Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.033 Lab, Spring 1995-96

Handout 3 — Simple TCP

February 6

Recitation: Feb 16

Schedule design meeting with TA around Feb 22

Due date: March 21

1 Introduction

The goal of this system project is to improve your understanding of networking, obtain some practical experience with implementing networking code, and to get acquainted with TCP, the transmission protocol used in the Internet. In short, your task is to write a C library that implements the core of the TCP protocol; to make it clear that we are not asking for a complete TCP implementation, we will call the protocol simple (broken) TCP, or STCP.

This handout explains what you need to do to complete this project. It specifies STCP, the interface of your C library, suggests an implementation strategy, explains how you can test your library, and what to hand in.

2 Simple TCP

The STCP protocol is a subset of the TCP protocol that provides enough of TCP that you should be able to talk to a regular TCP server. STCP provides reliable stream-based connections. It is able to recover from lost packets, duplicate packets, and corrupt packets. To achieve high performance STCP employs a window-based flow control. STCP differs from TCP in that it lacks support congestion control and security.

TCP is defined by a series of numbered Request for Comments (RFCs). RFC793 specifies most of the protocol. Other RFCs detail strategies for congestion control etc.; a compliant TCP has to implement all of these strategies. (RFC122 states what a compliant TCP implementation looks like.) Most of the RFC are on the web (see http://www.cis.ohio-state.edu/hypertext/information/rfc.html).

Implementing a compliant TCP is a difficult and a labor-intensive task, so we ask you to implement only the core subset of it, which is specified in RFC793. In short, your library should implement RFC793, except for precedence and security. The rest of this section gives a brief overview of what we expect, but for details you should read and study RFC793.

2.1 Interface

The basic communication model is the one used in the UNIX networking programming exercise (see handout lab 2): a client opens a connection with a server, sends and receives data over this full-duplex connection, and then closes the connection. To support this communication model your library should provide the following five functions:

```
typedef unsigned long uint32;   /* 32 bit integer */
typedef unsigned short uint16;  /* 16 bit integer */

void *stcp_connect(uint16 dst_port, uint32 ip_dst_addr, uint16 src_port,
        uint32 ip_src_addr);
void *stcp_listen(uint16 src_port, uint32 ip_src_addr);
int stcp_write(void *handle, void *addr, uint16 sz);
int stcp_read(void *handle, void *addr, uint16 sz);
int stcp_close(void *handle);
```

Servers call `stcp_listen` to signal that they are ready to accept a connection; `stcp_listen` takes as arguments the server's IP address and port. Clients call `stcp_connect` to open a connection; `stcp_connect` takes as arguments the IP port and address of the server, and the IP port and address of the client. The ports are necessary to allow one to have multiple TCP connections per machines. A connection is uniquely specified by the quad-tuple (server port, server address, client port, and client address). `stcp_listen` and `stcp_connect` return a handle that is to be used to identify the connection in subsequent calls.

Once a connection is created, the client and server can send data over it using `stcp_write` and `stcp_read`. Read takes a handle that identifies a connection, a pointer to a buffer where the incoming data has to be stored, and the size of the buffer. Read blocks until a full buffer is read or until a connection is closed by the other side. Read returns the number of bytes read. A read on a closed connection should return an error value.

Using `stcp_write` data can be sent over the connection. It takes a handle that identifies the connection, a pointer to a buffer containing the data to be sent, and the size of the buffer. Write returns the number of bytes written. Write on a closed connection returns an error.

A connection is closed by calling `stcp_close`. Close indicates that the caller will not write any more data on the connection. Until the other side closes, the caller can still read data from the connection. Thus, closing a connection turns the connection from full-duplex to uni-directional. Only when both sides close the connection, can the state associated with the connection be freed.

A client or a server may use multiple connections concurrently.

2.2 Protocol

The STCP protocol is the core of RFC793; STCP has exactly the same states as TCP. You need to implement at least the following items from the RFC:

1. Implement the state diagram as described in the RFC.

2. Implement the flow control as described.
3. Implement retransmission as described
4. Implement duplicate and reordering detection as described.
5. Implement the check sums. We will provide you with the internet check sum routine.

The following items can be ignored:

1. Ignore reassembly. (i.e., assume that all bytes of a STCP segment arrives completely in a single packet or none of the bytes arrive.) Never send something larger than the maximum segment size, which should smaller than maximum size of an Ethernet packet, if you are sending your packets over Ethernet (see below for more details).
2. Ignore security and precedence stuff.

Of course you should feel free to implement reassembly, if you like to do it for fun.

3 Implementation strategy

You should be able to implement STCP in about 2,000 lines of commented easy-to-read code (our implementation is in that ball park). So, if your implementation is, say, smaller than 100 lines or more than 4,000 lines of code, something is probably wrong.

There are two main implementation principles:

- Be liberal in what you accept, conservative in what you send.
- Be a good citizen: be conservative in the number of packets you are sending.

The rest of this section describes a possible plan of attack. It is assumed you are familiar with UNIX network programming through the lab exercise (see lab handout 2).

Study RFC 793 very carefully. Design and implement the state diagram in the RFC as a state machine. State machines are a good and convenient way of implementing networking protocols in general. The other important part of your design is the STCP control block you have to maintain for each connection. In addition, think about how you will implement retransmission, duplicate detection, and timers.

Once you have completed your design, implement the state machine and the TCP control block. Use the server/client skeleton from UNIX network programming exercise to send and receive packets. Assume initially that the network is reliable and does not reorder packets (the client/server skeleton code guarantees that because it setup a connection of type SOCK_STREAM instead of SOCK_DGRAM). Test this implementation. Add the code for checksumming. Test your implementation again.

Once you have a complete implementation for a benign environment, add code for dealing with duplicate packets, reordered packets, and retransmission. Now, if you turn the UNIX socket from SOCK_STREAM to SOCK_DGRAM your code should keep on working.

4 Testing strategy

Correctness of your implementation is crucial. Distributed programs are often hard to test, since they tend to be non-deterministic (i.e., each time you run the program, it may follow a different path through your code). This section describes a possible testing strategy.

Write test programs to setup a connection, and read and write data. Use these programs to test first phase of implementation (assuming network is reliable).

Then, turn on the drop rate in your library and test your reliability code. The drop rate is a global variable in your library that has values ranging from 0 till 100, corresponding to the percentage of packets your library throws away. In normal operation, the drop rate is zero. To test your code, however, you should set it to other values. Use a similar strategy to duplicate packets to test if your library can deal with those appropriately.

Once your code is working reliably, test the performance of your implementation. In particular, measure the latency of pingpong-ing a byte and measure the throughput for large data transfers. Also, count the number of packets you are sending for, say, a one-megabyte transfer. To show that your performance degrades gracefully when network packets are dropped, measure also the throughput under different drop rates. Explain your numbers. (To measure the performance of your implementation use one of the timing functions provided by UNIX, see manual page for "times".)

5 Sending packets and other UNIX stuff

To send and receive packets across a network you can use the skeleton client/server programs from the UNIX network programming exercise. The programs as written setup a reliable stream, which is in fact implemented using the TCP protocol. So, once your TCP implementation is working, you will have TCP running over TCP. In the real world, you would run you TCP implementation directly over IP, but the people running Athena are not very excited about this approach for the lab. Of course, you should write your software in such a way that your TCP implementation minimizes the number of dependencies on operating system specifics so that you can easily port your implementation to other network layers.

To turn the socket stream into an unreliable stream, you can should change SOCK_STREAM into SOCK_DGRAM in the socket calls in client.c and server.c. When called with SOCK_DGRAM, UNIX will open datagram connection, which is implemented using the Unreliable Datagram Protocol (UDP). UDP packets may get lost or maybe reordered. In addition, there is a maximum size on the data that you can write to a UDP socket (which is 8 Kbytes). To avoid fragmentation of UDP packets you should write packets small than the maximum size ethernet packet, which is 1514 - size of Ethernet header (14 bytes) - size of UDP header (8 bytes). (Don't use the constants directly in your programs, but use the C expression "sizeof struct udp".) Of course, if you are sending your packets across another network than Ethernet, you need to substitute the appropriate value.

Implement timers using signals (see manual page for "signal"). When the timer goes off, let the signal function set some global flag in the STCP control block, which is tested periodically by the main program. This is a kludge but works.

If you are running a PC with Linux in your dorm, feel free to use it instead of Athena. If you really want to get ambitious you can modify your Linux kernel to implement TCP directly over IP.

6 Further reading

There are many books written on TCP [1, 3]. You might want to check them out if you find the RFC too dense. Source for various implementations of TCP exist (e.g., NetBSD and Linux), but most of them are fairly hard to understand (up to the point that there are number of books that explain these codes line by line, e.g., [2]). Most RFCs can be found at http://www.cis.ohio-state.edu/hypertext/information/rfc.html.

7 Suggested schedule

On Friday Feb. 16 a recitation will be devoted to explaining this first project in more detail. Please, read this handout carefully and come prepared with questions, if you have any.

In the week of Feb. 22, schedule with the TA a short meeting to discuss your design. Come prepared to the meeting with a design and be ready to answer questions about it.

The due date of this first project is Thursday March 21. The paper describing your software can be submitted as the paper for case study 1. You do *not* have to write both a paper about your software and a paper on case study 1.

8 What to hand in?

Send us a pathname to a directory in your Athena locker that contains the source code (including Makefile) and documentation.

In addition, hand in a hard-copy paper, which documents your project. The documentation should be written as a short paper and will be forwarded to the writing program, if you like.

Your documentation should also provide insight in the performance of your implementation.

References

[1] D.E. Comer and D.L. Stevens. *Internetworking with TCP/IP (II): Design, Implementation, and Internals*.

[2] Stevens. *UNIX network programming*.

[3] A.S. Tanenbaum. *Computer Networks*.