In this project, you will write layered software to support one-way (half-duplex) communication between a single sender/receiver pair. Below, I provide you with the service interface for each of the layers, specified as function prototypes. Your job is to implement all but the bottom-most layer (which I provide). Each layer must be independent, which means that it cannot depend on any properties of the layer below, beyond what is promised at the service interface description.

Service Interface Description

1. Layer 1: Single Byte
   Layer 1 provides ordered single byte communication from the sender to the receiver. Note that you do not need to write this layer; just use it. If you look at the provided layer 1 implementation (discussed below), you will see that it simply uses a Unix pipe. You do not need to know anything about how that works to use the provided layer.

   Function prototypes:
   - bool l1_write(char b);
     Writes the byte specified by b. Returns true on success or false on error.
   - bool l1_read(char* b);
     Reads one byte and copies the byte to the address specified by b. Returns true on success or false on error.

2. Layer 2: Messages
   Layer 2 provides the ability to send and receive a message. A message is simply a sequence of bytes with a particular format. As the protocol designer, you will decide upon this format. Remember the fundamental network stack concept of encapsulation, wherein we use headers to convey protocol information. You should be able to draw a picture of the format used to exchange messages between peers, as well as describe how to interpret that format. The only way for the layer 2 functions to send or receive is by using the layer 1 functions l1_write and l1_read.

   When designing your peer-to-peer protocol, keep in mind that this is a general purpose protocol, and the definition of a message is correspondingly general (just a sequence of bytes, and not necessarily ASCII bytes), so you don’t want to preclude some particular byte from being able to appear in a message. The only assumption you can make about messages is that the length is no longer than 1024 bytes.

   Function prototypes:
   - int l2_write(char* buffer, int length);
     Sends a message consisting of the sequence of bytes starting at the address in buffer and having length length. Returns length on success or -1 on an error. You need to handle all errors that can be detected here, including the validity of arguments and the return value for any calls to l1_write. Note that buffer will point to the data the user wants to send. The implementation will need to attach a header before the message is given to layer 1.
• int l2_read(char* buffer, int maxLength);
Reads a message and stores it at the address specified by buffer. No more than maxLength bytes will be put into memory. If a message received by l2_read would require more than maxLength bytes, l2_read should return -1. Upon successful reception of a message, the size of the message (the number of bytes actually stored in buffer) is returned. Note that any header received from layer 1 will be stripped before the message is returned to the user.

Make sure that your l2_read does not allow the sender to overflow the buffer! It is not enough to recognize when this has happened and return an error; you must not store anything beyond the maxLength location of buffer.

3. Layer 3: Messages with Error Detection
Layer 3 adds simple error detection to the services provided by layer 2. The service interface for layer 3 looks the same as the layer 2 service interface, but the layer 3 read should also return a -1 if it detects an error in the received message. The errors we are looking for here are transmission errors, where one or more bytes might have been corrupted. We want to make sure that the message received is the same as the message that was sent. To accomplish this, we need some form of error detection. The simplest approach is to use a checksum.

Function prototypes:

• int l3_write(char* buffer, int length);
  Sends a message that consists of the sequence of bytes starting at the address specified by buffer and having length length. Returns length on success or -1 on error.

• int l3_read(char* buffer, int maxLength);
  Reads a message and stores it starting at the address specified by buffer. No more than maxLength bytes will be put into memory. If a message is received by l3_read that would require more than maxLength bytes, l3_read should return -1. Some error detection mechanism is used to detect transmission errors. If such an error is detected by l3_read, a -1 should also be returned. Upon successful reception of a message, the size of the message (the number of bytes stored in buffer) is returned.

4. Layer 4: Name/Value Pairs
Layer 4 will provide a mechanism for sending and receiving values that have an associated name. For example, a telephone directory service could be built on top of this layer by having clients send a request that could be either a person’s name or phone number; the server could do the appropriate lookup and send back the appropriate result as one of these name/value pairs. (Note that this is simply an illustrative example. I am not asking for any higher level software, not am I looking for anything that turns this one-way communication into something capable of both directions.)

Function prototypes:

• bool l4_write(char *name, int nameLength, char *value, int valueLength);
  Sends the (name, value) pair to the receiver, where nameLength specifies the number of bytes in name, and valueLength specifies the number of bytes in value. Returns true on success, or false on error.

• bool l4_read(char* name, int* nameLength, char* value, int* valueLength);
The `14_read` function reads a (name, value) pair into the buffers pointed to by `name` and `value`. The `nameLength` and `valueLength` parameters are in/out parameters. On input to the function call, the integers should contain the maximum sizes of their respective buffers. On successful return from the function call, the integers hold the actual byte counts received in their respective buffers. The return value of `14_read` should be true on success or false on error, including the case where the name buffer or the value buffer would overflow.

**Sample Code and Layer 1**

On my web page, you will find a sample program that should give you an idea of how to use and test your final code. (But you will, of course, thoroughly test each layer individually before implementing the next layer!) A very simple layer 1 implementation is also included.

**Deliverables and Grading**

You must provide one file for each of the 3 layers you will write — `12.cc`, `13.cc`, and `14.cc`. Each of these files must have their corresponding read and write functions exactly as specified above. I highly recommend using a makefile.

Remember that it is necessary for your layers to be independent. After submission, I may use a different (my own) layer 2 implementation with your layer 3 and layer 4 code, and everything should work correctly. Also keep in mind that when I test your code, I will generate error conditions, which must be handled properly (including some simulated transmission errors by providing a bit-level unreliable layer 1).

You must also include a README file that completely describes the protocol and implementation for each of your layers. If there are any special instructions for building your code, include that as well. You do not need to provide any code for layer 1 or a main program, as I will supply my own when testing your code.

Your work will be graded as follows:

<table>
<thead>
<tr>
<th>Style/Code structure</th>
<th>25%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layer 2</td>
<td>30%</td>
</tr>
<tr>
<td>Layer 3</td>
<td>30%</td>
</tr>
<tr>
<td>Layer 4</td>
<td>15%</td>
</tr>
</tbody>
</table>

The first category will be based on whatever code you turn in, so if you run short on time and cannot complete layer 4, you can still achieve up to 85%.