CS/ECE 438: Communication Networks Spring 2010
Machine Problem 2 Due: 9:00 PM, Friday, March 19th

Reliable File Transfer

Please read all sections of this document before you begin to code. Also, note that this is a challenging MP. **It is not possible to complete in just a few days**. You have lots of time – start early.

Summary

In this machine problem, you will develop a simple UDP-based client-server architecture for sending a file or multiple files from a server on one machine to a client on another. The client connects to the server. After accepting the connection, the server prints the client's IP address and port number in its `stdout`, spawns a client thread and starts sending the file to the client. The server has a maximum packet size of 1000B and so may need to fragment the file into multiple packets. The main thread of the server remains active waiting for other clients. After receiving the file from the server and saving it, the client prints the name of the file and its length to its `stdout` and closes the connection. The UDP communication channel you must use will randomly drop or corrupt packets. To compensate for transmission over this channel with unusually flaky delivery, you support reliable transmission by implementing a Sliding Window Protocol (SWP).

Objectives

The primary goal is to understand the issues and approaches used to build a reliable packet stream across an unreliable network. Specifically,

1. To implement fragmentation at the sender and reassembly at the receiver
2. To implement error checking
3. To implement a reliable sliding window algorithm

Guidelines

For this MP, you are encouraged to work in teams of two, although you may work alone if desired (using the same grading scale and thus doing twice as much work). Please find a partner as soon as possible. We will allocate a few minutes of the lecture on Friday February 9th to pair off people without partners. If you attend the lecture and make your lack of a partner known, we will pair you with another person or will make suitable accommodations. If you do not attend the lecture or fail to make your lack of partner known, you may have to work alone.

Collaboration, code sharing, and any form of group work with persons other than your partner are forbidden. Offenders will be dealt with appropriately. **Automated code-similarity analysis will be performed.** Every year, people have been caught cheating – all that is required to change that is honesty.

You are required to make sure that your programs have a valid makefile and compile, link, and run on the EWS machines. Programs suffering compilation or link errors when tested will earn no credit.

Check the newsgroup for updates and clarifications.

Please indent and document your code. Use meaningful variable names and follow other sensible style guidelines that enhance the clarity of your code.

All the materials suggested and provided for MP1 are also relevant here. You may find section 2.5.2 of Peterson and Davie useful for understanding the algorithm for SWP. You may also find Ch. 23 in the second edition of Stevens useful for thread programming. If you reuse ANY of the sample code from class, Peterson and Davie or Beej, you must acknowledge the source.

Specification

This problem requires you to develop a simple client-server model that implements the Sliding Window protocol (SWP) for reliable file transfer.

Client
The client must accept one required command-line arguments - the name of the server:

```
client <server>
```

A client first checks the syntax and parses the arguments, then translates the server name into an IP address. If any error occurs during this stage, the client prints a brief error message and exits. If all arguments are valid, the client opens a UDP connection to the server and begins reading. The first thing it should receive is the name and length of the file it will be receiving. Next, the client should start receiving the individual frames from the channel. The client must parse the framing information from the frame header to understand which frame it is actually receiving. The client will act as the receiver for the SWP protocol and send an ACK or a NACK frame to the sender, as defined by the operation of the specific SWP protocol that you implement. For example, if you implement selective repeat, the ACK frame can acknowledge the receipt of multiple frames at a time. The client then writes any acknowledged frame(s) to the output file in the correct order. After the client has written the last frame to the output file, it closes the connection. The client then prints the name of the file and its length in its `stdout` and exits.

Server
The server accepts a single command line argument: the name of the file to be sent.

```
server <input file name>
```

Control begins in the main server thread, which creates a UDP socket and prepares it for incoming connections. You may copy Beej's code for this purpose, but be sure to credit Beej. You should also pick a new port number to avoid collisions with other students. When a client connects, the main thread creates a new thread to manage the connection. After cleaning up any per-client state, the main server thread returns to waiting for client connections.

The per-client thread should first indicate that it is sending the file to the client by printing the IP address and the UDP port of the client on the screen. Next, the per-client thread must send the name of the file and the file length to the client. When sending the file data, the per-client thread MUST fragment the file data into frames before sending them to the client. Additionally, the per-client thread MUST implement a Sliding Window Protocol to ensure reliability during data transfer. Either Go-Back-N or Selective Repeat may be implemented.

During the operation of the SWP, the per-client thread will receive ACK frames from the client, update the send window and remove any acknowledged frames from the window. After the ACK frame for the last frame of the file has been received, the per-client thread cleans up any temporary state, closes the client socket, and exits.

Frame Format
To transmit individual frames, you should use a variable length frame format with a fixed-sized header. The size of the frame header is defined by you based on the information you need to pass between the sender and the receiver components of the SWP. The frame MUST contain a Checksum field for the provision of error checking. An example frame header might have fields for frame number, sequence number, frame type, and last frame indicator. The total size of the frame (including the header and trailer, if any) MUST NOT exceed 1kB. The details of the frame format that you have implemented must be explained in your Design Document.

Error Detection
You MUST implement CRC-16 as your error detecting code. CRC-16 should be calculated on the whole frame including the frame header and trailer (if your frame format has one). You can use page 92-97 of Peterson and Davie for reference about CRC-16.

Function mp2_sendto

To support the adversarial activity of introducing errors into the frames and also dropping frames along the UDP channel, we are providing a function named `mp2_sendto` that you MUST use instead of using the standard `sendto` function for sending any data through the UDP channel (no calls to any other routine to write data to a socket are allowed):

```
ssize_t mp2_sendto (int sockfd, void* buff, size_t nbytes, int flags,
        struct sockaddr *to, socklen_t * addrlen);
```

The `mp2_sendto` function has exactly the same format and semantics as `sendto`, but the latter will corrupt bits in the buffer and drop messages. Otherwise the function will simply write buff to a socket.

Use `mp2_sendto` just as you would use `sendto.` The header file `~cs438/MP2/mp2.h` contains a function prototype for `mp2_sendto`. Link the `mp2.o` file from the class directory into your client and server executables. For example, the link command for your server might appear as:

```
gcc o server mp2server.o ~cs438/MP2/mp2.o lsocket lnsl lpthread
```

You must link the file in the class directory rather than making a local copy to guarantee proper behavior when your assignment is graded. For interested students, source code is available in the same directory as the object file.

Threads

Multiple threads are necessary for the proper operation of the server. These include the thread for accepting connections from a client and the per-client thread.

You are required to use Posix threads to support concurrent operation (see, for example, Ch. 23 of Stevens). When using Posix threads, make sure that the first two lines of all source files include **`pthread.h`** followed by **`errno.h`**, and that all files are compiled with **`__REENTRANT`**:

```
gcc -c -D __REENTRANT  mp3server.c
```

and that all executables are linked with libpthread:

```
gcc -o mp3server … -lpthread
```

Posix threads **MUST** be used for this mp!

Congestion Control

To avoid congesting the network, you will implement a simplified version of TCP's congestion control algorithm. You should implement the two key parts of congestion control (more details on these schemes are given in Peterson and Davie):

Congestion avoidance: Upon receiving a packet loss, decrease the maximum window size by a half, and when all packets in the window are received, increase the window size by one. This is sometimes referred to as Additive Increase/Multiplicative Decrease, or AIMD: http://en.wikipedia.org/wiki/TCP_congestion_avoidance_algorithm

Slow start: Double the maximum window size for every RTT, by incrementing it by one for each received acknowledgement.

Upon every window size change output a "WINDOW SIZE" message to stdout indicating the change in window size, and the reason why the window size changed.

Include a compiler flag -DCONGCTRL that can be used to selectively activate or deactivate congestion control in your code. When the compiler flag is not specified, your code should execute the sliding window protocol as specified in the other parts of this document with a fixed maximum window size.

In your writeup, answer the following questions: (a) what is wrong with using loss as a signal of congestion in this MP? What change would you make to your protocol to address this problem? (b) Why do we need slow start? Why

isn't congestion avoidance enough? (c) What is the rationale behind performing additive increase and multiplicative decrease for congestion avoidance (as opposed to, say, multiplicative increase or additive decrease)?

Messages
To assist in debugging and testing, you should prefix any printed output with human readable messages. For example:

- **`ERROR: all error messages`**
- **`NO FILE: requested file not found`**
- **`ACCEPTED: valid request for existing file recv'd`**
- **`SENT #: sent packet/ACK #`**
- **`RESENT #: resent packet/ACK #`**
- **`ACK #: recv'd ACK #`**
- **`FRAME #: recv'd frame #`**
- **`REJECTED #: frame # rejected`**
- **`WINDOW SIZE: current size of sliding window (on every window size change, indicate the new size)`**

Design Document
You MUST submit a Design Document for this machine problem. The design document should be labeled with the names of both partners and consist of five parts.

Window Data Structures
Define the data structures you will use to keep track of the send and receive window information. On the sender side, for example, this structure should include the last acknowledgment received (LAR) and an array of outstanding frames. Provide versions of your data structures annotated with explanations of the purpose of each field.

Window Operations
Outline your strategy for processing frames and acknowledgments. Do not turn in C code. Rather, provide descriptive paragraphs, supplemented if necessary by pseudo-code like you would see in algorithms textbooks.

Frame Format
Describe the framing format you will use to identify the fragments of the file and manage the parameters for the SWP. You have to define and justify the fields and also mention their size.

Implementation Log
Discuss any interesting aspects of your implementation. As always, you SHOULD acknowledge the authors, if you copied sections of code from elsewhere (even if you modified it!).

Work Breakdown
If you worked with a partner, describe the breakdown of work. Provide details of who was responsible for what work. Specific areas of work to consider are design, implementation, debugging, testing, and documentation. If you worked alone, this section should be omitted.

Print out a copy of your design document and turn it in by the start of lecture on March 2nd. Electronic versions will not be accepted.

Testing and Grading
Testing your code to ensure it complies with the specification is your responsibility. This section is intended to give you some starting points and to warn you about some pitfalls.

Correctness and Functionality will be determined by the diff utility. Make sure you can transfer binary files, not just text files. You might test your program with some images.

Be careful with signed and unsigned values and with the edges of ranges. Testing with many values for each field is useful.

File sizes can vary from very small (on the order of bytes) to very large (on the order of billions of bytes). We do not want to spend weeks testing your code however, so you can safely assume that the files to be sent will fit into the machine's virtual memory.

The following mistakes will result in **NO CREDIT**!

- Compilation or link errors when tested by the staff.
- Failure to send all packets (data and control) through mp2_sendto**.**
- Sending packets that are more than 1000 bytes in length (including frame header).
- Implementing Stop-and-Wait instead of Go-Back-N or Selective Repeat.
- Using TCP.
- Violation of the academic honesty policy.

Hand in

In addition to the usual submission stuff, you need to submit a hard copy of your Design Document at the start of the lecture on March 19th.

Place your source code in your MP2 directory along with a **`Makefile`** that, on execution of the command **`make`**, causes the executables named **`mp2client`** and **`mp2server`** to be generated. Next, create a **`PARTNER`** file with the names and netids of both partners. Only one partner should turn in the MP. (If we get two submissions, we will choose an arbitrary one to grade.) The **`PARTNER`** file is essential for proper credit. BOTH partners should turn in a design document.

When everything is in order with your MP2 directory, you can transfer the directory electronically as follows:

First, remove all object files and executables from the directory, leaving only **source code**, the **Makefile**, the **`PARTNER`** file, and the **`README`** file. Submission of object files or executables will reduce your grade.

Next, from within the directory type: **`~ece438/Handin/handin`**

You may handin multiple times; each subsequent submission will overwrite previous ones.

MP 2 Grading Scheme

Total of 100 standard points possible plus 5 possible extra credit points.

Correctness and Functionality (60 pts)

- File transfer (7 points)
  - Correct and complete files sent to client (server source and client destination files will be diffed after transfer, no points will be given if files are not exactly the same)
- Fragmentation and Reassembly (7 pts)
  - Server fragments the file into a sequence of frames, client reassembles the frames into the file
- Error Checking (8 pts)
  - Implementation of CRC-16
- Implementation of SWP (19 pts)
  - Either Go-Back-N or Selective Repeat
- Implementation of Congestion Control (19 pts)

Design Document / Readme (15 pts)

- Design Document (15 pts)

General Behavior (25 pts)

- Concurrency: Server handles multiple clients
- Error Handling
- Command line arguments

- Error output from server and client to `stdout` (see handout for recommended message prefixes)
- Client and server exit and clean up gracefully

Deductions
Points will be deducted for the following:
- -10 Makefile with name 'Makefile' does not create ALL executables with single execution of 'make' command (with no arguments) on EWS machines
- -20 No 'Makefile'
- -10 Wrong executable (program) names
- -10 Handing in executable and/or object file
- -10 No PARTNER file (if working with a partner)
- -5 Poor code quality (no comments, vague variable/function names, code not indented)
- -30 A protocol other than SWP is implemented

No Credit
No credit will be received:
- if UNIX processes are used instead of `posix` threads
- if all frames are not sent through mp2_sendto
- if TCP is used
- if packets larger than 1000 Bytes are sent (including all headers and trailers)
- if any attempt to subvert/avoid the problem is used
- if there are compilation or link errors
- for failure to cite sources, *i.e.*, plagiarism

Extra Credit (5 points)
- Handle the transfer of multiple files (the command line argument will be a file with a list of files to transfer)