1 Objectives

The objectives of this assignment are:

- to help you think more effectively about protocols and how they are implemented;
- to familiarize you with some of the modularity, efficiency, and defensive programming considerations that arise in implementing protocols;
- to introduce Base-64 coding and a simple length-based framing mechanism; and
- to improve your programming skills.

This will be a nontrivial assignment for most of you. You should start soon, and expect to spend between 8 and 20 hours to complete.

2 Overview

For this assignment you will implement two protocol layers in a framework for composing layers to form a “protocol stack”. You may do this assignment using either C or Java. This assignment document is intended to be language-neutral; supporting or auxiliary code that I provide will be available in both C (not C++) and Java.

Your code will be evaluated on (i) the quality of its design, as determined by looking at it; and (ii) its correctness and robustness, as determined by running it against a driver framework written by me. (See the syllabus for details of program grading criteria.) Your code must conform to both the layering abstraction and to the given protocol specifications.

Note Well: The code you turn in for this assignment must be structured as described in this assignment, and must do exactly what is required by the protocol specifications, and nothing else. In particular, the code you turn in MUST NOT open files, produce any kind of console output, read from standard input, fork processes, or create threads. The functions you have to implement are computationally rather simple, and should not require many hundreds of lines of code to implement. You will not receive credit for code that violates this requirement.

Please read the notes on “Implementing Protocol Layers” and on “Buffering,” which will be available via the course web page. The rest of this assignment document is organized as follows. The next section introduces a buffer programming abstraction, which provides a convenient “container” for information being passed between layers. Section 4 introduces a specific programming abstraction for building layered systems, which uses the buffer abstraction. Then Sections 5 and 6 describes the two protocols you are to implement for this assignment. Finally, Section 7 contains further information on structuring and testing your code, as well as instructions on turning it in.
3 The Buffer Abstraction

It is convenient to define a common data abstraction to store protocol data (in particular, messages or packets) as it passes from the application program down through the stack, or from the network hardware up through the stack to the application. Such an abstraction allows auxiliary information—such as the number of bytes in the packet, or the interface on which it was received—to be kept with the packet itself. Most operating systems define some such common structure. This section describes a simple type called BUF.

Each instance of the BUF abstraction contains:

- A reference (in C, a pointer) to the data area of the buffer. Note that in general, not all of a buffer’s data area is in use at any time, because all messages/frames/packets are not the same size, and buffers generally come in “standard” sizes for efficiency reason.
- The total size of the data area, in bytes. (This will generally be the same for all buffers, but it is best to assume as little as possible about that.)
- A pointer to the beginning of the actual (“in use”) data in the buffer. This may be the same as the beginning of the data area, or it may differ, for example if a header has been “stripped” from an incoming packet.
- A count of the data bytes actually stored in the buffer. Data stored in a single buffer must be contiguous.
- Two references to other buffers, which can be used to implement two kinds of chains. The first is a chain of buffers that logically make up a single unit—typically a packet. If a packet is too big to fit in a single BUF, multiple Bufs can be chained together to hold the packet’s information. This also provides a convenient way to add a header to a packet (if space is not scarce): simply prepend a BUF containing (only) the header to the chain of Bufs containing the rest of the packet.

The second kind of chain is a list of complete packets, (each of which is a contained in a chain of Bufs).

- Space for auxiliary layer-specific information. Such information is sometimes useful for communication between layers. For example, incoming packets to the network layer may have associated auxiliary information identifying the interface on which they arrived; this information is handy to detect when packets are being forwarded out the same interface they came in on. As another example, it is nice to have the total number of bytes in a multi-BUF packet stored in the first BUF of the chain. The auxiliary information is opaque to the BUF subsystem—that is, the BUF routines never read or write it.

- A reference (known as a “callback”) to a deallocation method. When a layer or module is finished processing a buffer, it calls this function (giving a reference to the BUF as argument) to deallocate the buffer. By overriding (replacing) this reference with a pointer to its own method, a layer can arrange to be informed when a BUF that it originally allocated is freed by another layer—for example, so it can also free any auxiliary data structure associated with the BUF. When created, Bufs refer to a default deallocation method, which should ultimately be called by any layer-provided deallocation method.

Figure 1 illustrates the layout of data in the data area; this scheme makes it easy to trim data from the front or back of the message: just increase the pointer to the beginning of the valid data, or decrease the size of the valid data, respectively. **Note well** that data before the b_start pointer and data after location b_start + b_len do not exist in the buffer. Figure 2 shows how these concepts are realized using the fields of the buffer header; the contents of the buffer in this example as the five-character sequence “aaaaa”.

Figure 3 illustrates the two kinds of chains. The first packet in the list (labeled “Packet 0”) is contained in three Bufs; Packet 1 is made up of a single BUF, and Packet 2 is made up of two Bufs.

You will be given auxiliary code that will handle allocation and deallocation of Bufs. For C, this will consist of files buf.h and buf.c; for Java, it will be a class Buf.java. In either case you may want to add auxiliary methods to those provided, but you **MUST NOT** modify the code I provide. The code will contain a constant STDBUFSIZE,
Figure 1: Valid data in a buffer data area

<table>
<thead>
<tr>
<th>b_data</th>
</tr>
</thead>
<tbody>
<tr>
<td>b_bufsize</td>
</tr>
<tr>
<td>b_start</td>
</tr>
<tr>
<td>b_len</td>
</tr>
<tr>
<td>b_aux</td>
</tr>
</tbody>
</table>

Figure 2: Buffer header usage

Figure 3: Buffer chains
which is the default size of the data area of a newly created/allocated BUF. You may assume that STDBUFSIZE is approximately 2K bytes. However, anything in your code that depends on the value of STDBUFSIZE must use the symbolic constant.

4 The Layer Abstraction

For this assignment we will define a generic layer abstraction, which defines the “shape” of each protocol layer, i.e. the methods it exports. In Java, we do this simply by defining a layer interface. (In C++ we could probably define a class of some sort, but we are not going to do so. If you are object-oriented, use Java.)

In C, we define a data type (structure) containing pointers to the exported methods. (Actually, in C it is sufficient for your code to just define the methods of the proper types; the driver code can declare the protocol stack using just the method names.) The protocols you implement for this assignment must match this signature. In Java, you will create a class that implements the layer interface for each protocol; in C you will declare and initialize an instance of the layer structure and initialize it to contain pointers to your functions.

Each layer exports three methods:

- An “init” method, which initializes any common data structures used by the protocol. This method takes two arguments, a character string and an integer. In general these might be used to pass information about the protocol instance (e.g. an IP address or an interface name), but they will not be used by our protocols.

- An output method `downFn()`, which processes output data on its way to the network, i.e. data received from the layer above on its way to the layer below. This method takes as input a BUF, an index that tells its position in the stack, a list of all the protocols in the stack, an optional reference to session state information, and an optional reference to a container for return information (for example, a BUF chain). The down method returns an integer code indicating whether an occurred, and possibly the nature of the error.

- An input method `upFn()`, which processes input data on its way to the application, i.e. data received from the layer below on its way to the layer above. This method takes the same parameters and has the same return type as the down method.

So how does control pass from a layer to the next layer above/below it? This is ultimately the responsibility of the original caller at the bottom/top of the stack. It passes a reference to the whole stack, along with the index of the first protocol called, to the method. When the down (up) method completes its own processing, it increments (decrements) the index, and checks whether that element of the stack is non-null. If so, the method invokes the corresponding down (up) method of that layer, passing it the incremented (decremented) index as an argument. When the down (up) call returns, the return value is returned by the original method. This “linkage” mechanism can be implemented pretty much the same way for all protocols in the normal case; error-handling methods may differ for some protocols, but they probably won’t for this assignment.

Observe that for this assignment the layers are oblivious to each others’ function; that is, layers conforming to this abstraction should be composable in any sequence that makes sense semantically. Note also that the layer array must have a NULL entry to mark both the “top” and “bottom” of the protocol stack.

4.1 C Layer Definition

The generic layer is defined in C as follows:

```c
struct layer_struct {
  int (*initFn)(int, char *);
};
```
typedef struct layer_struct layer;

A layer can be “declared” by initializing a layer structure to point to the appropriate methods; a stack can be declared by creating an array, as follows:

```c
int testlayerInit(int indx, char *name) {...}
int testlayerDown(BUF b, int where, layer stack[],
    void *ignored, void *retval) {...}
int testlayerUp(BUF b, int where layer stack[],
    void *ignored, void *retval) {...}
int fooInit(int i, char *dummy) {...}
int fooOutput(BUF b, int whoami, layer stack[],
    void *ignored, void *ret) {...}
int fooInput(BUF b, int whoami, layer stack[],
    void *ignored, void *ret) {...}
```

```c
layer stack[4] = {
    NULL,
    { testlayerInit, testlayerDown, testlayerUp },
    { fooInit, fooOutput, fooInput },
    NULL
}
```

To initiate processing of, say, an output message contained in a BUF chain b, one would call:

```c
error = testlayerDown(b,1,stack,NULL,NULL);
```

The linkage portion of testlayerDown (or any other “down” method) should look something like:

```c
if (stack[where+1] != NULL)
    error = stack[where+1]->down(bb,where+1,stack,...,...);
return error;
```

The “up” linkage code is similar except the index is decremented instead of incremented.

### 4.2 Java Interface

The Java interface is somewhat simpler.

```java
public interface Layer {
    /* Note: any class implementing this function should take care of
     * any needed housecleaning in the normal Java initialization
     * and finalization routines.
     */
```
Specific layer implementations take the form of a class that implements the Layer interface:

```java
public class Base64Proto implements Layer {
...
    public int Base64Init(int arg, String foo);
    public int Base64Down(Buf input, ... ) { ... }
    public int Base64Up(Buf input, ... ) { ... }
...
}

public class LFramer implements Layer {
    public int LFInit(...) { ... }
    public int LFDown(...) { ... }
    public int LFUp(...) { ... }
}
```

A stack can be created by initializing an array of Layers with specific class instances, as in C:

```java
Layer stack[4] = {
    null,
    Base64Proto,
    LFramer,
    null
};
```

Note that your layer must export all three methods required by the generic layer type signature. So you need to define an Init() method, even if it does nothing.

You may assume that any BUF arguments passed to your methods from outside or from another layer are non-null. Note that you are responsible for ensuring that this assumption is valid in each protocol you implement. Your code may not assume that the optional session and return information references are non-null; for this assignment those arguments should be ignored anyway.

5 Protocol 1: Base64 Encoding

Your first layer will implement the Base64 encoding, as specified in Section 5.2 of RFC 1521 (which see). Base64 is a way of encoding arbitrary byte sequences as 7-bit ASCII characters. It was originally designed to enable binary content to pass through electronic mail systems that were only designed to transfer text messages. (Some such mail systems were known to do evil things like clearing the high-order bit of every byte in the body of a message.)

The encoding is very simple: the input is read in 24 bits (three bytes) at a time; the 24 input bits are then treated as 4 6-bit chunks, as shown in Figure 4. (If the input is not a multiple of three bytes long, it is padded with bytes of zero.) Each 6-bit value is then mapped to one of the ASCII characters 'A'–'Z', 'a'–'z', '0'–'1', '+' and '/'.
Table. The character '=' is used to indicate how much padding was added to the input to make it a multiple of three bytes long. (See RFC 1521 for the mapping table, and for the details of the padding scheme.)

Base64 allows characters other than the 65 mentioned above (such as carriage returns or linefeeds) to be embedded in the encoded output, and specifies that they be ignored by decoding software. In fact, the standard specifies that no more than 76 characters be placed on an output line, thus requiring line break characters to be placed in the encoded output stream.

The “Down” routine of your implementation takes as input a sequence of arbitrary bytes stored in a BUF chain, and passes a BUF chain containing the result of encoding those bytes to the Down routine of the next lower protocol in the stack. The Down routine never indicators an error, no matter what happens. If it is passed a null reference to a BUF chain, it should simply crash. It always passes the result of encoding the data to the next layer. (Note that the result of encoding an empty message is an empty message.) Similarly, the “up” routine takes a properly-encoded complete message in a BUF chain as input, and passes the result of decoding that message to the next higher layer’s Up routine in a BUF chain.

Implementation considerations:

- You have a function to get the next byte of data from a BUF chain; that routine should remove (and free) buffers from the front of the chain as they are exhausted. (This is like turning a BUF chain into an input “stream”, and hides the messiness associated with buffer chains from the main logic of the protocol.)

- You have a function to append a single byte to a BUF chain; that routine should allocate and append a new buffer when the last buffer in the chain is out of space.

- The downFn() routine should read blocks of up to three bytes from the input into an integer and encode them as four output bytes. Encoding a block of fewer than three bytes means that the input is gone, processing is complete, and the BUF chain containing the encoded data should be passed to the next lower layer’s Down function.

- Bytes of the 24-bit quantum are in big-endian order. So if a is an (at least 32-bit) integer that contains the bytes read so far, the next byte b can be added by shifting and OR-ing, as follows:

\[
\begin{align*}
a & \leftarrow NBPB; \\
a & \mid= b;
\end{align*}
\]

where NBPB is the number of bits per byte, i.e. 8.

- The upFn() routine should read four input characters bytes at a time and translate them into three 8-bit bytes in a similar manner. Use a table for encoding; for decoding, it may be just as easy to use character arithmetic. (In Java, be sure you are using the character set “ISO-8859-1”, which matches what the standard requires, and also has one-byte character codes.)
Protocol 2: Length-based Framing

You will also implement a simple length-based framing protocol. This framing protocol accommodates frames containing up to 16,777,215 bytes in length. The length of the frame is contained in an eight-byte header, as depicted in Figure 5. The first three bytes of the header contain the sequence of ASCII characters SYN SYN SOH (i.e. byte values 22, 22, 1 [decimal]); this is a special marker that allows a receiver to find the beginning of a frame if it “loses synchronization”, e.g. in case a frame header is corrupted. The next three bytes are the 24-bit frame length, in big-endian order (i.e. LEN0 is the most significant byte, LEN2 is the least significant).

\[
65536(\text{LEN0}) + 256(\text{LEN1}) + \text{LEN2} \text{ bytes of payload}
\]

Figure 5: Frame Format

The last two bytes of the header are encoded to detect corruption of the frame length field. CHK0 is the 8-bit exclusive-or of the three LEN bytes. In other words, bit \( i \) of CHK0 is the even parity of bit \( i \) of LEN0, LEN1, and LEN2, for \( i = 0, 1, \ldots, 7 \). This can be computed in one line of C or Java (given appropriately-typed variables):

\[
\text{chk0} = \text{len0} \oplus \text{len1} \oplus \text{len2};
\]

CHK1 is the ones complement of the 8-bit ones-complement sum of LEN0, LEN1, and LEN2. The ones-complement sum is computed by adding the bytes using regular addition and then “wrapping” any carries out of the byte back around and adding them into the low-order bits. (This is an 8-bit version of the checksum used in TCP. See RFC 1071 for further details.) Correctness of CHK0 is verified by the bitwise-XOR of the three length bytes with CHK0 being all 0’s. Correctness of CHK1 is verified by the 8-bit ones-complement sum of the three length bytes and CHK1 being all 1’s.

The input to the framing protocol \( \text{upFn()} \) method is a BUF chain containing received data. The data consists of frames separated by garbage. The sequence of buffer chains presented in a series of calls to the framing \( \text{upFn()} \) method should be viewed as a contiguous stream of bytes. That is, the input may arrive in pieces (as it might, for example, over an asynchronous TTY line). The \( \text{upFn()} \) method’s job is to process the input, extract the payloads of the frames, and pass them unchanged to the next higher layer. The method searches the input for a correct header (i.e. the initial 3-byte sequence, followed by five bytes that include correct checksum bytes), computes the length \( l \), and collects the next \( l \) bytes. When \( l \) bytes have been received, next layer’s \( \text{upFn()} \) method is immediately invoked with the BUF chain containing them as input.

Because any given call to \( \text{upFn()} \) may only present part of a frame, **state must be maintained across calls** to \( \text{upFn()} \). In particular, the layer must keep track of (i) how much of the header it has seen, (ii) if a correct header has been parsed, how much more of the payload remains to be received. Of course, the portion of the payload received so far must also be stored. I strongly suggest that you use a simple finite-state automaton to keep track of how much of the header has been seen. (If you don’t know how to do this, ask me in class.) Once

Note that the garbage between frames may in general contain partial or incorrect headers; when an incomplete or corrupt header is encountered, it should be silently ignored. There is no upper bound on the amount of data in the chain passed in a single \( \text{upFn()} \) call. However, the BUF input to the \( \text{upFn()} \) method will **not** include more than one “packet” (i.e. there will be no “vertical” chaining of the kind shown in Figure 3).

The output of the \( \text{upFn()} \) method is a “packet” chain containing all frames whose reception was completed on that call. The \( \text{upFn()} \) method should call the next higher layer with these packets **one at a time**. The \( \text{upFn()} \) method should invoke the higher layer **only** when a frame is completed; otherwise it should return success.

The \( \text{downFn()} \) function performs the sender processing, i.e. adds the framing header to a given frame. The only work involved is determining the total size of the input payload; in general this requires traversing the entire BUF
It may be necessary to move existing data or prepend a BUF to the input chain in order to have room for the framing header; your upFn() method SHOULD NOT copy the data into a new chain. If a BUF chain containing more than 16,777,215 bytes is passed to the downFn() method of this framing protocol, the method should return the ERRORMORE error code, without freeing the BUF chain. No other errors should be signalled by the methods.

7 Other Coding Considerations

- Note well that there are no preconditions on the value of data received from the network. Therefore you must think defensively, and write “up” methods that are prepared to deal with any data; you must not abort the whole program because the input data was not correctly formatted—for example, a framing header was malformed, or an encoded Base64 frame did not contain the right number of characters. In these cases your code should take some recovery action (e.g. ignore the incorrect input and free the BUF chain) in order that the system can continue operation.

- For those who are unused to programming in C, note especially that data stored in BUF’s (and in general, transmitted over the network) is not null-terminated. The extent of the data in the buffer is indicated by the length field, not by the presence of any null character. (Also, the data itself may contain nulls, so inferring end-of-data when you encounter a 0 byte will lead to incorrect results.)

- Each method should free any BUF’s in the input chain that are not passed on to the next layer. If your code is not fastidious about this you will lose marks.

- Avoid unnecessary copying. Protocol implementations must be efficient, and copying data from one location in memory to another is one of the most costly operations. Do not move data if you don’t have to! For example, do not move data within a buffer unless you have a really good reason. It’s almost always cheaper and faster to allocate a new buffer and splice it into the chain when you need space in front of existing data.

- Your code should contain no constants like “20” or “2048”. Use descriptive symbolic names (#define or final) instead. If I find “naked” constants in your code, your score on the program will be reduced.

- Make sure you consider the corner cases: buffers containing zero bytes of data, input buffers containing large amounts of data, input provided one byte at a time, etc.

- You can test your code by making sure that when the downFn() methods are applied in sequence to a given input BUF chain, and then the upFn() methods are applied (in the reverse order) to the result, the resulting chain contains the original sequence of bytes. (You will have to write a simple layer to put at the bottom and/or top of the stack in order to save the chains produced.) In other words, the function implemented by the stack going upward should be the inverse of the function implemented going downward. Note however that in general going upward first and then downward may not produce the original data. For example, the Base64 specification allows a good deal of latitude with respect to line lengths and extra characters, so the input function must be able to handle frames that look very different from what the output function produces.)

8 Turning In Your Code

PLEASE FOLLOW THESE INSTRUCTIONS CAREFULLY. Failure to do so will make it harder to evaluate your program, and may affect your grade.

For C programmers: If you write C, you will turn in exactly four files, named exactly as follows:

- myLFrame.h: definitions of data types and constants for your framing implementation. (At the very least, you must define a state block data structure.)
• **myLFrame.c**: “init”, “up”, and “down” C functions matching the signatures of the fields of the `layer` structure. (Note that the names of these functions can technically be anything; please make them descriptive.)

• **myBase64.h**: auxiliary definitions (data structures and constant) for your Base64 implementation. If you do not need any such definitions, just include an empty file so “make” doesn’t choke when I try to compile your code.

• **myBase64.c**: C functions for the three methods, as well as declarations of the state variables.

**DO NOT DECLARE VARIABLES OR FUNCTIONS IN ANY .h FILE.** Header files, which may be included in multiple compilation units, should contain only definitions of data types and constants; any declarations should be “external”.

**For Java programmers:** You will turn in exactly two files:

• **LFrame.java**, which defines the public class `LFrame`, which implements the `Layer` interface.

• **Base64.java**, which defines the public class `Base64`, which implements the `Layer` interface.

Whatever language you use, **do not create any other files.** If you believe you absolutely require another file, ask permission, after explaining your reasoning to the instructor.

You will turn in a tar file called “`(yourname)-0.tar`” that contains **only the source files named above and nothing else.** You will be given instructions how to turn in the tar file.