Error detection and correction

For this project, you will handle some bit-inversion errors introduced during transmission. Specifically, we will simulate communications across different media, some of which introduce different types of error. Your goal will be to write two different data link layers, each performing a different type of error management.

1 Getting the code

We will do these projects on the CS department server, castor.cs.amherst.edu, or on the workstations in Seeley Mudd 007. To get started, login to one of these systems and bring yourself to a shell, then follow these steps to get the starting source code:

```
$ mkdir cs281
$ cd cs281
$ tar -xzvpf ˜sfkaplan/public/COSC-281/simulator-project-1.tar.gz
$ cd project-1
```

This code forms a simulator. In this simulator, a partial network stack (using only the layers we’ve covered so far) is created for each of two hosts, connected by some (simulated) medium. At runtime, you can select from three possible media:

1. **PerfectMedium**: Connect two hosts with no errors ever introduced. The user specifies Perfect at the command line to use this medium.

2. **LowNoiseMedium**: Connect two hosts with infrequent, uniformly distributed bit inversions. The user specifies LowNoise at the command line to use this medium.

3. **BurstyNoiseMedium**: Connect two hosts with infrequent error bursts. During these error bursts, which have a maximum length set by a constant in the class, the probability of bit inversions is uniformly distributed and relatively high. The user specifies BurstyNoise at the command line to use this medium.

A physical layer object connects directly to a medium. There is only one type of physical layer. It accepts a sequence of bytes which it then sends, one bit at a time, across the medium. The receiving physical layer reconstructs the bytes, one at a time, delivering each complete byte to its data link layer.

Currently, there are two data link layers:

1. **DumbDataLinkLayer**: This particular data link layer uses start/stop tags and byte packing\(^1\) to frame any data that its network layer asks it to send. It creates a single frame for any sequence of requested bytes, no matter the length, and most critically, it performs no error management. To use this data link layer, the user specifies Dumb at the command line.

\(^1\)That is, byte-based escape codes for data that happens to match tag values.
2. **ParityDataLinkLayer**: This data link layer takes the input provided by the network layer and splits it into multiple frames (if the input is long enough). It uses the same *start/stop tags* as the DumbDataLinkLayer, but it also calculates the parity of the original data in the frame and appends that to the end of the frame for error checking. To use this data link layer, the user specifies `Parity` at the command line.

The *network layer*, of which there is only one type, simply sends a few messages via data link layer, and then (on the other host), receives those messages. Therefore, this network layer is merely a client to drive the data link layers, printing the messages sent and received to verify the accuracy of communication.

## 2. Running the simulator

After copying the code, you should be able to compile and run it. You must specify, on the command line, which *Medium subclass* and which *DataLinkLayer subclass* to use. You do so by providing the leading portion of the name of the subclass on the command line. For example, if you want to use the DumbDataLinkLayer with the PerfectMedium, you invoke the simulator like so:

```
$ java Simulator Perfect Dumb
```

You will then see messages printed by the network layer about messages sent and messages received. If you try the ParityDataLinkLayer with one of the imperfect media, you will see that, when an error is caught, that data link layer prints messages of its own about the error and **does not pass the faulty, received data to its network layer**. Your data link layers should behave similarly.

The NetworkLayer, in its `send()` method, contains a few sample strings, assigned to local variable `messages`, that it then tries to transmit via its data link layer. If you want to transmit different messages, simply change what is assigned to this variable.

## 3. Writing new data link layers

**Your Assignment:** You must create **two new data link layers** that are subclasses of the abstract `DataLinkLayer` class:

1. **CRCDataLinkLayer**: Use the CRC checksum method to detect errors on each frame. As above, when an error occurs, print an error message, show the (incorrect) data, and do **not** notify the network layer. A user should be able to specify the string `CRC` at the command line to use this data link layer. [Hint: Consult the text to select a generator polynomial to drive your CRC. Be sure that you choose one that can handle burst errors at least as long as those that the BurstyNoiseWire can introduce.]

2. **HammmingDataLinkLayer**: Use Hamming 1-bit error correcting codes on each frame. Correct any one bit errors and pass the corrected data to the network layer. A user should be able to specify the string `Hamming` at the command line to use this data link layer.
Note that all three layers should divide each message into smaller frames (unlike DumbDataLinkLayer). Be sure to modify the NetworkLayer to send messages that are long enough such that at least three frames must be transmitted for a given message.

**How to add a new data link layer to the simulator:** To add a new data link layer, simply copy the source code of one of the existing data link layer subclasses (e.g., ParityDataLinkLayer.java) into a new file of your own (say, CRCDataLinkLayer.java). Edit the file and rename the class, and then change the methods so that it detects/corrects errors differently.

Note that you do not need to change Simulator.java for it to recognize your new data link layer. You are welcome to look inside Simulator.java, which uses the command line input to form the names of subclasses, and then applies reflection to create objects of those classes. Thus, so long as your subclasses have the right kind of name (e.g., the names of data link layer subclasses end with DataLinkLayer), then the Simulator code will use them just as they do the existing classes.

## 4 The BitVector class

Attached is a new, helpful class for working with groups of bits, named BitVector. This class makes it easy to set the values of an arbitrary sequence of bits, where you can specify each bit by its index (just like an array). A BitVector object can be constructed from a byte array (that is, every bit of the byte array is used to initialize the vector of bits), or a BitVector can be converted into an array of bytes (all of the bit values are compacted into a sequence of byte values). Here is a listing of the methods for BitVector objects:

- **public BitVector ()**
  
The default constructor. Make an empty BitVector that contains no assigned bits. (Unassigned bits are assumed to be 0).

- **public BitVector (byte[] byteArray, int begin, int end)**
  
  Construct a new BitVector whose assigned bits are initialized from the bit values taken from the bytes starting at byteArray[begin] and ending at byteArray[end - 1]. From a given byte, the high-order bits are assigned to lower indices in the bit vector (that is, bytes are read from left to right as they are assigned into the vector).

- **public void setBit (int index, boolean value)**
  
  Set the bit at the given index to the given value (0 for false, 1 for true). All non-negative index values are valid; any assignment to a higher index than previously performed on the object simply extends its length, or the known portion of the vector, where this index is now the last known element of the vector.

- **public boolean getBit (int index)**
  
  Return the value (false for 0, true for 1) of the bit at the given index. All non-negative index values are valid. If an index is requested beyond the length of the vector, then false is returned. That is, the vector is assumed to be infinite, and those unassigned bits are 0 by default.
• public int length ()
  Return the length of the known portion of the bit vector. That is, if \( k \) is the highest index assigned using setBit, return \( k + 1 \).

• public byte[] toByteArray ()
  Compact the bit values of the vector into an array of bytes. Only the known portion of the vector is placed into this byte array. Bits at indices 0 through 7 are placed into byte 0 of the array, from high-order bit to low-order bit (that is, left to right), and then bits at indices 8 through 15 are placed into byte 1, etc.

5 How to submit your work

Use the cs281-submit command to turn in your complete Project-2 directory, including everything necessary to compile and run your simulator, like this:\[2\]

This assignment is due at 11:59 pm on Friday, March 8.
This deadline is final; there will be no further extensions!

\[2\]If these commands are perplexing, then search online for a tutorial on UNIX shell commands. You should also feel free to ask for help.