Programming Project #3: Reliable File Transfer

In this assignment you will be writing the transport-level code needed to implement the reliable data transfer protocol Go-Back-N as described in section 3.4 of your textbook. You will then use your Go-Back-N implementation to write a program that reliably copies a file across the unreliable network. I've already designed the file transfer protocol and written the server that will receive and save the file, your assignment is to write the client that will read and send the file to my server.

Unfortunately for us, the school has spent a large amount of money developing a highly reliable network. While using UDP might still result in the occasional dropped or corrupted packet, the likelihood of that happening is small enough that UDP alone is a poor tool for testing your implementation. Fortunately, I'm a terrible programmer. I have written my own version of the socket() interface that tends to corrupt or lose packets at an alarming rate. You will need to write you client using my socket library, overcoming the unreliable nature of my code. Correctly implementing the Go-Back-N protocol should achieve that result.

Functional Requirements

At a minimum, your submission must do the following to receive full credit.

- Basic requirements:
  - The executable should be called **`rft-client`**. Project name!
  - Your program must be written in C/C++, and must compile and run on Ubuntu 20.04LTS. We will test your submission in the virtual lab.
  - Your program should take three required command line arguments.
    - -h <hostname> # the name of the computer where the server is running.
    - -p <port> # The port number the server is using.
    - -f <filename> # The name of the file to transfer.
  - You must support an optional debug flag, setting the verbosity of debugging messages.
    - -d <num> # Number between 0 and 6, with 0 the least verbose.

      ↳ If not implemented, look at Project 1+2
- File transfer functionality:
  - Your program must use the datagramS structure as described in the **`datagram.h`** file. You may also use the utility routines in the **`datagram.cpp`** file.
  - Your program should communicate with the server using my communication interface, found in the **`network.h`** and **`network.cpp`** source files. See the section titled "The Unreliable Transport Library"

- The application layer protocol for sending the file is very simple. The server will interpret any datagram it receives as having the format of the datagramS structure. Assuming the datagram arrives in order and uncorrupted the data field of the datagramS structure is written to the output file. When a valid datagram with a payload length of zero is received, the server assumes the end of the file. It closes the output file and quits.

- Go-Back-N functionality:
  - The server implements the receiver half of the Go-Back-N protocol as defined in the Extend FSM found in figure 3.21 from the textbook.
  - Your client program should implement the sender protocol defined in figure 3.20.
  - You client must limit the window size (a.k.a. N) to 10 datagrams. In other words, you must never have more than 10 unacknowledged datagrams in the network at any given time.

    window size, N ≤ 10
  - The datagramS structure contains the following fields:
    - Sequence number # An unsigned 16 bit integer
    - Acknowledgment number # An unsigned 16 bit integer

      special int in C++
    - Checksum # An unsigned 16 bit integer computed using # the computeChecksum() utility function # found in **`datagram.cpp`**
    - Payload Length # Number of bytes in the data[] field. # Must be <= MAX_PAYLOAD_LENGTH
    - Data # Raw, binary data to be written directly to the # output file.
  - Your program should set the sequence number in each datagram it sends. There is no need for the client to set an acknowledgement number.
  - Your program should read the acknowledgment number in any datagram it receives. The server does not set sequence numbers; the sequence number found in received packets is meaningless.
  - The Go-Back-N protocol requires the use of an interrupt timer. Ideally it would be implemented using Linux kernel signals and/or multiple threads. Multithreaded, reentrant network code can be tricky to implement correctly. To avoid this complexity the communication library I provide uses non-blocking reads. It is highly inefficient, but it allows you to use the timerC class found in timer.h/timer.cpp. See the section on active polling for details.

    uninterrupted timer

Testing with the RFT Server

In my pub directory on Isengard there is a reliable file transfer server that you should use for testing (~promig3/pub/bin/rft-server). The server takes five command line parameters:

- -f <filename> The name of the file to write.
- -p <port #> The port number used to listen for datagrams.
- -c <corrupt #> The likelihood of corruption (a float between 0-1)
- -l <loss #> The likelihood of loss (a float between 0-1)
- -d <debug level> The debug level

solution for testing!

The sever program is not very complex. When you start it, it will listen for datagrams on the specified port. Any data received will be written to specified file exactly as it is received. When the server receives a datagram with a payload length of zero, it assumes the file transfer is complete. It closes the file and exits.

Hints and Suggestions.

The most common mistake I've seen students make is over-complicating their solution. The FSM found in figure 3.20 contains everything you need for transferring the file using GBN. There are less than 20 lines of code in that FSM. There will be some overhead in your program as you read the file and setup the network, but the heart of your implementation should not be any more complex than the FSM. Of course, this does not mean the assignment will be quick and you can put it off. Sometimes the shortest programs are the hardest to write and debug.

A few specific hints:

char* not str::string

- The input files will be raw binary data. Do not try to treat them as if they contain strings.
- Use a std::array with an initial size of 10 to store sent, unacknowledged, packets. This is the variable named sndpkt[] in the FSM.
  - My code looks like: **`std::array<datagramS, 10> sndpkt;`**

look @ FSM

- Use modular arithmetic when indexing the sndpkt array, ensuring there are never more than 10 datagrams in the array.
  - **`sndpkt[sequnum % 10].seqNum = nextseqnum;`**
- The server allows you to adjust the likelihood of corruption and loss. I recommend writing the file transfer part of the program first, without implementing the Go-Back-N part of the code. Test and be sure your program successfully transfers a file with when there is no loss or corruption. Then go back and add the Go-Back-N parts of the program.

What to submit.

This assignment is due by the end of the day on XXXXX. There will be a 1% per day penalty for late submissions.

You should submit a tarball of a single directory containing a makefile, a README.txt with your name and any information I need to compile the program and the source files needed to build your program. The single directory must be named with your username. The grader should not need to do anything other than untar your files, cd into your directory, type make and begin testing.

Do not include any core, object or binary files in the tarball. The Makefile I provided includes a target named **submit** that will create the tarball in the format we are looking for. All you need to do is:

$> make submit

In addition to functionality, you will be graded on the quality of the code, including readability, comments and the use of proper programing practices.

The Unreliable Transport Library

The unreliable transport library and other files you will need to create the program are available from github.

The datagram utilities

In addition to the datagram structure, the datagram source files contain:

- **`std::string toString(const struct datagramS &datagram)`**
  Returns the contents of the datagram as a string sutiable for printing to the terminal. Useful for debugging.

unsigned int16

- **`uint16_t computeChecksum(const struct datagramS &datagram)`**
  Computes a simple checksum over the seqNum, ackNum, payloadLength, and data[0..payloadLength] fields in the datagram. Note that it does not set the checksum field, it only returns the checksum value.

- **`bool validateChecksum(const datagramS &datagram)`**
  Computes the checksum value for the datagram (by calling computeChecksum) and compares it with the value stored in the checksum field of the datagram. Returns true if the values match, false otherwise.

unreliableTransport Class

The unreliableTransport class is the mechanism you will use to communicate with the server. I've abstracted out some of the mundane details of creating a UDP socket so you don't need to worry about them. An important feature of this class is that it performs non-blocking reads from the network. When you call udt_receive() it will return the number of bytes read or zero if there is no data available on the network. An exception will be thrown if there is an error. This style of network programing is called polling, it is inefficient but sometimes the simplest way to approach a problem. When polling reading data from the network will look like:

```
while (connection->udt_recieve(datagram) == 0) {
      if (timer.timeout()) {
             // deal with timout
      }
}
// process datagram
```

In addition to the default constructor/destructor the public methods of the unreliableTransport class are:

- **`unreliableTransportC(&hostname, &portNum);`**
  The constructor, it will create a UDP socket bound to the host and port passed in. An exception is thrown if there is an error.

- **`void udt_send(const datagramS &data) const;`**

Sends the datagram to the server using the unreliable transport layer. It will only fail if there is an unexpected network problem outside the programmers control. In this case an exception is thrown. You should catch and display the error message then exit the program.

- **`ssize_t udt_receive(const datagramS &data)`**
  Attempts to receive one datagram from the network. If there is data available it will return the number of bytes read, which should be sizeof(datagramS). If no data is avalable udt_recieve() returns 0. It will only fail if there is an unexpected network problem outside the programmers control. In this case an exception is thrown. You should catch and display the error message then exit the program.

timer Class

A simple pollable timer class built from the standard chrono library. As previously noted, the ideal implementation of Go-Back-N would use an interrupt to signal a timeout. To avoid the complexity of writing reentrant network code we are using polling. The code sample at the start of the unrelabileTransport section includes an example of how to use a polling timer.

- **`timerC(int milliseconds)`**
  Create a new timer with a default duration of **`milliseconds`**. The timer is not started, only created.

- **`void setDuration(int milliseconds)`**
  Update the default duration for the timer. Changing the duration of a timer while it is running is illegal and will throw an error.

- **`void start()`**
  Start the timer, duration **`milliseconds`** after start() is called, timeout will return true.

- **`void stop();`**
  Stop the timer. Stopping the timer doesn't really mean anything since we are not using interrupts, but it helps match the code in the GBN algorithm.

- **`bool timeout();`**
  If the timer is running and duration **`milliseconds`** have passed since start() was called, timeout() will return true. Otherwise, it will return false.