TCP for Minet

Project 2

Overview

In this part of the project, you and your partner will build an implementation of TCP for the Minet TCP/IP stack. You will be provided with all parts of the stack except for the TCP module. You may implement the module in any way that you wish so long as it conforms to the Minet API, and to the reduced TCP standard described here. However, Minet provides a considerable amount of code, in the form of C++ classes, that you may use in your implementation. You may also earn extra credit by implementing additional parts of the TCP standard.

The Minet TCP/IP Stack

The Minet TCP/IP Stack is documented in a separate technical report. The low-level details of how Minet works, including the classes it makes available to you, the modules out of which the stack is constructed, and how the modules interface with each other is documented there. Of course, it also doesn't hurt to look at the code. You will be given the source code to all of the Minet modules except for tcp_module. You will find udp_module.cc quite helpful to begin.

Your IP Addresses

Each project group will be assigned 255 IP addresses to use for the rest of the semester. These addresses are of the form 10.10.x.y, where x will depend on your group id and y will range from 1 to 255. These addresses are special in that packets sent to them will not be forwarded beyond the lab network.

Building Minet

For this project you will first copy the files to your own directory, then use the same build process as you used in project 1.

```
> cp /home/zqy272/EECS340/minet-netclass-w14-proj2.tgz <your directory>
> tar xvf minet-netclass-w14-proj2.tgz
> cd minet-netclass-w14-proj2/
> make clean;make
```

Three modules need root permission to run. As a result, go to bin/ sub-directory, remove the device_driver2, reader and writer module with user privilege.

```
> rm device_driver2
> rm reader
> rm writer
```

Next, create links to three specially prepared modules with root privilege (still under bin/ sub-directory).

> ln –s /usr/local/eecs340/device_driver2
> ln –s /usr/local/eecs340/reader
> ln –s /usr/local/eecs340/writer

Running and Testing the TCP Module

The TCP Module, which you will write in this assignment, will be compiled as a MINET module and does not run separately. To run and test the TCP Module, you need to start the MINET stack, which automatically starts up the TCP Module along with various other modules as documented in the Minet Technical Report.

Before you run Minet you will need to assign it an address from the range you were assigned for this project. To do this first execute *setup.sh* to generate the *minet.cfg* file. Whenever you switch to a new machine, you need to execute *setup.sh* again to generate the correct *minet.cfg* file for that machine. Next, edit *minet.cfg* and set MINET_IPADDR to whichever address you want to use from the range assigned to you, by editing the line MINET_IPADDR="10.10.x.y" (properly substitute x and y with numbers). Also, you need to grant write permission to fifos/ether2mon and fifos/ether2mux. Go to fifos/ sub-directory, type the following command.
> chmod a+w ether2mon
> chmod a+w ether2mux

Remember that Minet uses xterms to display its output. If you are logged into the TLAB host remotely, make sure an X-server (e.g. xwin32 or xming) is running on your PC and set the DISPLAY variable appropriately. You may find a helpful tutorial at http://pdinda.org/remoteunix/index.html

Type "./start_minet.sh" to run the Minet stack. Some xterm windows should popup (assuming all the modules including tcp_module are compiled), which show messages from the corresponding modules in the Minet Stack. Now, if you go to any other TLAB machine, you should be able to ping the IP address you've assigned to your Minet stack.

To stop the Minet stack after testing your TCP module, type "./stop_minet.sh".

Testing tcp_module

To test the tcp_module, you can run the "tcp_server" provided on your host and then use "nc" (netcat) from another machine to talk to it. The simple version of "http_server" should also run on top of minet stack along with "http_client" on another machine.

Example:
Assuming Minet stack is running as described in the previous section, type './start_minet.sh "tcp_server u 5050"'. This starts the tcp_server using the Minet stack for communication. Now you can talk to the tcp_server from another machine using nc:

Example (assuming tcp_server is running on the machine 10.10.5.2):
nc 10.10.5.2 5050

Read the man page on nc for more detail.

Dedicated lab Machines

You may use any of the TLAB machines, either from the console, remotely via ssh, or through a VNC session.

TCP Specification

The core specification for TCP is RFC 793, which you can and should fetch from www.ietf.org. In general, you will implement TCP as defined in that document, except for the parts listed below.

- You only need to implement Go-Back-N
- You do not have to support outstanding connections (i.e., an incoming connection queue to support the listen backlog) in a passive open.
- You do not have to implement congestion control.
- You do not have to implement support for the URG and PSH flags, the urgent pointer, or urgent (out-of-band) data.
- You do not have to support TCP options.
- You do not have to implement a keep-alive timer
- You do not have to implement the Nagle algorithm.
- You do not have to implement delayed acknowledgements.
- You do not have to generate or handle ICMP errors.
- You may assume that simultaneous opens and closes do not occur
- You may assume that sock_module only makes valid requests (that is, you do not have to worry about application errors)
- You may assume that exceptional conditions such as aborts do not occur.
- You should generate IP packets no larger than 576 bytes, and you should set your MSS (maximum [TCP] segment size) accordingly, to 536 bytes. Notice that this is the default MSS that TCP uses if there is no MSS option when a connection is negotiated.

Chapter 3 of your textbook also serves as an excellent introduction to TCP concepts and should be read before the RFC. **You will also find the TCP chapters in Rick Steven's book, "TCP/IP Illustrated, Volume1: The Protocols" extremely helpful. They will show you what a working stack behaves like, down to the bytes on the wire. Make sure that you read about and understand the TCP state transition diagram in 18.6.**

Recommended Approach

There are many ways you can approach this project. The only requirements are that you meet the TCP specification detailed above, that your TCP module interfaces correctly to the rest of the Minet stack, and that your code builds and works on the TLAB machines. We recommend, however, that you use C++ and exploit the various classes and source code available in the Minet TCP/IP stack. Furthermore, we recommend you take the roughly the following approach.

Important Reading

1. Read Chapter Three of your textbook
2. Read RFC 793 (Section 3 of the RFC contains a wealth of information related to implementation) and the Stevens chapters. The RFC contains a very detailed implementation guide for the TCP protocol. Understanding and following that can be of great help. In Stevens, Chapters 17, 18 and 21 (for timeouts) are especially useful. The State Transition Diagram (Figure 18.12) summarizes the major part of implementation in this project.
3. Read the "Minet TCP/IP Stack" handout. You will be able to understand the Minet code and terminology better after that.
4. The code related to this assignment uses a fair bit of C++ STL and C++ templates. So getting an overview of that would be helpful. Learn about the *deque* STL data structure.

Initial Phase (Understanding TCP Data Structures)

5. Fetch, configure, and build Minet if you have not already done so.
6. Examine the code in tcp_module.cc and udp_module.cc (both under src/core/ sub-directory). The TCP module is simply a stub code that you need to flesh out. It just connects itself into the stack at the right place and runs a typical Minet event loop. UDP module is a bit more fleshed out. It has almost exactly the same interface to the IP multiplexor and to the Sock module as your TCP module will have.
7. Extend the TCP module so that it prints arriving packets and socket requests. You should be able to run the stack with this basic module, send traffic to it from another machine using netcat (nc), and see it arrive at your TCP module. You may find the classes in packet.h, tcp.h, ip.h, and sockint.h to be useful.

   Minet.h: MinetInit(), MinetDeinit(), MinetSend(), MinetReceive() – for sending and receiving packets to and from Minet handles. Look at Minet.h for other useful functions

   packet.h : useful functions include GetPayloadLoad() – returns the payload from the packet as a buffer, PushFrontHeader (IP header in this project), PushBackHeader (TCP header in this project), PopFrontHeader (retrieves the IP header), PopBackHeader (retrieves the TCP header), FindHeader(packettype) – returns header in packet of packettype, packettype can be Headers::IPHeader and Headers::TCPHeader..

   buffer.h: Buffer is a very useful class. It helps you with text buffers, like your packet payload, the IP and TCP headers. Some useful functions are Clear(), GetSize(), GetData(target buffer, number of bytes, offset) – useful to retrieve part of or the entire buffer, ExtractFront(number of bytes) – removes n bytes from front of buffer and returns that, AddBack(buffer1) – adds contents of buffer1 to back of caller object. You can also find other useful functions in buffer.h

ip.h: SetProtocol(), SetSourceIP(), SetDestIP(),GetProtocol(), GetSourceIP(), GetDestIP(),SetTotalLength(), GetHeaderLength(), GetTotalLength()

tcp.h:
SetSourcePort(),SetDestPort(),SetSeqNum(),SetAckNum(),SetHeaderLen(),SetFlags(), SetWinSize(),RecomputeChecksum(). Set have their Get counterparts as well. It also has useful Macros like IS_ACK, IS_FIN, SET_ACK etc.

8. Now is a good time to familiarize yourself with Minet's various configuration variables. You should check out the various MINET_DISPLAY options too.
9. Learn how to use MinetGetNextEvent's timeout feature. You will be using this to implement TCP's timers. There is also a timeout variable in the class ConnectionToStateMapping (see point 10) which can be used to store the next timeout.
10. tcpstate.h and tcpstate.cc have code representing the state of a TCP connection. Study this carefully and understand what it contains. Think of a connection as being a finite state machine and consider using the states described in RFC 793. You may find the various classes in constate.h to be helpful here. In particular, your connection should have various timers associated with it. Your connection also has input and output buffers associated with it.
11. constate.h has a class (*ConnectionToStateMapping*) that maps connection addresses (the *Connection* class in sockint.h) to TCP connection state. Familiarize yourself with that. The *Connection* class represents the five tuple = (src_ip,src_port,dest_ip,dest_port, protocol). With this you can map each new connection with a particular TCP State. There is another class *ConnectionList* which can store a list (queue) of *ConnectionToStateMapping*s. This can be very useful to store the state and mappings of all the connections you have open presently in one data structure. Useful functions include *FindMatching*(connection) – returns the appropriate pointer if the connection is present in the connection list. Note that *ConnectionList* is implemented as the C++ STL data structure *deque (deque is a C++ STL data structure useful for storing a list of objects)*, thus all its functions like *push_front*(), *erase*(), *end*() etc are available as well.

Implementing TCP Interactive Data Flow

12. Add code to your TCP module to handle incoming IP packets. Begin by adding code to handle passive opens. Even without the Sock module, you can test this code by using a hard-coded connection representing a passive open. Note that the element of time enters in here. You will need to use one of your timers to deal with lost packets. You may find the classes in tcp.h and ip.h to be useful. (Passive open refers to the situation when you receive a SYN in LISTEN state. You need to send a SYN-ACK and set a timeout for the expected ACK from the remote side)
13. It is a good idea to write your own functions for sending and receiving IP packets that wrap calls to Minet. These functions can then form a framework for testing if your code works correctly in the face of packet corruption, drops, and reordering.

You can simulate drops by just not sending the packet with some probability. You can simulate corruption by randomly scribbling on a packet you're about to write with some probability. You can simulate reordering by keeping a queue of outgoing packets and changing their order in the queue occasionally. (*MinetSend* is used to send packets to the IP Layer or the Socket Layer)

14. Add code to your TCP module to handle active opens. Again, you do not need to use the Sock module here. You can hard code the active open for now. (Active open corresponds to the CONNECT socket call. You need to send a SYN to remote side, initialize your TCP variables like *sequence number* and the send window variables and also set a timeout)
15. Add code to your TCP module to handle data transfer. Again, note the element of time and think of how to implement your timers after *MinetGetNextEvent* responds with a timeout event. Remember that you do not have to implement congestion control, only flow control. A good approach to data transfer is first to implement a Stop-And-Wait protocol and get it working, and then extend it to do Go-Back-N. RFC 793 has important details on this.
16. Add code to your TCP module to handle closes. (You receive a FIN from the remote site or a CLOSE from the socket layer). The closing phase of TCP involves various states. See the TCP State Transition Diagram in Stevens and RFC 793 Section 3 for details on how to implement this part.
17. At this point, your TCP module should be able to carry on conversation with a hard-coded partner. Congratulations! You are finished with the most difficult part!

Implementing TCP-Socket Layer interface

18. Re-read the discussion of the interface between the Sock module and the TCP module in the Minet TCP/IP Stack handout.
19. Make sure you understand the SockRequestResponse class. SockRequestResponses will advance your connections' state machines just like IP packets do. They will also affect the set of outstanding connections (item 9).
20. Add code that keeps track of outstanding requests that your TCP module has passed up to the Sock module. Recall that the interface is asynchronous. When you send a request to Sock module, the response may arrive at any time later. The only guarantee is that responses will arrive in the same order that requests were sent.
21. Add code to support the CONNECT request. This should simply create a connection address to state mapping and initiate an active open.
22. Add code to support the ACCEPT request. This should simply create a connection address (with an unbound remote side) to state mapping and initiate a passive open.
23. Add code to pass incoming connections on a passively open connection address (one for which you have received an ACCEPT) up to the Sock module as zero byte WRITE requests.
24. Add code to support the CLOSE request. This should shut down the connection gracefully, and then remove the connection

25. Add code to support the WRITE request. This should push data into the connection's output queue.
26. Add code to send new data up to the Sock module as WRITE requests. Note that the Sock module may refuse such a WRITE. In such cases, the TCP module should wait and try to resend the data later. We are currently working on a better flow control protocol between the Sock module and the TCP module.
27. Verify that you are generating and handling STATUS requests correctly.

Testing your tcp_module

28. Now try to use your stack with an application. tcp_server and tcp_client should work.
29. Now you ought to be able to use your http_client and http_server1 from the project A, simply by running it with "U" instead of "K". The more advanced http_servers will probably not work since the socket module's implementation of select is buggy. Note that while a Minet stack currently supports only a single application, you can run multiple Minet stacks on the same machine or on different TLAB machines to test your code.

Extra Credit: Flow and congestion control

For extra credit, you may implement the flow control and congestion control parts of TCP as they are described in your textbook and in the other sources. Please note that while this is not much code, it does take considerable effort to get right.

Caveat Emptor

The Minet TCP/IP Stack is a work in progress. You can and will find bugs in it. There are three bugs that we are currently aware of

- The first IP packet sent to a new address is dropped. This is not actually a bug, but rather an implementation decision. What's going on is that ip_module will drop an outgoing packet if arp_module has no ARP entry for the destination IP address. However, arp_module will arp for the address and so the next packet sent to the destination address will probably find an ARP entry waiting for it. One way to "populate" arp_module so that you don't have to worry about this is to ping your stack from the destination IP address.
- In some situations, reusing connections is difficult due to a bug in the Sock module. The work-around is to rerun the stack on every connection. We will fix this eventually.
- The socket module's support for minet_select() is quite buggy, so it is highly likely that select-based applications will not work.

Mechanics

- Your code must function as a tcp_module within the Minet TCP/IP Stack, as described in a separate, eponymous handout.
- Your code should be written in C or C++ and must compile and run on the machines in the TLAB.

- Try to confine your changes to tcp_module.cc and new files.
- Project 2 will be due by midnight on 2/21. Please email your project to networkingta@gmail.com. You will be expected to provide tcp_module.cc and a README. If you modify our Makefile for some reason, you should hand that in too. The README should include the names of the project team, a brief specification of work undertaken by each member, and anything specific about your submission that you want to inform the grader.
- We will expect that running make in the Minet directory will generate the executable tcp_module (under bin/ sub-directory) and that this module will meet the specification described in this document and in the "Minet TCP/IP Stack" handout.

Hints

In your TCP Module code, output enough DEBUG messages, so that you can know what your TCP module is doing at each stage. Print information about each outgoing and incoming packet.

Things That May Help You

- RFC 793 is essential.
- Chapter 3 of your book. Section 3.5 is a good introduction to TCP. Sections 3.6 and 3.7 are about congestion control. You should read them, but you do not have to implement congestion control.
- **Rick Stevens, "TCP/IP Illustrated, Volume1: The Protocols"**
- Doug Comer, "Internetworking With TCP/IP Volume I: Principles, Protocols, and Architecture"
- Rick Stevens, "Advanced Programming in the Unix Environment"
- The handout "Unix Systems Programming in a Nutshell"
- The handout "Make in a Nutshell"
- The handout "The TLAB Cluster"
- The C++ Standard Template Library. Herb Schildt's "STL Programming From the Ground Up" appears to be a good introduction
- GDB, Xemacs, etc.
- CVS (http://www.loria.fr/~molli/cvs-index.html) is a powerful tool for managing versions of your code and helping you and your partner avoid stepping on each other's toes.