Packet-oriented Communication Using a Stream Protocol

or

Making TCP/IP on Berkeley UNIX a little more pleasant to use

Don Libes

National Institute of Standards and Technology
Gaithersburg, MD 20899

ABSTRACT

The only DoD protocols supporting the Transport Layer in the OSI Model are UDP and TCP. UDP is packet-oriented while TCP is stream-oriented. It is often useful to mix characteristics of both. This paper describes a software package that simulates the useful properties of UDP while using TCP. Also included are some functions for easier establishment of connections than by using TCP/IP primitives.

In addition, some observations gained during the experience of implementing this on a 4.2BSD UNIX system are described.

Keywords: packet, stream, communication protocol, TCP/IP, UDP, TCP.

Introduction

This paper describes software that performs packet-oriented communications using a stream protocol. Conceptually, this is a simple idea, yet users are expected to rediscover the solution each time, repeatedly reimplementing and debugging a wheel that should be built a single time.

The software was motivated by a project [Libes 85] that simulated a common memory in a local area network. The software described here was implemented in a Berkeley UNIX environment. The second half of the paper will elaborate on problems specific to this environment.

Background - TCP and UDP

The OSI model [Stallings 88] consists of seven layers. The fourth layer is the Transport layer. This layer controls end-to-end functions of network processes such as routing and ensuring the integrity of data between endpoints. There are several designs for doing this, but one of the most prevalent is provided by the DoD (DARPA) protocol suite commonly referred to as “TCP/IP”.

TCP/IP includes two general purpose transport-level protocols, namely, UDP and TCP

UDP (User Datagram Protocol) has the following characteristics:

1. unreliable - No guarantee is made of delivery. This is not perjorative. Rather, the application typically knows that guaranteed delivery is not necessary or may be too expensive. For example, a process monitoring power usage may be able to extrapolate missing reports sufficiently well for its use.

   In actual practice, data can also arrive out of order. I.e., packets sent later can arrive before packets sent earlier. Also, packets may arrive twice.

2. short - A maximum amount of data can be passed in a single application output request. This is generally set to the capacity of the underlying Data Link and Physical layers, again for the same reasons as above. A realistic example is that the data portion of an Ethernet packet can be at most 1500 bytes.

3. packet boundaries preserved - Since packets are short enough, applications are expected to provide enough buffer space for an arbitrary size packet. Upon completion of input, the receiving application is guaranteed to have exactly what the sending application output.

In summary, UDP is an extremely primitive transport mechanism, providing excellent efficiency due to its lack of demands upon the lower levels of the communications software. Sometimes this is a good match, such as when used by `rwho`(1), a BSD program that monitors active users on a LAN. In contrast, TCP (Transmission Control Protocol) requires more processing power and generally consumes a much larger amount of buffer space. But its benefits are often necessary. TCP has the following characteristics:

1. reliable - Delivery of data is guaranteed. In addition, input data order is the same as output.

2. long - Data transfers may be fragmented by lower levels of communication software but this occurs transparently.

3. boundaries are not preserved - reception of data gives no indication of how it corresponds to units of transmission. The application is presumed to handle this or not need it. Much like UNIX files, communication is simply assumed to be a “stream of bytes”.

Obvious hole

Mixing and matching attributes from UDP and TCP will create different styles of communications. For example, a packet protocol guaranteeing certain reliabilities would be useful, as would a packet protocol with longer maximum lengths. Yet these and other styles have not been deemed popular enough to create yet more protocols. Of course, any of these protocols can be use to simulate any other protocol, given a sufficiently powerful software set and adequate buffer space.

How it works - the basic idea

Our technique chose to follow the simulation idea. Our software simulates a completely reliable packet protocol with unbounded-length packets. The basic idea is that user data is stored in an envelope demarcating packet boundaries. This envelope is given to the TCP service for delivery. At the receiving end, one complete envelope is removed from the stream. The data is extracted out of the envelope and handed to the user. Thus, the receiver atomically and reliably receives exactly the information that the sender sent.

The packet format is as follows:

	Length of data MSB first	Data
Bytes	0–3	4–2^{32}

UNIX `read` becomes `sized_read`, etc.

In the UNIX environment, interprocess communication appears as normal I/O or file system activity [Leffler 83]. That is, `read` and `write` are used to read and write data, respectively, into and out of the process space.

Two new calls were constructed, taking exactly the same parameters as `read` and `write`. They are declared as follows:

```
int cc = sized_read(fd,buffer,maxsize);
int cc = sized_write(fd,buffer,size);
int fd;
char *buffer;
int maxsize;
```

`sized_write` packetizes the given buffer and sends it over the TCP stream described by `fd`. `fd` is the stream, `buffer` is the user's data area, and `size` is the number of bytes in the data.

`sized_read` is analogous to `read` except that `sized_read` will only read one packet at a time. `maxsize` represents the maximum size of an acceptable packet.

Exceptional conditions are a mess

If successful, both `sized_read` and `sized_write` return the number of characters in the packet. This may include zero characters (which does not signal an exceptional condition). Various problem are indicated by returning a -1. Currently, such problems include:

- bad packet header
- writer closes connection (i.e., dies) while in middle of packet
- reader closes connection (i.e., dies) while writer is writing

4.2BSD UNIX has always had a confusing array of ways in which `read` and `write` return error reports. For example, if the writer dies, the reader may receive either a `0` from `read`, or a `−1` with `errno == ECONNRESET` depending on the circumstances. If the reader goes away while writing is occurring, the `SIGPIPE` software signal will be sent to the writer. If the reader hangs and the writer continues to write, eventually kernel buffers become exhausted and the writer will be halted.

There is no question that the Berkeley designers made many compromises when integrating the TCP/IP software into their system. If anything, our software ameliorates some of these problems and does not provoke any new ones.

Nonetheless, the one case of a software signal being returned stands as is. If the application wishes to protect itself against a recalcitrant communicator, it should surround `sized_write` calls with alarms. The following code illustrates such protection.

```
#include <setjmp.h>
#include <signal.h>

jmp_buf write_context;

/* abort write() if recvr closes pipe as we are writing */
void sigpipe_handler()
{
        longjmp(write_context,SIGPIPE); /* abort write() */
}

/* abort write() if recvr refuses to read it fast enough */
void sigalrm_handler()
{
        longjmp(write_context,SIGALRM);
}

main() {
        signal(SIGPIPE,sigpipe_handler);
        signal(SIGALRM,sigalrm_handler);
        ...
        switch (setjmp(write_context)) {
```

```
        case 0:
              alarm(timeout);
              sized_write(fd,buf,size);
              alarm(0);
              break;
        case SIGPIPE:
              /* write() failed because recvr closed pipe */
              alarm(0);
              break;
        case SIGALRM:
              /* write() failed to complete fast enough */
              break:
        }
  }
```

The first time through `main`, the 0 case is executed. If the alarm is not disable by `alarm(0)`, the alarm signal handler is invoked passing control to the switch in such a way the the `SIGALRM` case is taken. (A very complete explanation of `setjmp` and `longjmp` can be found in [Libes 88].)

Socket interface

As implemented in 4.2BSD, access to TCP/IP is through the "socket" interface. The socket interface is quite general, and at the same time, complex for even the simplest things. A few simple routines in a high-level library would be wonderfully helpful.

Until that is done, our software includes a few calls to manage socket connections. By using these, an application need not ever use any of the primitive interprocess communication system calls. The following two calls replace nine system calls and several other network library calls.

To open a connection, the user makes the following call:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include "inet.h"

int socket = initport(port,role,sockettype,host_in)
int role;
int sockettype;
char *host_in;
```

`initport` returns a socket given a role, socket type and remote host name. Ports are defined by name or address using one of the following macros (here specifying SMTP which is handled on port 25):

```
PORT_NUMBER(25)
PORT_NAME("SMTP")
```

Programs are presumed to use connections in the client-server paradigm. Therefore, the application has a choice of either `SERVER` or `CLIENT` for the role. The socket type may be either of `SOCK_STREAM` (for TCP service) or `SOCK_DGRAM` for (for UDP service). `host_in` defines the computer where a client expects to find the service. The local computer may be selected by passing a zero character pointer or a null string. Servers are expected to do similarly, since they do not know in advance which computer may be requesting their service.

If `initport` returns with a non-negative socket, clients may immediately beginning communicating with the requested service. Sockets returned to servers are used to listen for service request by clients. In order to find out if a client is requesting a connection, the server calls:

```
fd_set readers;
FD_ZERO(&readers);

int client = select_server_stream(socket,readers);
```

Each time `select_server_stream` returns, it indicates the client that is requesting service. The actual data returned can be passed to `sized_read` to get the data itself.

`socket` is exactly the socket returned by `initport`. `readers` is a set of file descriptors used to remember the server's open connections.

Once connected, applications use the descriptor returned by `initport` as the file descriptor required in `sized_read` and `sized_write`.

Performance

Some versions of TCP try to make better use of network bandwidth by aggregating small data packets. For example, transmission of a single-byte packet will be delayed for a short time while the TCP service waits for more bytes to come. Unfortunately, it is not always possible to control this "feature". (Some implementations support disabling it completely through the use of an option called `TCP_NODELAY`. Ours did not.)

The reason for this optimization is that TCP is most often used as a service for interactive applications (such as `rlogin` or `telnet`). In such cases, humans are sitting at terminals typing at relatively slow speeds. Rarely do humans wait for characters to be sent individually. Rather, humans tend to type the letters of a word or an entire phrase in quick succession.

At the physical layer, the data portion of a single-byte TCP packet is overwhelmed by the amount of overhead information. Therefore, it is worthwhile for the computer to wait for more characters before sending even a few out. Most implementations we have seen range from 100 to 200ms while waiting for more bytes to send in a packet. This translates to 5 to 10 characters per second. Evidently, this is a rate that implementors feel most humans can maintain while typing a single phrase.

It is our observation that as the packet size goes up, the time spent waiting to aggregate more packets goes down. This holds true until whatever internal buffer size being used by TCP is reached. For example, 1024-byte buffers are extremely common. Thus, padding small packet buffers with 1K of junk can actually flush your data out of the system immediately. Of course, larger packet sizes can potentially degrade performance due to other network traffic.

Some experimentation may be necessary to determine if padding is necessary when communicating using small packets.

How to get the software - FTP

This software may be retrieved by ftp to the Internet site `durer.cme.nist.gov` using the account name: `anonymous`. The software is called `pub/sized_io.shar.Z` and is stored as a compressed shar file.

Comments or problems with the software may sent electronically to `libes@cme.nist.gov` or `cme-durer!uunet!libes` or mailed to:

Don Libes
National Institute of Standards and Technology
Bldg 220, Rm A-127
Gaithersburg, MD 20899

References

[Leffler 83] Leffler, S., Fabry, R., Joy, W., "4.2BSD Interprocess Communications Primer", Computer Systems Research Group, U.C. Berkeley, July, 1983.

[Libes 85] "User-Level Shared Variables", Usenix Conference Proceedings, Portland, Oregon, June 1985.

[Libes 88] "The C Forum: Win a Gold Medal in the Longjmp", Micro/Systems Journal, M&T Publishing, Vol. 4, No. 1, January 1988.

[Stallings 88] "Computer Communications Standards - The Open Systems Interconnection (OSI) Model and OSI Related Standards", William Stallings, Stallings/Macmillan, 1988.