CSCI-351 Programming Assignment 3

Background

DC1 Global Enterprises is a company that builds, launches, maintains and leases communication satellites. The satellites can be used for many purposes, including broadcast TV, battlefield communications, and telephony. The global economic downturn has affected many of DC1's customers, and as a result, DC1 was unable to sell its services. Nearly half of DC1s satellites are in orbit, sitting idle.

The Scenario

The software on the satellites DC1 maintained was designed to be loaded from the ground, and never intended to be loaded while the satellite was 22,000 miles in orbit. Because of this, the initial design team used Trivial FTP without consideration for the error detection/correction that would be needed for a wireless link. One of DC1's crafty co-ops realized almost immediately that a very small supplemental firmware updating application could be loaded along side the existing firmware in the satellite so that the stock firmware could be programmed. An anxious engineer implemented these changes, and pushed his changes to the satellites and started updating the machines. Days after the updates were performed, several of the satellites started acting erratically (corrupting data, entering the wrong orbit, landing on people's houses, etc.).

It became obvious that the firmware being uploaded to the satellites had bit errors that caused the erratic behavior. Unfortunately, since there was not enough memory on the satellite to do a copy and replace of the firmware, the firmware would had to be programmed in place and the size, latency, and lack of reliable communications did not lend itself to an ARQ protocol.

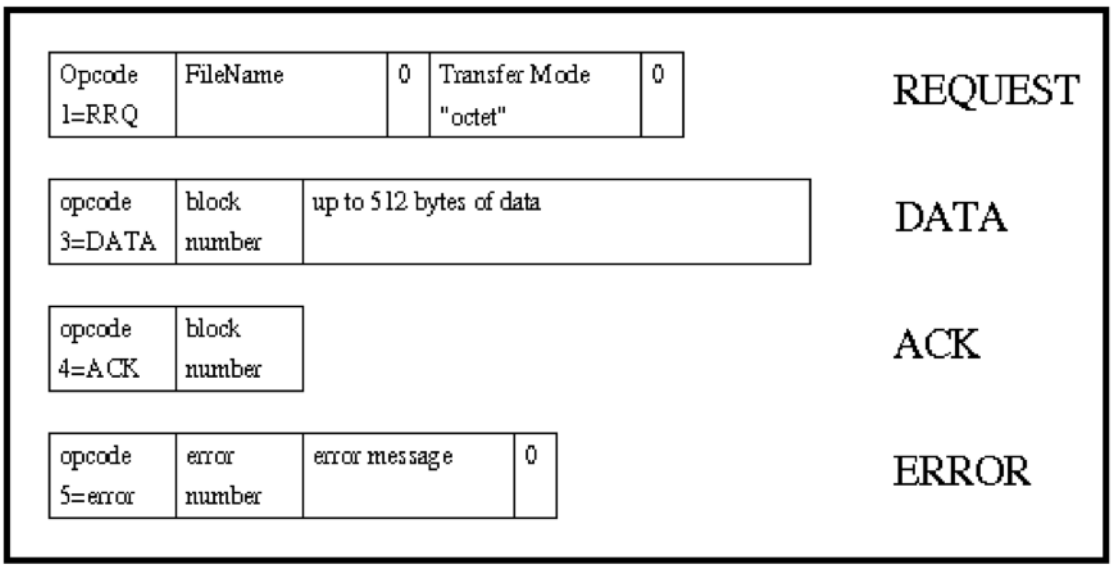
Since our crafty engineer went to RIT, he knew that hamming codes could be used to deal with single bit errors, which were the most common errors encountered in the programming process for these satellites, so will be used to correct the errors that occur in less than 1 out of every 1 million bits transmitted.

Our lofty engineer was able to implement the encoding algorithm, and implement the server and test it, but met an untimely demise when one of the satellites he initially reprogrammed got way out of orbit and fell on his head. With the client never being written, most of the satellites are still running corrupt firmware. DC1 is in a bind and needs you to update the decoding code to push it to the satellites. Unfortunately, this project is more or less trivial if done in c/c++, but in order to avoid bankruptcy, DC1 signed an agreement with Microsoft and Oracle to use their languages (.net and/or Java) in the satellites for an influx of cash.

There is one final problem. Our engineer never used source control, and when the satellite hit him, it also took out his laptop, where he had written the code. The server code is deployed and tested, but not available to look at, and the client has requested that new code be written on both sides to make maintenance easier for future engineers.

Requirements

You will need to write an extension of your tftp program in java or c# that changes the protocol slightly to support this scenario.
You recall that TFTP had the following formats:



Note: Opcodes are 2 bytes long (RRQ would be 01, ACK is 04)

Op code 02 was designed to write files, and was never implemented. This op code is now used to retrieve data with corrupted data (for testing). It is your job to reconstruct the data is a format that can be used.

Op code 01 is used to retrieve files with no errors, but parity bits are still interleaved. This will allow you to test extracting the data bits without worrying about errors. This is the code that was sent for the example below.

Op code 06 is used to send a NACK - a negative ACK. When you get an error on a packet and can't recover (as in 2 bit errors), you should send a NACK to re-requst the packet.

The algorithm we have selected chunks the data into 32bit blocks, which means that for each 32 bits, there are 5 parity bits for hamming code, 26 data bits, and one overall parity bit.

In order to get this to work, you will need to take the 512 bytes of data and break it into 32 bit blocks, you will then need to decode those values (which will generate 26 bit numbers, and put them back into the file). By now, it should be clear that for every 512 byte block of data, I should only get 416 bytes written to the file. Things you will need to worry about are bit ordering, network byte ordering, and extracting and combining bits from byte arrays.

For the purpose of the hamming code, we are using even parity, and bit[0] is the low order bit, while the overall parity bit resides in bit[31]. Note, this may not be the same order as the data. Your project will be built and run on the computer science machines. There is a server running on kayrun.cs.rit.edu, port 7000 that supports this encoded format. A list of files your can pull from this server will be provided.

The command line parameters should now take a option for error or noerror. The parameters will send op codes 02 and 01 respectively. With op code 02, 2 bit errors will occur on roughly 1/2 the packets, and you will need to send a NACK. With op code 01, there should never be the need to send a NACK. Additionally, a NACK should not need to be sent for a 1 bit error, as those can be corrected.

```
kiev> mono HammingTFTP.exe
Usage: [mono] HammingTFTP.exe [error|noerror] tftp-host file
kiev> mono HammingTFTP.exe noerror kayrun foobar.txt
kiev> mono HammingTFTP.exe error kayrun raven
kiev> ls -l raven
-rw------- 1 jsb fac 7041 Apr 11 15:42 raven
-rw------- 1 jsb fac   12 Apr 11 15:42 foobar.txt
```

The Data Details

Below is an example of data extracted using wireshark from kayrun, of the file foo.txt. (While this is the order of the data, you may find that the network stack is inverting the byte order for you)

The first 4 bytes I get back are the op code and the block number, which is to be expected. The next 4 bytes are:

```
10001000 00000001 01001011 10001001
```

These ARE in network byte order, as the data is actually:

```
10001001 01001011 00000001 10001000
```

from there, the hamming code is as expected. the 32nd bit (the one on the far left) is a 1, and the bits to the far right are the first of the hamming parity bits (bit position 1 is the far right)
if you extract the parity bits, you should get:

```
00010010 10010110 00000100 00
```

invert the bits inside the individual bytes and you get

```
in binary:
01001000 01101001 00100000 00
in hex:
0x48     0x69     0x20
```

These are your first 3 bytes of the file, with 2 leftover bits
Corresponding to:

```
Hi(space)
```

The next 4 bytes are

```
10101000 10100110 00101101 01010100
```

you first swap these bytes and get:

```
01010100 00101101 10100110 10101000
```

you again extract the hamming bits and you get

```
10101000 01011010 10011001 00
```

but you remember those 2 bits we had left over from before? Yup, they go on the right, so you get:

```
00101010 00010110 10100110 0100
```

swap the bit order inside the bytes again and you get

```
01010100 01101000 01100101 0010
```

which are the next 3 bytes:

```
The
```

It is your responsibility to ensure that your program is capable of loading the data correctly. This sample simply talks about the first 6 bytes of the file, and is data sent across the network. It is possible that your Datagram packet will invert the byte order.

The Parity Bit and Extra Data

You may recall that with Hamming code, bits 1, 2, 4, 8, 16 and 32 would be the parity bits for a 32 bit data stream. Due to the way that Hamming code works, parity bit 32 would apply only to bits 32-63. In this case, the parity bit only checks itself. To make the 32nd bit useful, we are going to treat the 32 bit is an overall parity bit, not related to the Hamming code. In order to properly detect 2 bit errors, you should check all the hamming bits, and correct any data. If the overall parity bit (which is checking ALL the bits) is still wrong (for even parity), you know that you have bad data, and should send a NACK. If in fact the parity bit checks out, you can send an ACK and start receiving the next pack.

The next issue you will likely encounter is null bytes at the end of the stream. Since null bytes at the end of the stream don't affect the actual data, those bytes should be dropped. In other words, if you extract the last 4 bytes of data, and byte 1 is 0x46 and bytes 2-4 are 0x00, you should not write bytes 2-4 to your file. This will ensure that the transferred file size is the same as the original.

Style

Write clean, modular, well-structured, and well-documented code. Here are some general guidelines:

1. Make sure that your name appears somewhere in each source file.
2. Each class and every method in the class should have a header that describes what the class/method does.
3. You do not have to document every single line of code you write. Provide enough documentation so someone who is not familiar with your program can figure it out
4. Names of classes, method, constants, and variables should be indicative of their purpose.
5. All literal constants other than trivial ones (e.g. 0 and 1) should be defined symbolically.
6. Functions should be defined, and kept small. Do not put all the code in a single function.

Testing

I do not provide test cases for you. It is your responsibility to ensure the program functions fully. It is permissible to share test cases with classmates, and discuss issues you have had, but it is NOT permissible to share code.

Submitting

Place all your source files in a single directory and zip the files up (in zip format). Submit the archive to myCourses, in the project 3 folder.

You may include a README file in this directory, if you wish, but no other files are acceptable.

Improperly submitted projects will results in a minimum of one letter grade deduction.