For this assignment, you will implement a reliable message passing protocol on top of UDP/IP. You may use either Stop & Wait (ARQ) or Sliding window for your implementation. More points will be awarded for the latter choice.

Each packet in your protocol will have the following format:

```
4 1 1 2 [0, 1024]
seqnum  ACK  control  length  DATA
```

Notice that there are no address fields, as we can encapsulate this message inside a UDP message which contains the correct host and port.

The 4 byte `seqnum` field contains the sequence number of the message and the 1 byte `ACK` field indicates whether the message is an ACK (0x01) or not (0x00). The 1 byte `control` field indicates the type of message according to the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>Data</td>
</tr>
<tr>
<td>0x01</td>
<td>Connection setup</td>
</tr>
<tr>
<td>0x02</td>
<td>Connection teardown</td>
</tr>
</tbody>
</table>

Your implementation must allow for a variable length DATA field whose length, in bytes, is in the `length` field. (The `memcpy()` function will again prove useful. See the `man` page.) Notice that the maximum data size is 1024 bytes, but the actual size of the data sent depends on a particular message. In other words, your protocol should not send 1024 bytes of data in every packet.

It is important that you store `seqnum` and `length` values in a message in network byte order. (Recall the `htonl()`, `htons()`, `ntohl()`, and `ntohs()` functions.) If you do not, the bytes will be swapped when messages are exchanged between hosts with different architectures.

**Connection setup and teardown**

Before any data can be sent between a sender (client) and receiver (server), a connection must be established. A connection message from a client looks like this:

```
4 1 1 2
initial seqnum  0x00  0x01  0x0000
```

Notice that the setup message contains no data. The initial sequence number is randomly chosen by the client.

The receiver will respond with an ACK that looks like this:

```
4 1 1 2
initial seqnum  0x01  0x01  0x0000
```

The sequence number in the ACK should be the same as the one sent by the client. This acknowledges that the server agrees upon this initial sequence number.

If the client does not receive a response from the server before its timeout interval, it should resend the request with the same initial sequence number. The client will retry some constant number of times before giving up.

When the data transfer is finished, the channel must be torn down. This will signal the client to save all of the received data and properly close the file descriptor it is being written to.

The teardown process is nearly identical to the setup process except that the control value `0x02` is used and the sequence number is the last sequence number used by the client.
Data transmission

Given the discussion above and your understanding of reliable message passing, sending data packets should be straightforward. Remember that the server must check the sequence number of every packet it receives. If it receives a packet with a sequence number it has already seen, it should discard the packet and resend an ACK. Likewise, the client needs to check the sequence number of all ACKs it receives.

Service interface

The design of the service interface is up to you. An object oriented design is best. One idea is to create a class for a sending channel and a class for a receiving channel. The sending channel might have functions for setting up the channel, sending data on the channel, and tearing down the channel. Likewise for the receiving channel.

Testing

To simulate a lossy network, put a construct like this in your receive function so that sometimes ACKs are not sent:

```cpp
#include <stdlib.h>  // for rand()
...
bool Error(int chance)  // produce an error with probability 1/chance
{
    return (rand() < (RAND_MAX / 4));
}
...
Receive(...)
{
    ...
    // receive a packet
    if (!Error(4))  // lose a packet 1/4 of the time
    {
        // send ACK
    }
    else
        cout << "receive: Oops, packet lost!" << endl;
}
```

Do a similar thing in your `Send()` functions to simulate packet duplication.

Think carefully about all the possible scenarios we discussed in class, in addition to the other possibilities that may not have been raised yet.

Deliverables and Grading

As with previous projects, I expect a well written README document explaining your protocol.

Points will be assigned as follows:

<table>
<thead>
<tr>
<th>Design and style</th>
<th>35%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stop &amp; Wait</td>
<td>50%</td>
</tr>
<tr>
<td>Sliding window</td>
<td>15%</td>
</tr>
</tbody>
</table>

You can see from the table that a working, well-documented Stop & Wait protocol can earn you 85%. To receive 100% you need to implement Sliding Window.