CS 360 Special Topics: Computer Networking

Spring 2007 **DUE: April 24, 2007**
Assignment #4 **25 points**

Building a Reliable Transport Layer

The focus of this assignment is to understand how TCP is implemented.

The Reliable Transport Layer

The reliable transport layer you implement (TCP360) will be based on TCP. This layer will be built on top of UDP. It will be your job to build a TCP packet, fill it with data received from the user, and send it off via a UDP socket. Your code needs to manage the send and receiver buffers, acknowledgments, retransmissions, reordering, and error checking. The code from the third assignment will need to be modified to use use this new transport layer rather the TCP provided by the operating system.

To stress test your transport layer, you can use the 360FileTransfer code provided on the CS360 website. *Your code will be tested against this application for grading!*

The function prototypes for TCP360 are defined in TCP360.h. These are the functions that are exposed to the user. You must implement these functions as specified in the header file. **Do not change the function prototypes or the file TCP360.h in any way.**

As part of your Reliable Transport Layer you need to implement:

- Three way handshake to open a connection
- Handshake to close the connection
- Send and Receive buffer for each connection (2 KB)
- Sliding window protocol
- Reliable transmission
  - lost packets (using a constant value for a timeout interval)
  - corrupt packets (via the Internet Checksum, RFC 1071)
  - out of order packets
  - duplicate packets

You **do not** need to implement:

- Persistence timers
- Fast retransmit
- Congestion control
- Estimation of RTT or D
- Nagle/fixes for silly window syndrome
- TCP Options

You will **NOT** need to implement this on circe. Your TCP360 code should NOT contain ntohl()/htonl() calls. You should not, however, remove these calls from your existing client and server.

Details:

If you receive an out-of-order packet, you may choose to either keep that packet around and send a cumulative ACK if the missing packets arrive, OR silently drop that packet. Any corrupt packets should be silently dropped, forcing a retransmission.

System Architecture

This project will consist of a number of independent parts. Some will be provided to you, and some you will need to implement. The parts will communicate via **Messages** (defined in Message.h which is provided to you) sent between each part via thread-safe queues. The components of the system are:

The Network Layer (provided)
The TCP360 Layer (you implement this)
A thread-safe queue (you implement this)
A Timer interface (provided)

A thread-safe Queue

You will need to implement a thread-safe queue to allow each component of the system to pass messages to other components. The interface you must write to is in Queue_TS.h. You MAY NOT alter this header file in any way. Refer to the comments for each function prototype to determine exactly what each function is to do. You need to augment the struct that represents the queue (**`struct Queue,`** defined in Queue_TSData.h) to contain pthread locking data.

A Timer Interface

The Timer interface allows you to create an alarm that will ring at some point in the future. The alarm will place an ALARM_EVENT Message in the specified message queue when an alarm goes off. This may be used to create an alarm to denote when a TCP packet's acknowledgment has timed out. For retransmission alarms, it is recommended that the receiving message queue be associated with a specific TCP360 connection. Your interaction with the timer interface will be through the functions **`createAlarm`**(), **`cancelAlaram`**(), **`TimerThreadStart()`**, **`TimerThreadStop()`**, and the messages placed in your message queue. When an alaram is created, an unique alarmID (int) is created. The ALARM_EVENT message contains this alarmID so you can distinguish between the various timers you need to have running. Other timing events that you may use this interface for include timers for the 3-way handshake to start a connection and the connection shutdown. Note that since this multithreaded environment has unpredictable scheduling, you may receive an ALARM_EVENT for an alarm you have canceled.

The Network Layer

You will not use UDP directly, instead you will access UDP via the NetworkLayer interface provided to you. This NetworkLayer creates a single UDP "connection" between two hosts and multiplexes all TCP connections between those two hosts on that one connection. Since UDP over Ethernet is ridiculously reliable, the NetworkLayer will (optionally) drop, reorder, duplicate, and corrupt packets randomly. The function prototypes are provided in NetworkLayer.h. Refer to the comments in that file to see how to use the API. You will need to call **`NetworkLayerStart()`** to initialize the network layer and **`NetworkLayerShutdown()`** to gracefully shutdown the connection (once, immediately before your TCP360 layer terminates). The function **`void NetworkLayerEnableErrors(int flag)`** allows you to enable or disable the random packet errors. See NetworkLayer.h for the flags you can use to enable individual errors. You can bitwise-OR the flags together to enable more than one type of error. When errors are enabled, they are enabled on ALL connections. It is recommended that you disable errors until you are ready to debug your error handling TCP360 code. The code you submit MUST call **`NetworkLayerEnableErrors(GARB_DROP|GARB_CORR|GARB_DUPL|GARB_REOR|GARB_SHOW)`**, which will enable all the errors and print a message when an error is introduced. The precompiled functions are in NetworkLayer.o which is available on the class website. A separate NetworkLayer.o is provided for the lab machines and zeus.

The NetworkLayer interfaces with the TCP360 Layer in two ways. First, to send data, the TCP360 layer uses the function call **`NetworkLayerSendto()`**. To receive data, the NetworkLayer starts a thread that listens on a UDP port. All the TCP data that arrives on that port is wrapped in a Message and pushed into the message queue in the TCP360 layer. It is the responsibility of the TCP360 layer to dispatch this data to the correct connection.

The TCP360 Layer

This where the bulk of your coding will be. This layer must support multiple TCP360 connections at once (think about your server). A thread should be used to read data off of the Message Queue (where incoming TCP360 packets from the NetworkLayer are placed) and dispatch those packets to the correct connection (via a Message Queue within each connection), based on the destination port and source address and port. Each connection needs to track its own send and receive window size and provide send and receive buffers. The functions that are exposed to the user (your MathPacket client and server) are found in TCP360.h. You must implement these functions as specified in the comments listed with the function prototypes. DO NOT alter the file TCP360.h in any way.

The TCP360 packet is defined in **`struct TCP360Packet`** in TCP360Data.h. This structure is similar to the actual TCP packet but is different in a few key ways. The TCP360Packet does not contain space for the TCP Options field. The TCP360 protocol does not allow for any TCP options. The packet may contain up to 1024 bytes of user data. The struct allocates this space statically as an array of unsigned chars. The struct contains an int that denotes how many bytes of the array are used. Your TCP360 layer must fill out this payload size correctly, the NetworkLayer will rely on it being correct and will set it correctly for packets it receives. A payload size of 0 is valid for an ACK packet.

Three-Way Handshake to Open a Connection

```
CLIENT                                   SERVER

SYN (SeqNo: X) -->
                                         <--  SYN ACK (SeqNo: Y,  AckNo: X+1)
ACK (SeqNo: X+1, AckNo: Y+1)  -->

DATA (SeqNo: X+2)  -->
                                         <--   DATA (SeqNo: Y+1)
```

Handshake to Close a Connection

```
HOST1                                    HOST2

FIN (SeqNo: Z) -->
                                         <--  FIN ACK (SeqNo: W, AckNo: Z+1)
                                         <--  FIN (SeqNo: W+1)
FIN ACK (SeqNo: Z+1, AckNo: W+2) -->
```

The MathPacket

The MathPacket and calculation stream will not change for this assignment. You still should be using **`Version 00000010`** of the MathPacket.

The key to testing this assignment will be checking to see if you receive the MathPackets in the correct order by examining the `Seq#` field.

The Client
The Server

The client and server should not change except to use the new TCP360 transport layer and to change the command line arguments. The server should take three arguments: the UDP port the remote TCP360 layer will listen on, the UDP port this TCP360 layer will listen on, and the TCP360 port that the server will listen on.

The client should take 5 command line arguments: the DNS name of the remote machine to connect to, the UDP port the remote TCP360 layer will listen on, the UDP port this TCP360 layer will listen on, the TCP360 port of the server on the remote machine, and the TCP360 port for this client to use (the TCP360 layer will not automatically assign ports to your client, your client must specify which port it wants to use).

Example:.

```
homer$ ./client_TCP360 zeus 10002 10001 99 88
zeus$ ./server_TCP360 10001 10002 99
```

What and How to Submit

You must submit all the .c and .h files you use for this project, as well as your Makefile. **DO NOT INCLUDE any .o files** in your submission. The source code should be well documented and conform to the Coding Standards. You will also need to submit ONE Makefile that will build your client as an executable file named `client_TCP360` and build your server as an executable file named `server_TCP360`. The names of the targets for these two files should be client_tcp360 and server_tcp360, respectively. The Makefile should also build both client and server by default if no command line arguments are given to the `(g)make` command.

You will need to roll all of these files, and the .svn directory, into a gzipped tar file named **PUNetID_cs360s07_PA4.tar.gz** and submit that electronically using the submit script on **zeus** as detailed in lecture. You also need to turn in a hard copy of all .c and .h files you created at the beginning of class on April 24th. Both the electronic and paper submissions must be made by 1pm, April 24th. *Please do not create a directory in your .tar.gz file!*

Additional Submissions

In order to keep everyone on pace for this project, you will have intermediate deadlines you will need to meet. All submission deadlines, *except the final one*, will be at **11 pm**.

April 6: You must submit your thread safe queue and a description of the architecture of your system (Arch.txt). Submit this electronically **`PUNetID_cs360s07_PA4_1.tar.gz`**, and schedule a short meeting with me to discuss your design of the system (Friday April 6-Monday April 9).

April 9: You must be able to send a TCP packet between two threads using the Queue_TS. (Roadmap step #3) Submit this electronically **`PUNetID_cs360s07_PA4_2.tar.gz.`** This should be a stand alone test case named sendMessage.c which will build an executable (when the Makefile is run) named sendMessage.

April 13: You must submit your TCP360 layer and be able to Open a connection. Submit this electronically **`PUNetID_cs360s07_PA4_3.tar.gz.`** (Road map step #5) Your client and server from project 3 should be used to open the connection.

April 18: You must be able to send data (which is free from introduced errors) between your client and server using TCP360. Submit this electronically **`PUNetID_cs360s07_PA4_4.tar.gz.`** (Roadmap step # 6)

April 24: Final project due! You must handle all the described transmission errors. Submit as described above. (Roadmap step #7)

How will this be graded?

See Programming Assignment #3!
Your project will also be tested against 360FileTransfer for stress testing.

Hints

You may need to look at the TCP RFC, the chapters in TCP/IP vol 1&2 describing TCP, and the description of TCP in your textbook.

You will not be implementing setsockopt for this project. If you used this function in project #3, remove the function call.

A reference server will be up at: zeus.cs.pacificu.edu UDP listen port: 9999 UDP send port: 10000. My server will be running on TCP360 port 9999.

Use your assigned ports http://zeus.cs.pacificu.edu/chadd/cs360s07/ports.html.

Don't stop when you satisfy an intermediate deadline! Keep working towards the next one!

In a complex project like this, you should commit to Subversion often! Name your project **PUNetID_cs360s07_PA4** in your Subversion repository so that I will be able to look in your repository and review your commits and commit messages. **This will be a graded portion of the assignment**.

The endpoints of communication are defined by (srcAddr:srcPort, destAddr:destPort).

You should one exactly one instance of TCP360 on each machine. Do not try to start up two clients on the same machine. Bad things will happen. You will need to manage the TCP360 port numbers, these are completely unrelated to the underlying UDP port numbers.

Sample Road Map:

Be sure to test each step before moving on!

1) Thread Safe Queue
2) Build a TCP packet
3) Send a TCP packet between two threads via a TS Queue.
4) Send TCP packet across the wire using the NetworkLayer/Receive TCP packet via TS Queue
5) Open/Close a connection
6) Transfer user data
7) Error handling in TCP360

I recommend standalone test cases for steps 1-4.

Sample client input/output:
This is the same as project #3.

Start on this today! Be prepared to demo your client and server in class on April 24th!