the sender timing out. The sender will resend messages until the receiver returns an acknowledgement number.

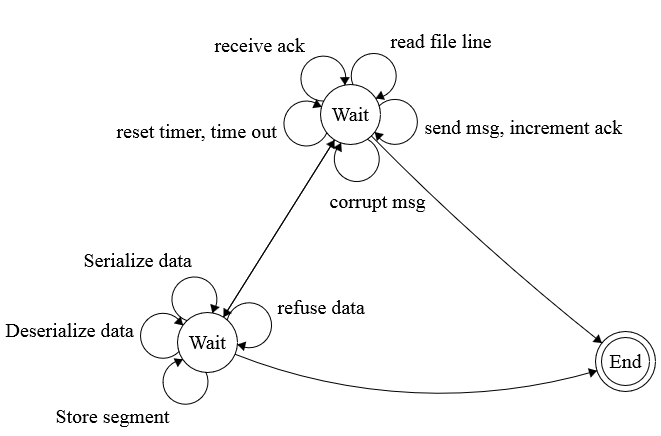
**Finite State Machine:**

These finite state machines have been modified since the beginning of the project. A copy of the machine can be found in the original documents submitted and below. The first machine is the modified Sender. The reason for two states called wait was because of a limitation in the software used to develop the fsms. The software does not allow the user to resize the state and the line, so once the team ran out of space the best option for readability was to create another state called wait. The reason for the change was because the team did not realize some of the subtle details needed to send an object over a UDP connection. This is the reason for the serialize data and deserialize data. Also there was no event for refuse data because it was implied in other events, but for clarity, the refuse data was added.

**Original Sender:**



**Modified Sender:**



**Psudocode for the sender:**

/\* initialization \*/

nextseqnum = 1;

sendbase = 1;

**event: rUDP\_send(message)** /\* message received from the application above \*/

-----------------------------------------------------------------------

if(nextseqnum < sendbase + N)

seg[nextseqnum] = make\_seg(nextseqnum, data, checksum);

UDP\_send(seg[nextseqnum]); /\* pass segment to the UDP socket \*/

if(sendbase == nextseqnum)

reset\_timer();

nextseqnum += seg[nextseqnum].length;

} else

refuse\_data(message);

**event: timeout**

-----------------------------------------------------------------------

UDP\_send(seg[sendbase]);

reset\_timer();

sendbase += 50; /\* to prevent overflow at low values of N \*/

**event: rUDP\_rcv(rcvseg) && notcorrupt(rcvseg)**

-----------------------------------------------------------------------

sendbase = rcvseg.acknum; /\* ACK received, with acknum field value \*/

if (sendbase < nextseqnum)

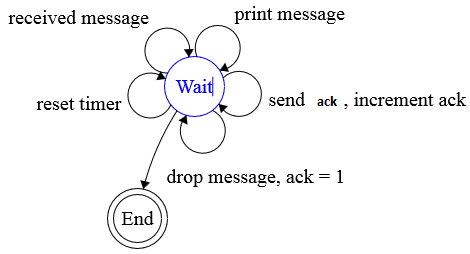
reset\_timer();

**event: rUDP\_rcv(rcvseg) && corrupt(rcvseg)**

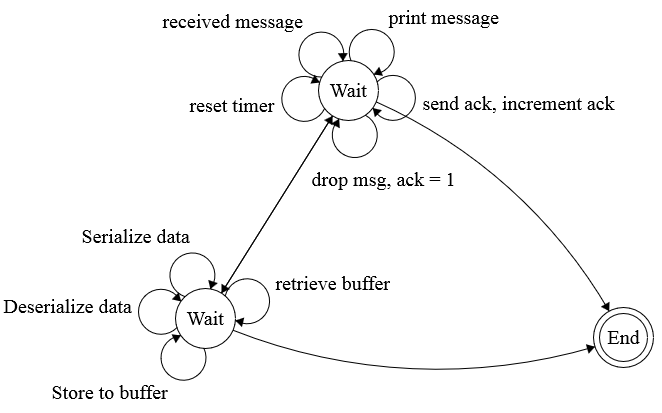
-----------------------------------------------------------------------

/\* do nothing \*/

**Original Receiver:**



**Modified Receiver:**



**Psudocode for the receiver:**

**/\* initialization \*/**

nextacknum = 1;

**event: rUDP\_rcv(rcvseg) && corrupt(rcvseg)**

-----------------------------------------------------------------------

seg[nextacknum] = make\_seg(nextacknum, checksum);

UDP\_send(seg[nextacknum]);

**event: rUDP\_rcv(rcvseg) && notcorrupt(rcvseg)**

-----------------------------------------------------------------------

buffer\_seg(rcvseg); /\* buffer this segment \*/

deliver\_data(); /\* deliver to the application all segments

which follow the sequence without gap \*/

nextacknum = update\_acknum(); /\* update the nextacknum \*/

seg[nextacknum] = make\_seg(nextacknum, ,checksum);

UDP\_send(seg[nextacknum]); /\* send the feedback to the sender \*/

**Know Bugs:**

This is the list of currently known bugs inside of the application. The bug issue will be described, the estimated time worked on the bug and if the bug was resolved will be included.

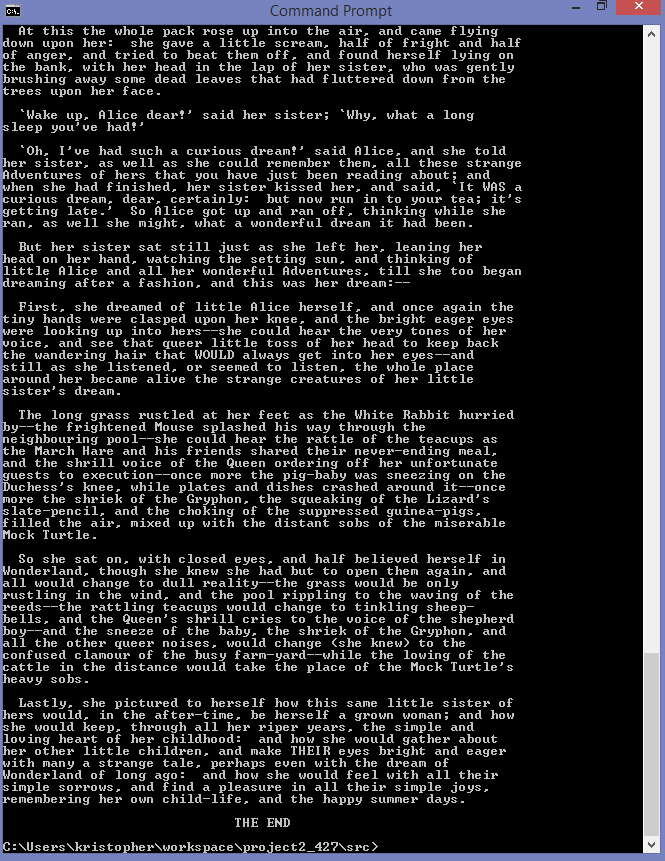
|  |  |  |
| --- | --- | --- |
| **Bug description** | **Time worked on:** | **Resolved** |
| The ObjectInputStream header becomes corrupt during the second attempt of sending a packet to the receiver | 12 hours | Yes |
| The UDP\_rcv method did not receive the packets correctly | 30 minutes | Yes |
| The sender will repeatedly re-send the same message until nextseqnum < sendBase + window size | 1 hour | Yes |
| Timer not working as intended | 5 hours | Yes |
| Buffer not working | 4 hours | Yes |

**Buffer size and use:**

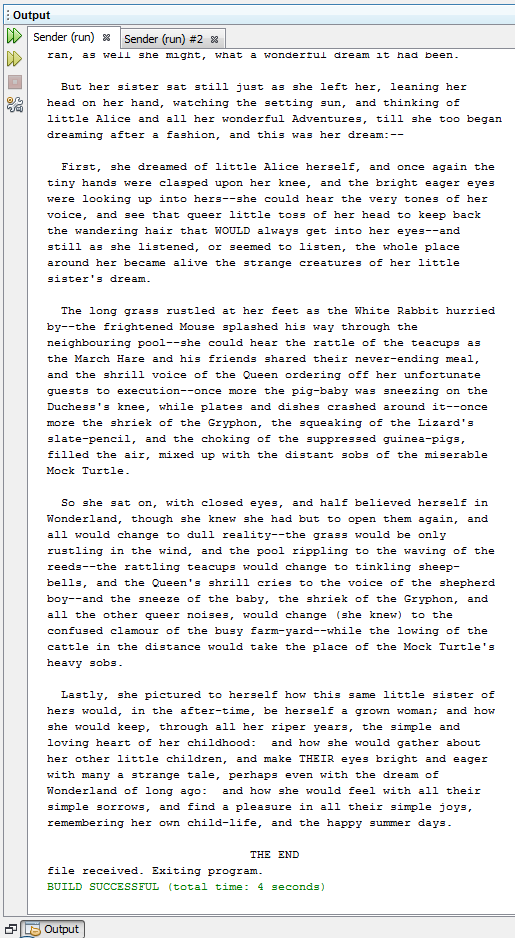
The buffer size can be changed by changing the value of N within the program. The default value is 100, but it can be increased or decreased as needed. Values of N smaller than 100 can cause the buffer to become overloaded. To combat that, when the timeout function is called sendBase is increased by 50. If it is not, the program loops endlessly trying and failing to send whatever chunk of data it got stuck on. The time to send the full file with n=100 is about 2 minutes and 30 seconds. With n=50 the time is about 10 minutes and 40 seconds.

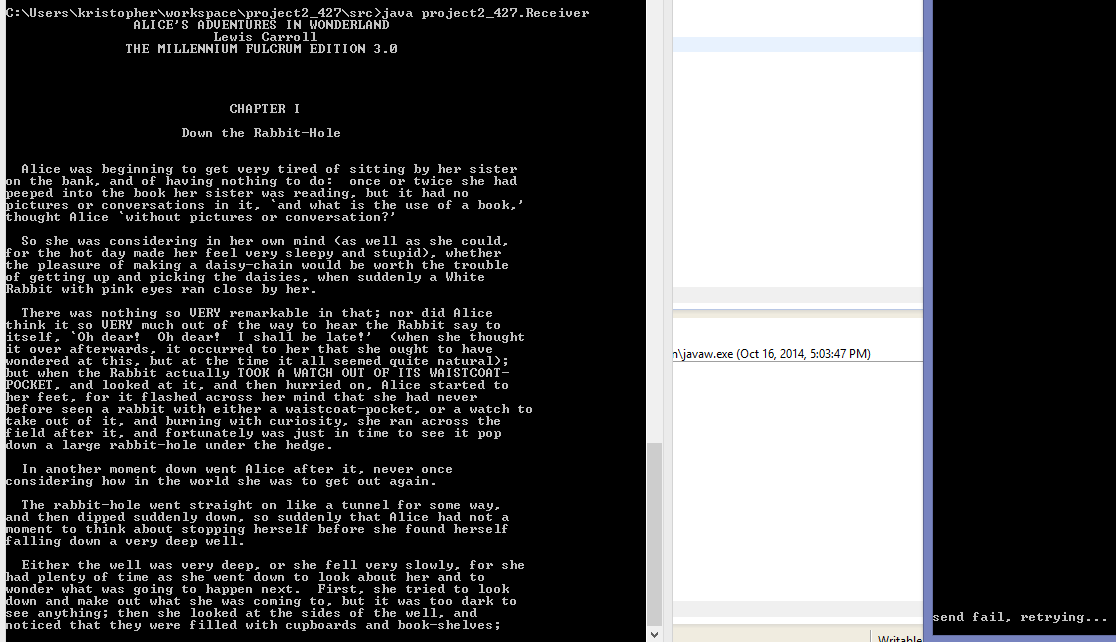
**Test scenario:**

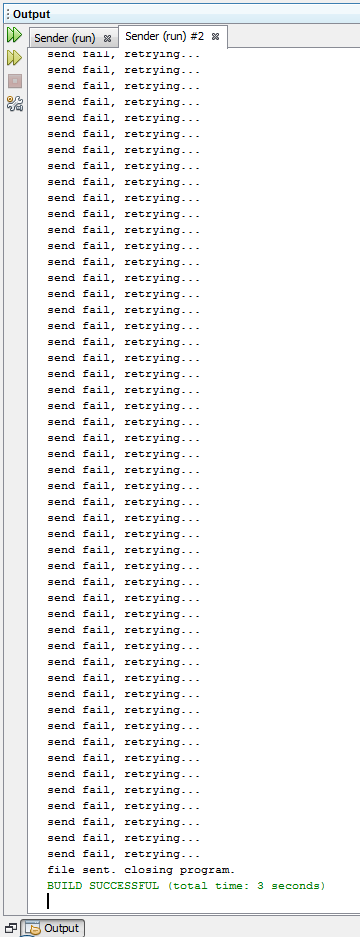
The test scenario demonstrates the ability of the sender and receiver to communicate between each other. This test scenario does not include the buffering or resending of lines. The main purpose is to prove that the first bug with the ObjectInputStream is no longer an issue.



This test scenario involved putting the acknowledgement number to be greater than sendBase + window size on purpose to indicate that the message retrying would show. The next step to build onto this is having the timer time out to resend the message. The proof is below for this test scenario.







The total time of execution was:

