

Lab: Layer 2

Background: Frame of Reference

In the previous lab, we learned how to broadcast bits by switching a light bulb on and off. This light bulb system constitutes the *physical layer* of our simulated network protocols, much like the cables, hubs, wireless transmitters, etc., form the physical layer we use to connect to the Internet. This physical layer is referred to as *layer 1*. The `BitHandler` class you developed already was the first step on our journey into *layer 2*: the *data link layer*. The most widespread example of a layer 2 protocol is Ethernet. In this lab, you'll implement our own Layer 2, according to the format your "Group" decided on.

So far we have used our `BitHandler` class to broadcast meaningless strings of bits. When we receive such a string, how can we know if it was meant for us, or who it came from, or if we have correctly identified the same bits intended by the sender?

Every *frame* (the term we use for a layer-2 packet) will have a single 1-bit preamble: a leading 0. The leading 0 is simply used for synchronization purposes.

Although each node on a LAN (segment) can see every frame that is sent, it is expected that only the designated recipient will examine the contents of the frame. All layer-1 `BitHandler` objects will see the bits, and pass them up to layer 2, but only the layer 2 object of the designated recipient will keep the frame – all others should drop it.

Occasionally, however, it will be useful to designate that a frame *is* intended for *all* nodes on the network. Your group's protocol should have a way to designate these kinds of frames.

To determine that a frame has been transmitted and read correctly, some redundant information must be tacked on to the end of a frame. Your Group's protocol must implement some kind of error detection mechanism: it could be a full-blown Cyclic Redundancy Check, or a single parity bit, or a short multi-bit parity field.

Step 1: Framed

Copy your files from lab1 into a new directory (or, you may copy the code from someone in your group who got a perfect score on the lab1). In this directory, create a new file/class called `L2GXFrame`, where X is replaced with A, B, C, or D, depending on which group you are in. The `L2GXFrame` stores an instance variable for each field in your layer 2 frame: e.g., the source and destination MAC addresses (probably stored as

integers), `vlanId`, `length`, etc., along with a payload `String`. Provide a constructor that takes in values you need, and stores them, then computing derived fields, like `length` and the error detection field value. Also, create getters for all fields.

In `L2GXFrame`, write a static method `toBinary` that takes in an integer value and an integer length, and returns a string of bits of the given length representing the given value in binary. You may assume that the value will fit in the number of bits given by length.

Also, write a static method `computeErrorCheck` that takes in a bit `String`, and computes the error checking value, of the correct number of bits.

Create `L2GXFrame`'s `toString` method to create and return the bit string corresponding to the whole frame. Don't forget to prepend the "0" preamble to the beginning of it.

Finally, if your protocol needs it, create a `public static int` constant called `BCAST_ADDR` and initialized it to the value that represents a broadcast address. This is going to be useful later.

At this point it would probably be wise to test your code in `L2GXFrame` to see if it creates legal-looking frames. Just comment out the code in `Test.java` and put in new code to create some `L2GXFrames` and print out the results of calling `toString()` on them.

Step 2: Getting a Handle on L2GX

Create a file/class called `L2GXHandler` (again, with `X` replaced by your Group's letter). This handler will be responsible for 1) sending a given `L2GXFrame` to layer-1 to be sent, and 2) receiving a string of bits from layer-1, and creating a `L2GXFrame` from them. When receiving a frame from layer 1, this class will also decide if that frame should be received or dropped and if received, passing the received frame to any object at an upper layer that has requested to receive frames.

To pass a frame to layer-1 to be sent, the class stores a variable `handler` of type `BitHandler`. To pass a frame up to a layer above, the class stores a `layer2listener` object of type `Layer2Listener`.

Create the class constructor to take a `String host`, `integer port`, and `integer` represents the unique layer-2 identifier for this object (perhaps called `macAddr`?). The constructor creates a new `BitHandler`, passing in the `host` and `port`. The constructor must then set itself ("this") as a listener of the `BitHandler`. (This is how we make the connection from layer 1 to layer 2.) Store the `macAddr` in an instance variable. Create a getter for the `macAddr`.

Create the `toString` method so as to return a string representation of the `macAddr` value.

Create a second constructor that takes only `macAddr` and uses default values for the other 2 parameters. Call the other constructor with these values. (See `BitHandler` for an example of this.)

Step 3: Picture Frame

Copy `Layer2Display.java` from `/home/cs/332/sp2016/lab2/`. The example constructor creates a couple of fields that can be used to create a frame and then send it. You will have to heavily alter this code to allow the user to specify the values you need to create your `L2GXFrame`. Also, fix the code so that the Layer 2 address (`macAddr`) of this display's handler is shown in the title of the `JFrame`.

Replace the comment `/* SEND LAYER2 FRAME HERE */` in `actionPerformed` with code for sending the appropriate frame. Do this by calling the `L2GXFrame()` constructor and then using the handler to send the frame. (Note that this code won't compile yet, as there are many missing pieces – including the `send()` method in the handler!)

Now, go to `L2GXHandler.java` and implement the `send()` method. This method takes a `L2GXFrame` and converts it to its string representation. Next, the code should repeatedly wait for `BitHandler.HALFPERIODS` until the lower layer handler is silent. When it is silent, it calls the handler's `broadcast()` method to send the packet. Note that with our implementation you actually don't have to worry about collisions coming up from the lower layer. If your code waits until the handler `isSilent()`, then starts sending, you won't see a collision.

Step 4: Parsing a Received Layer2 Frame

The code for generating a Layer 2 frame is done. But, we need code to receive a frame.

Add a constructor to `L2GXFrame` that takes in a string of bits and parses them to find (and store) the frame's addresses, payload, etc. (Writing a `toDecimal` method may be quite helpful here.) This constructor should also look for errors in the packet – does it start with the required "0"? Is the length correct? Does the error check pass? The code should throw an exception (`IllegalArgumentException`) if the packet is invalid for any reason. (The code does NOT check if the destination address is correct, however. That will be done elsewhere.)

Step 5: Hark! Layer2 Speaketh

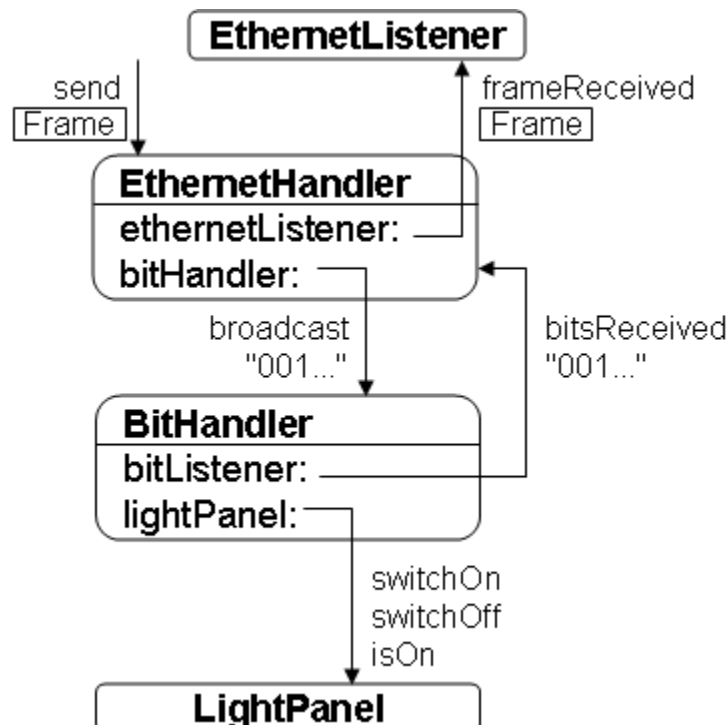
Create a file/interface called `Layer2Listener` that has the following method implemented.

```
void frameReceived(L2GXHandler h, L2GXFrame f)
```

Edit `L2GX2Handler` class so that it stores a (single) `Layer2Listener`. Add an API so that a listener can be registered: similar to how it is done in `BitHandler`. We'll make `Layer2Display` be a `Layer2Handler` listener in the next step.

First, however, we need a way for Layer 2 to receive a packet (from Layer 1). I.e., we have to make our `L2GXHandler` be a *listener* of Layer 1. Look at `BitDisplay.java` and notice how it registers itself to be a `BitHandler` listener. Add similar code to `L2GXHandler` to be a `BitHandler` listener. (Notice that this code only supports one listener for an object... At some point we may want to generalize this so that we can have a list of listeners.)

Whenever it receives bits, the `L2GXHandler` will parse those bits and create a `L2GXFrame`. Then, it should check if the frame is destined for itself. If not, it drops the frame and goes on. If the frame is for itself, it checks if there is a listener registered, and if so, passes that frame up to its `Layer2Listener`, by calling the listener's `frameReceived` method. Here is the big picture (in this picture, replace in your mind the word `Ethernet` with `Layer2` or `L2GX`).



Finally, modify the `Layer2Display` class so that it listens for layer2 frames. When a frame is received, the code simply builds up a string and displays it in the `displayField` field. You may make this text as nice as you want.

(In fact, if you want to make the `Layer2Display` much fancier, be my guest. It could, e.g., have fields that are used for generating frames, and separate fields that display the values of fields in received frames.)

Testing

Update your `Test.java` file so that you can test if your code sends and receives layer 2 frames correctly.

Submit all your code in `/home/cs/332/current/<yourid>/lab2`. Recall that Prof. Norman is a, how shall we say it kindly, “stickler” for documentation and clean code.

(Oh, and make sure you put your name at the top of your files in a nice header...)

Honors: Create A Layer2 Switch

Create a new class called `Switch`. Write a method called `addSegment`, which should take in the IP address and port number of a light system, and create a `L2GXHandler` for that segment. Now go ahead and complete the `Switch` class, so that it receives frames and forwards them according to the algorithm we read about in the book. Make sure that your `Switch` also times out entries in your port# \leftrightarrow MACaddr cache.

To test your `Switch`, you'll need to create multiple `LightSystems`. You can run multiple `LightSystems` on the same computer by constructing each `LightSystem` with a different port number. For example, you might run one on `LightSystem.DEFAULT_PORT` and another on `LightSystem.DEFAULT_PORT + 1`.