

From: vic@brutus.tlug.org (Vic Metcalfe)  
Newsgroups: comp.unix.programmer,comp.unix.answers,comp.answers,news.answers  
Subject: [comp.unix.programmer] Unix-socket-faq for network programming  
Followup-To: comp.unix.programmer  
Summary: This posting offers answers to frequent questions about network programming in the unix environment using sockets.  
Approved: news-answers-request@MIT.EDU

Archive-name: unix-faq/socket  
Posting-Frequency: monthly  
Last-modified: 1998/05/21  
URL: <http://www.ibrado.com/sock-faq/>

Programming UNIX Sockets in C - Frequently Asked Questions  
Created by Vic Metcalfe, Andrew Gierth and other contributors  
May 21, 1998

This is a list of frequently asked questions, with answers about programming TCP/IP applications in unix with the sockets interface.

---

Table of Contents:

1. General Information and Concepts
  - 1.1. What's new?
  - 1.2. About this FAQ
  - 1.3. Who is this FAQ for?
  - 1.4. What are Sockets?
  - 1.5. How do Sockets Work?
  - 1.6. Where can I get source code for the book [book title]?
  - 1.7. Where can I get more information?
2. Questions regarding both Clients and Servers (TCP/SOCK\_STREAM)
  - 2.1. How can I tell when a socket is closed on the other end?

- 2.2. What's with the second parameter in bind()?
- 2.3. How do I get the port number for a given service?
- 2.4. If bind() fails, what should I do with the socket descriptor?
- 2.5. How do I properly close a socket?
- 2.6. When should I use shutdown()?
- 2.7. Please explain the TIME\_WAIT state.
- 2.8. Why does it take so long to detect that the peer died?
- 2.9. What are the pros/cons of select(), non-blocking I/O and SIGIO?
- 2.10. Why do I get EPROTO from read()?
- 2.11. How can I force a socket to send the data in its buffer?
- 2.12. Where can I get a library for programming sockets?
- 2.13. How come select says there is data, but read returns zero?
- 2.14. Whats the difference between select() and poll()?
- 2.15. How do I send [this] over a socket?
- 2.16. How do I use TCP\_NODELAY?
- 2.17. What exactly does the Nagle algorithm do?
- 2.18. What is the difference between read() and recv()?
- 2.19. I see that send()/write() can generate SIGPIPE. Is there any advantage to handling the signal, rather than just ignoring it and checking for the EPIPE error? Are there any useful parameters passed to the signal catching function?
- 2.20. After the chroot(), calls to socket() are failing. Why?
- 2.21. Why do I keep getting EINTR from the socket calls?

- 2.22. When will my application receive SIGPIPE?
- 2.23. What are socket exceptions? What is out-of-band data?
- 2.24. How can I find the full hostname (FQDN) of the system I'm running on?
- 3. Writing Client Applications (TCP/SOCK\_STREAM)
  - 3.1. How do I convert a string into an internet address?
  - 3.2. How can my client work through a firewall/proxy server?
  - 3.3. Why does connect() succeed even before my server did an accept()?
  - 3.4. Why do I sometimes lose a server's address when using more than one server?
  - 3.5. How can I set the timeout for the connect() system call?
  - 3.6. Should I bind() a port number in my client program, or let the system choose one for me on the connect() call?
  - 3.7. Why do I get "connection refused" when the server isn't running?
  - 3.8. What does one do when one does not know how much information is coming over the socket ? Is there a way to have a dynamic buffer ?
- 4. Writing Server Applications (TCP/SOCK\_STREAM)
  - 4.1. How come I get "address already in use" from bind()?
  - 4.2. Why don't my sockets close?
  - 4.3. How can I make my server a daemon?
  - 4.4. How can I listen on more than one port at a time?
  - 4.5. What exactly does SO\_REUSEADDR do?
  - 4.6. What exactly does SO\_LINGER do?

- 4.7. What exactly does `SO_KEEPALIVE` do?
- 4.8. How can I `bind()` to a port number  $< 1024$ ?
- 4.9. How do I get my server to find out the client's address / hostname?
- 4.10. How should I choose a port number for my server?
- 4.11. What is the difference between `SO_REUSEADDR` and `SO_REUSEPORT`?
- 4.12. How can I write a multi-homed server?
- 4.13. How can I read only one character at a time?
- 4.14. I'm trying to `exec()` a program from my server, and attach my socket's IO to it, but I'm not getting all the data across. Why?
5. Writing `UDP/SOCK_DGRAM` applications
  - 5.1. When should I use `UDP` instead of `TCP`?
  - 5.2. What is the difference between "connected" and "unconnected" sockets?
  - 5.3. Does doing a `connect()` call affect the receive behaviour of the socket?
  - 5.4. How can I read `ICMP` errors from "connected" `UDP` sockets?
  - 5.5. How can I be sure that a `UDP` message is received?
  - 5.6. How can I be sure that `UDP` messages are received in order?
  - 5.7. How often should I re-transmit un-acknowledged messages?
  - 5.8. How come only the first part of my datagram is getting through?
  - 5.9. Why does the socket's buffer fill up sooner than expected?
6. Advanced Socket Programming
  - 6.1. How would I put my socket in non-blocking mode?

6.2. How can I put a timeout on connect()?

7. Sample Source Code

---

1. General Information and Concepts

1.1. What's new?

Changed on May 21/1998

o 2.12. Where can a get a library for programming sockets? (fixed a couple of links)

o 7. Sample Source Code (put examples on a new ftp server)

1.2. About this FAQ

This FAQ is maintained by Vic Metcalfe ([vic@acm.org](mailto:vic@acm.org)), with lots of assistance from Andrew Gierth ([andrew@erlenstar.demon.co.uk](mailto:andrew@erlenstar.demon.co.uk)). I am depending on the true wizards to fill in the details, and correct my (no doubt) plentiful mistakes. The code examples in this FAQ are written to be easy to follow and understand. It is up to the reader to make them as efficient as required. I started this faq because after reading `comp.unix.programmer` for a short time, it became evident that a FAQ for sockets was needed.

The FAQ is available at the following locations:

Usenet: (Posted on the 21st of each month)

`news.answers`, `comp.answers`, `comp.unix.answers`,  
`comp.unix.programmer`

FTP:

`ftp://rtfm.mit.edu/pub/usenet/news.answers/unix-faq/socket`

WWW:

`http://www.ibrado.com/sock-faq`

`http://kipper.york.ac.uk/~vic/sock-faq` `http://www.ntua.gr/sock-faq`

Please email me if you would like to correct or clarify an answer. I would also like to hear from you if you would like me to add a question to the list. I may not be able to answer it, but I can add

it in the hopes that someone else will submit an answer. Every hour I seem to be getting even busier, so if I am slow to respond to your email, please be patient. If more than a week passes you may want to send me another one as I often put messages aside for later and then forget about them. I'll have to work on dealing with my mail better, but until then feel free to pester me a little bit.

### 1.3. Who is this FAQ for?

This FAQ is for C programmers in the Unix environment. It is not intended for WinSock programmers, or for Perl, Java, etc. I have nothing against Windows or Perl, but I had to limit the scope of the FAQ for the first draft. In the future, I would really like to provide examples for Perl, Java, and maybe others. For now though I will concentrate on correctness and completeness for C.

This version of the FAQ will only cover sockets of the AF\_INET family, since this is their most common use. Coverage of other types of sockets may be added later.

### 1.4. What are Sockets?

Sockets are just like "worm holes" in science fiction. When things go into one end, they (should) come out of the other. Different kinds of sockets have different properties. Sockets are either connection-oriented or connectionless. Connection-oriented sockets allow for data to flow back and forth as needed, while connectionless sockets (also known as datagram sockets) allow only one message at a time to be transmitted, without an open connection. There are also different socket families. The two most common are AF\_INET for internet connections, and AF\_UNIX for unix IPC (interprocess communication). As stated earlier, this FAQ deals only with AF\_INET sockets.

### 1.5. How do Sockets Work?

The implementation is left up to the vendor of your particular unix, but from the point of view of the programmer, connection-oriented sockets work a lot like files, or pipes. The most noticeable difference, once you have your file descriptor is that read() or write() calls may actually read or write fewer bytes than requested. If this happens, then you will have to make a second call for the rest of the data. There are examples of this in the source code that accompanies the faq.

## 1.6. Where can I get source code for the book [book title]?

Here is a list of the places I know to get source code for network programming books. It is very short, so please mail me with any others you know of.

Title: Unix Network Programming

Author: W. Richard Stevens (rstevens@noao.edu)

Publisher: Prentice Hall, Inc.

ISBN: 0-13-949876-1

URL: <http://www.noao.edu/~rstevens>

Title: Power Programming with RPC

Author: John Bloomer

Publisher: O'Reilly & Associates, Inc.

ISBN: 0-937175-77-3

URL: <ftp://ftp.uu.net/published/oreilly/nutshell/rpc/rpc.tar.Z>

Recommended by: Lokmann Merican (lokmanm#pop4.jaring.my@199.1.1.88)

Title: UNIX PROGRAM DEVELOPMENT for IBM PC'S Including OSF/Motif

Author: Thomas Yager

Publisher: Addison Wesley, 1991

ISBN: 0-201-57727-5

## 1.7. Where can I get more information?

I keep a copy of the resources I know of on my socks page on the web. I don't remember where I got most of these items, but some day I'll check out their sources, and provide ftp information here. For now, you can get them at <http://www.ibrado.com/sock-faq>.

There is a good TCP/IP FAQ maintained by George Neville-Neil (gnn@wrs.com) which can be found at <http://www.visi.com/~khayes/tcpipfaq.html>

## 2. Questions regarding both Clients and Servers (TCP/SOCK\_STREAM)

### 2.1. How can I tell when a socket is closed on the other end?

From Andrew Gierth (andrew@erlenstar.demon.co.uk):

AFAIK:

If the peer calls close() or exits, without having messed with

SO\_LINGER, then our calls to read() should return 0. It is less clear what happens to write() calls in this case; I would expect EPIPE, not on the next call, but the one after.

If the peer reboots, or sets l\_onoff = 1, l\_linger = 0 and then closes, then we should get ECONNRESET (eventually) from read(), or EPIPE from write().

I should also point out that when write() returns EPIPE, it also raises the SIGPIPE signal - you never see the EPIPE error unless you handle or ignore the signal.

If the peer remains unreachable, we should get some other error.

I don't think that write() can legitimately return 0. read() should return 0 on receipt of a FIN from the peer, and on all following calls.

So yes, you must expect read() to return 0.

As an example, suppose you are receiving a file down a TCP link; you might handle the return from read() like this:

```
rc = read(sock,buf,sizeof(buf));
if (rc > 0)
{
    write(file,buf,rc);
    /* error checking on file omitted */
}
else if (rc == 0)
{
    close(file);
    close(sock);
    /* file received successfully */
}
else /* rc < 0 */
{
    /* close file and delete it, since data is not complete
       report error, or whatever */
}
```

2.2. What's with the second parameter in bind()?

The man page shows it as "struct sockaddr \*my\_addr". The sockaddr

struct though is just a place holder for the structure it really wants. You have to pass different structures depending on what kind of socket you have. For an AF\_INET socket, you need the sockaddr\_in structure. It has three fields of interest:

sin\_family

Set this to AF\_INET.

sin\_port

The network byte-ordered 16 bit port number

sin\_addr

The host's ip number. This is a struct in\_addr, which contains only one field, s\_addr which is a u\_long.

### 2.3. How do I get the port number for a given service?

Use the getservbyname() routine. This will return a pointer to a servent structure. You are interested in the s\_port field, which contains the port number, with correct byte ordering (so you don't need to call htons() on it). Here is a sample routine:

```
/* Take a service name, and a service type, and return a port number. If the
   service name is not found, it tries it as a decimal number. The number
   returned is byte ordered for the network. */
int atoport(char *service, char *proto) {
    int port;
    long int lport;
    struct servent *serv;
    char *errpos;

    /* First try to read it from /etc/services */
    serv = getservbyname(service, proto);
    if (serv != NULL)
        port = serv->s_port;
    else { /* Not in services, maybe a number? */
        lport = strtol(service,&errpos,0);
        if ( ( errpos[0] != 0 ) || (lport < 1) || (lport > 5000) )
            return -1; /* Invalid port address */
        port = htons(lport);
    }
    return port;
}
```

#### 2.4. If bind() fails, what should I do with the socket descriptor?

If you are exiting, I have been assured by Andrew that all unices will close open file descriptors on exit. If you are not exiting though, you can just close it with a regular close() call.

#### 2.5. How do I properly close a socket?

This question is usually asked by people who try close(), because they have seen that that is what they are supposed to do, and then run netstat and see that their socket is still active. Yes, close() is the correct method. To read about the TIME\_WAIT state, and why it is important, refer to ``2.7 Please explain the TIME\_WAIT state.".

#### 2.6. When should I use shutdown()?

From Michael Hunter (mphunter@qnx.com):

shutdown() is useful for deliniating when you are done providing a request to a server using TCP. A typical use is to send a request to a server followed by a shutdown(). The server will read your request followed by an EOF (read of 0 on most unix implementations). This tells the server that it has your full request. You then go read blocked on the socket. The server will process your request and send the necessary data back to you followed by a close. When you have finished reading all of the response to your request you will read an EOF thus signifying that you have the whole response. It should be noted the TTCP (TCP for Transactions -- see R. Steven's home page) provides for a better method of tcp transaction management.

S.Degtyarev (deg@sunsr.inp.nsk.su) wrote a nice in-depth message to me about this. He shows a practical example of using shutdown() to aid in synchronization of client processes when one is the "reader" process, and the other is the "writer" process. A portion of his message follows:

Sockets are very similar to pipes in the way they are used for data transfer and client/server transactions, but not like pipes they are bidirectional. Programs that use sockets often fork() and each process inherits the socket descriptor. In pipe based programs it is strictly recommended to close all the pipe ends that are not used to convert the pipe line to one-directional data stream to avoid data losses and deadlocks. With the socket there is no way to allow one process only to send data and the other only to receive so you should

always keep in mind the consequences.

Generally the difference between `close()` and `shutdown()` is: `close()` closes the socket id for the process but the connection is still opened if another process shares this socket id. The connection stays opened both for read and write, and sometimes this is very important. `shutdown()` breaks the connection for all processes sharing the socket id. Those who try to read will detect EOF, and those who try to write will receive SIGPIPE, possibly delayed while the kernel socket buffer will be filled. Additionally, `shutdown()` has a second argument which denotes how to close the connection: 0 means to disable further reading, 1 to disable writing and 2 disables both.

The quick example below is a fragment of a very simple client process. After establishing the connection with the server it forks. Then child sends the keyboard input to the server until EOF is received and the parent receives answers from the server.

```
/*
 *   Sample client fragment,
 *   variables declarations and error handling are omitted
 */
s=connect(...);

if( fork() ){ /*   The child, it copies its stdin to
                the socket */
    while( gets(buffer) >0)
        write(s,buf,strlen(buffer));

    close(s);
    exit(0);
}

else { /* The parent, it receives answers */
    while( (l=read(s,buffer,sizeof(buffer))){
        do_something(l,buffer);

        /* Connection break from the server is assumed */
        /* ATTENTION: deadlock here */
        wait(0); /* Wait for the child to exit */
        exit(0);
    }
}
```

What do we expect? The child detects an EOF from its stdin, it closes

the socket (assuming connection break) and exits. The server in its turn detects EOF, closes connection and exits. The parent detects EOF, makes the wait() system call and exits. What do we see instead? The socket instance in the parent process is still opened for writing and reading, though the parent never writes. The server never detects EOF and waits for more data from the client forever. The parent never sees the connection is closed and hangs forever and the server hangs too. Unexpected deadlock! ( any deadlock is unexpected though :-)

You should change the client fragment as follows:

```
if( fork() ) { /* The child */
    while( gets(buffer) )
        write(s,buffer,strlen(buffer));

        shutdown(s,1); /* Break the connection
for writing, The server will detect EOF now. Note: reading from
the socket is still allowed. The server may send some more data
after receiving EOF, why not? */
    exit(0);
}
```

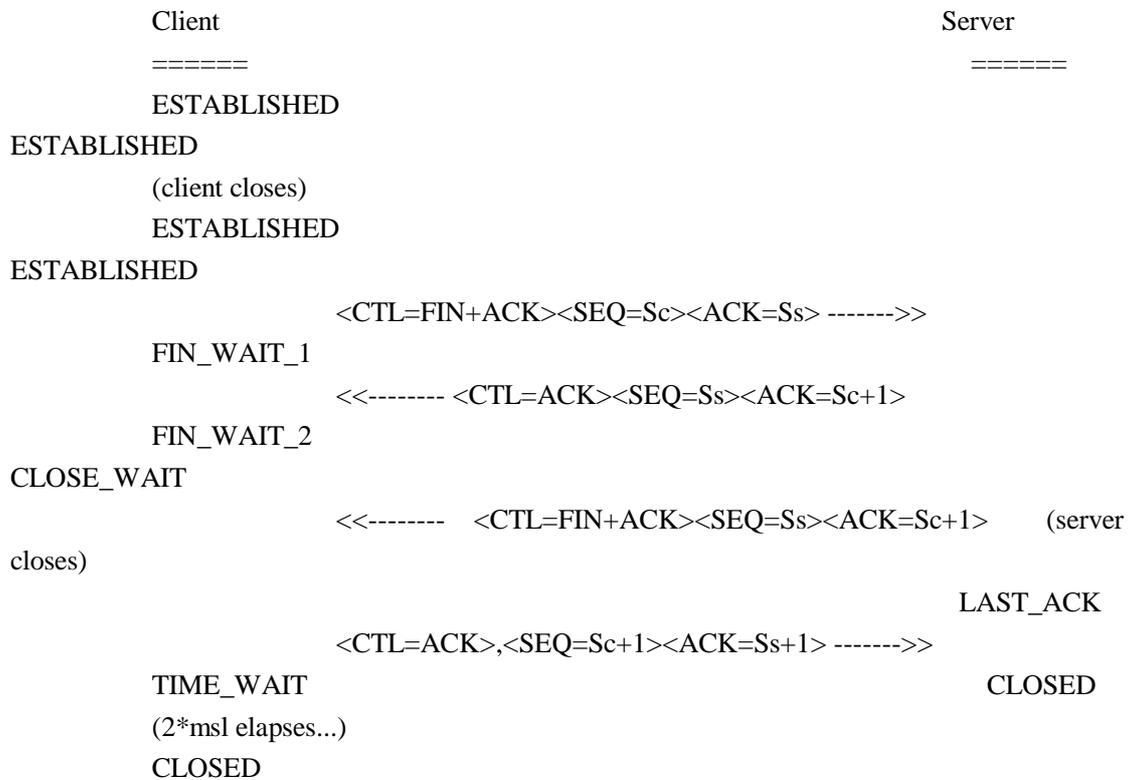
I hope this rough example explains the troubles you can have with client/server synchronization. Generally you should always remember all the instances of the particular socket in all the processes that share the socket and close them all at once if you wish to use close() or use shutdown() in one process to break the connection.

2.7. Please explain the TIME\_WAIT state.

Remember that TCP guarantees all data transmitted will be delivered, if at all possible. When you close a socket, the server goes into a TIME\_WAIT state, just to be really really sure that all the data has gone through. When a socket is closed, both sides agree by sending messages to each other that they will send no more data. This, it seemed to me was good enough, and after the handshaking is done, the socket should be closed. The problem is two-fold. First, there is no way to be sure that the last ack was communicated successfully. Second, there may be "wandering duplicates" left on the net that must be dealt with if they are delivered.

Andrew Gierth (andrew@erlenstar.demon.co.uk) helped to explain the closing sequence in the following usenet posting:

Assume that a connection is in ESTABLISHED state, and the client is about to do an orderly release. The client's sequence no. is  $S_c$ , and the server's is  $S_s$ . The pipe is empty in both directions.



Note: the +1 on the sequence numbers is because the FIN counts as one byte of data. (The above diagram is equivalent to fig. 13 from RFC 793).

Now consider what happens if the last of those packets is dropped in the network. The client has done with the connection; it has no more data or control info to send, and never will have. But the server does not know whether the client received all the data correctly; that's what the last ACK segment is for. Now the server may or may not care whether the client got the data, but that is not an issue for TCP; TCP is a reliable rotocol, and must distinguish between an orderly connection close where all data is transferred, and a connection abort where data may or may not have been lost.

So, if that last packet is dropped, the server will retransmit it (it is, after all, an unacknowledged segment) and will expect to see a suitable ACK segment in reply. If the client went straight to CLOSED, the only possible response to that retransmit would be a RST, which would indicate to the server that data had been lost, when in fact it had not been.

(Bear in mind that the server's FIN segment may, additionally, contain data.)

DISCLAIMER: This is my interpretation of the RFCs (I have read all the TCP-related ones I could find), but I have not attempted to examine implementation source code or trace actual connections in order to verify it. I am satisfied that the logic is correct, though.

More commentary from Vic:

The second issue was addressed by Richard Stevens (rstevens@noao.edu, author of "Unix Network Programming", see ``1.6 Where can I get source code for the book [book title]?"). I have put together quotes from some of his postings and email which explain this. I have brought together paragraphs from different postings, and have made as few changes as possible.

From Richard Stevens (rstevens@noao.edu):

If the duration of the TIME\_WAIT state were just to handle TCP's full-duplex close, then the time would be much smaller, and it would be some function of the current RTO (retransmission timeout), not the MSL (the packet lifetime).

A couple of points about the TIME\_WAIT state.

- o The end that sends the first FIN goes into the TIME\_WAIT state, because that is the end that sends the final ACK. If the other end's FIN is lost, or if the final ACK is lost, having the end that sends the first FIN maintain state about the connection guarantees that it has enough information to retransmit the final ACK.
  
- o Realize that TCP sequence numbers wrap around after  $2^{32}$  bytes have been transferred. Assume a connection between A.1500 (host A, port 1500) and B.2000. During the connection one segment is lost and retransmitted. But the segment is not really lost, it is held by some intermediate router and then re-injected into the network. (This is called a "wandering duplicate".) But in the time between the packet being lost & retransmitted, and then reappearing, the connection is closed (without any problems) and then another connection is established between the same host, same port (that is, A.1500 and B.2000; this is called another "incarnation" of the connection). But the sequence numbers chosen for the new

incarnation just happen to overlap with the sequence number of the wandering duplicate that is about to reappear. (This is indeed possible, given the way sequence numbers are chosen for TCP connections.) Bingo, you are about to deliver the data from the wandering duplicate (the previous incarnation of the connection) to the new incarnation of the connection. To avoid this, you do not allow the same incarnation of the connection to be reestablished until the `TIME_WAIT` state terminates.

Even the `TIME_WAIT` state doesn't completely solve the second problem, given what is called `TIME_WAIT` assassination. RFC 1337 has more details.

- o The reason that the duration of the `TIME_WAIT` state is  $2 * \text{MSL}$  is that the maximum amount of time a packet can wander around a network is assumed to be `MSL` seconds. The factor of 2 is for the round-trip. The recommended value for `MSL` is 120 seconds, but Berkeley-derived implementations normally use 30 seconds instead. This means a `TIME_WAIT` delay between 1 and 4 minutes. Solaris 2.x does indeed use the recommended `MSL` of 120 seconds.

A wandering duplicate is a packet that appeared to be lost and was retransmitted. But it wasn't really lost ... some router had problems, held on to the packet for a while (order of seconds, could be a minute if the TTL is large enough) and then re-injects the packet back into the network. But by the time it reappears, the application that sent it originally has already retransmitted the data contained in that packet.

Because of these potential problems with `TIME_WAIT` assassinations, one should not avoid the `TIME_WAIT` state by setting the `SO_LINGER` option to send an RST instead of the normal TCP connection termination (`FIN/ACK/FIN/ACK`). The `TIME_WAIT` state is there for a reason; it's your friend and it's there to help you :-)

I have a long discussion of just this topic in my just-released "TCP/IP Illustrated, Volume 3". The `TIME_WAIT` state is indeed, one of the most misunderstood features of TCP.

I'm currently rewriting "Unix Network Programming" (see ``1.6 Where can I get source code for the book [book title]?"). and will include lots more on this topic, as it is often confusing and misunderstood.

An additional note from Andrew:

Closing a socket: if `SO_LINGER` has not been called on a socket, then `close()` is not supposed to discard data. This is true on SVR4.2 (and, apparently, on all non-SVR4 systems) but apparently not on SVR4; the use of either `shutdown()` or `SO_LINGER` seems to be required to guarantee delivery of all data.

## 2.8. Why does it take so long to detect that the peer died?

From Andrew Gierth ([andrew@erlenstar.demon.co.uk](mailto:andrew@erlenstar.demon.co.uk)):

Because by default, no packets are sent on the TCP connection unless there is data to send or acknowledge.

So, if you are simply waiting for data from the peer, there is no way to tell if the peer has silently gone away, or just isn't ready to send any more data yet. This can be a problem (especially if the peer is a PC, and the user just hits the Big Switch...).

One solution is to use the `SO_KEEPALIVE` option. This option enables periodic probing of the connection to ensure that the peer is still present. BE WARNED: the default timeout for this option is AT LEAST 2 HOURS. This timeout can often be altered (in a system-dependent fashion) but not normally on a per-connection basis (AFAIK).

RFC1122 specifies that this timeout (if it exists) must be configurable. On the majority of Unix variants, this configuration may only be done globally, affecting all TCP connections which have keepalive enabled. The method of changing the value, moreover, is often difficult and/or poorly documented, and in any case is different for just about every version in existence.

If you must change the value, look for something resembling `tcp_keepidle` in your kernel configuration or network options configuration.

If you're sending to the peer, though, you have some better guarantees; since sending data implies receiving ACKs from the peer, then you will know after the retransmit timeout whether the peer is still alive. But the retransmit timeout is designed to allow for various contingencies, with the intention that TCP connections are not dropped simply as a result of minor network upsets. So you should still expect a delay of several minutes before getting notification of the failure.

The approach taken by most application protocols currently in use on the Internet (e.g. FTP, SMTP etc.) is to implement read timeouts on the server end; the server simply gives up on the client if no requests are received in a given time period (often of the order of 15 minutes). Protocols where the connection is maintained even if idle for long periods have two choices:

1. use `SO_KEEPALIVE`
2. use a higher-level keepalive mechanism (such as sending a null request to the server every so often).

#### 2.9. What are the pros/cons of `select()`, non-blocking I/O and `SIGIO`?

Using non-blocking I/O means that you have to poll sockets to see if there is data to be read from them. Polling should usually be avoided since it uses more CPU time than other techniques.

Using `SIGIO` allows your application to do what it does and have the operating system tell it (with a signal) that there is data waiting for it on a socket. The only drawback to this solution is that it can be confusing, and if you are dealing with multiple sockets you will have to do a `select()` anyway to find out which one(s) is ready to be read.

Using `select()` is great if your application has to accept data from more than one socket at a time since it will block until any one of a number of sockets is ready with data. One other advantage to `select()` is that you can set a time-out value after which control will be returned to you whether any of the sockets have data for you or not.

#### 2.10. Why do I get `EPROTO` from `read()`?

From Steve Rago (sar@plc.com):

`EPROTO` means that the protocol encountered an unrecoverable error for that endpoint. `EPROTO` is one of those catch-all error codes used by `STREAMS`-based drivers when a better code isn't available.

And an addition note from Andrew (andrew@erlenstar.demon.co.uk):

Not quite to do with `EPROTO` from `read()`, but I found out once that on some `STREAMS`-based implementations, `EPROTO` could be returned by

accept() if the incoming connection was reset before the accept completes.

On some other implementations, accept seemed to be capable of blocking if this occurred. This is important, since if select() said the listening socket was readable, then you would normally expect not to block in the accept() call. The fix is, of course, to set nonblocking mode on the listening socket if you are going to use select() on it.

2.11. How can I force a socket to send the data in its buffer?

From Richard Stevens (rstevens@noao.edu):

You can't force it. Period. TCP makes up its own mind as to when it can send data. Now, normally when you call write() on a TCP socket, TCP will indeed send a segment, but there's no guarantee and no way to force this. There are lots of reasons why TCP will not send a segment: a closed window and the Nagle algorithm are two things to come immediately to mind.

(Snipped suggestion from Andrew Gierth to use TCP\_NODELAY)

Setting this only disables one of the many tests, the Nagle algorithm. But if the original poster's problem is this, then setting this socket option will help.

A quick glance at tcp\_output() shows around 11 tests TCP has to make as to whether to send a segment or not.

Now from Dr. Charles E. Campbell Jr. (cec@gryphon.gsfc.nasa.gov):

As you've surmised, I've never had any problem with disabling Nagle's algorithm. Its basically a buffering method; there's a fixed overhead for all packets, no matter how small. Hence, Nagle's algorithm collects small packets together (no more than .2sec delay) and thereby reduces the amount of overhead bytes being transferred. This approach works well for rcp, for example: the .2 second delay isn't humanly noticeable, and multiple users have their small packets more efficiently transferred. Helps in university settings where most folks using the network are using standard tools such as rcp and ftp, and programs such as telnet may use it, too.

However, Nagle's algorithm is pure havoc for real-time control and not much better for keystroke interactive applications (control-C,

anyone?). It has seemed to me that the types of new programs using sockets that people write usually do have problems with small packet delays. One way to bypass Nagle's algorithm selectively is to use "out-of-band" messaging, but that is limited in its content and has other effects (such as a loss of sequentiality) (by the way, out-of-band is often used for that ctrl-C, too).

More from Vic:

So to sum it all up, if you are having trouble and need to flush the socket, setting the TCP\_NODELAY option will usually solve the problem. If it doesn't, you will have to use out-of-band messaging, but according to Andrew, "out-of-band data has its own problems, and I don't think it works well as a solution to buffering delays (haven't tried it though). It is not 'expedited data' in the sense that exists in some other protocols; it is transmitted in-stream, but with a pointer to indicate where it is."

I asked Andrew something to the effect of "What promises does TCP make about when it will get around to writing data to the network?" I thought his reply should be put under this question:  
Not many promises, but some.

I'll try and quote chapter and verse on this:

References:

RFC 1122, "Requirements for Internet Hosts" (also STD 3)  
RFC 793, "Transmission Control Protocol" (also STD 7)

1. The socket interface does not provide access to the TCP PUSH flag.

2. RFC1122 says (4.2.2.2):

A TCP MAY implement PUSH flags on SEND calls. If PUSH flags are not implemented, then the sending TCP: (1) must not buffer data indefinitely, and (2) MUST set the PSH bit in the last buffered segment (i.e., when there is no more queued data to be sent).

3. RFC793 says (2.8):

When a receiving TCP sees the PUSH flag, it must not wait for more data from the sending TCP before passing the data to the receiving process.

[RFC1122 supports this statement.]

4. Therefore, data passed to a write() call must be delivered to the peer within a finite time, unless prevented by protocol considerations.
5. There are (according to a post from Stevens quoted in the FAQ [earlier in this answer - Vic]) about 11 tests made which could delay sending the data. But as I see it, there are only 2 that are significant, since things like retransmit backoff are a) not under the programmers control and b) must either resolve within a finite time or drop the connection.

The first of the interesting cases is "window closed" (ie. there is no buffer space at the receiver; this can delay data indefinitely, but only if the receiving process is not actually reading the data that is available)

Vic asks:

OK, it makes sense that if the client isn't reading, the data isn't going to make it across the connection. I take it this causes the sender to block after the receive queue is filled?

The sender blocks when the socket send buffer is full, so buffers will be full at both ends.

While the window is closed, the sending TCP sends window probe packets. This ensures that when the window finally does open again, the sending TCP detects the fact. [RFC1122, ss 4.2.2.17]

The second interesting case is "Nagle algorithm" (small segments, e.g. keystrokes, are delayed to form larger segments if ACKs are expected from the peer; this is what is disabled with TCP\_NODELAY)

Vic Asks:

Does this mean that my tcpclient sample should set TCP\_NODELAY to ensure that the end-of-line code is indeed put out onto the network when sent?

No. tcpclient.c is doing the right thing as it stands; trying to write as much data as possible in as few calls to write() as is feasible.

Since the amount of data is likely to be small relative to the socket send buffer, then it is likely (since the connection is idle at that point) that the entire request will require only one call to write(), and that the TCP layer will immediately dispatch the request as a single segment (with the PSH flag, see point 2.2 above).

The Nagle algorithm only has an effect when a second write() call is made while data is still unacknowledged. In the normal case, this data will be left buffered until either: a) there is no unacknowledged data; or b) enough data is available to dispatch a full-sized segment. The delay cannot be indefinite, since condition (a) must become true within the retransmit timeout or the connection dies.

Since this delay has negative consequences for certain applications, generally those where a stream of small requests are being sent without response, e.g. mouse movements, the standards specify that an option must exist to disable it. [RFC1122, ss 4.2.3.4]

Additional note: RFC1122 also says:

[DISCUSSION]:

When the PUSH flag is not implemented on SEND calls, i.e., when the application/TCP interface uses a pure streaming model, responsibility for aggregating any tiny data fragments to form reasonable sized segments is partially borne by the application layer.

So programs should avoid calls to write() with small data lengths (small relative to the MSS, that is); it's better to build up a request in a buffer and then do one call to sock\_write() or equivalent.

The other possible sources of delay in the TCP are not really controllable by the program, but they can only delay the data temporarily.

Vic asks:

By temporarily, you mean that the data will go as soon as it can, and I won't get stuck in a position where one side is waiting on a response, and the other side hasn't received the request? (Or at least I won't get stuck forever)

You can only deadlock if you somehow manage to fill up all the buffers

in both directions... not easy.

If it is possible to do this, (can't think of a good example though), the solution is to use nonblocking mode, especially for writes. Then you can buffer excess data in the program as necessary.

#### 2.12. Where can I get a library for programming sockets?

There is the Simple Sockets Library by Charles E. Campbell, Jr. PhD. and Terry McRoberts. The file is called `ssl.tar.gz`, and you can download it from this faq's home page. For c++ there is the Socket++ library which is on `ftp://ftp.virginia.edu/pub/socket++-1.10.tar.gz`. There is also C++ Wrappers. The file is called `ftp://ftp.huji.ac.il/pub/languages/C++/C++_wrappers.tar.gz`. Thanks to Bill McKinnon for tracking it down for me! From `http://www.cs.wustl.edu/~schmidt` you should be able to find the ACE toolkit. PING Software Group has some libraries that include a sockets interface among other things. My link to their web site has gone stale, and I don't know where their new site is. Please send me an email if you find it.

Philippe Jounin <[ph.jounin@computer.org](mailto:ph.jounin@computer.org)> has developed a cross platform library which includes high level support for http and ftp protocols, with more to come. You can find it at `http://perso.magic.fr/jounin-ph/P_tcp4u.htm`, and you can find a review of it at `http://www6.zdnet.com/cgi-bin/texis/swlib/hotfiles/info.html?fcode=000H4F`

I don't have any experience with any of these libraries, so I can't recommend one over the other.

#### 2.13. How come select says there is data, but read returns zero?

The data that causes select to return is the EOF because the other side has closed the connection. This causes read to return zero. For more information see ``2.1 How can I tell when a socket is closed on the other end?''

#### 2.14. Whats the difference between select() and poll()?

From Richard Stevens ([rstevens@noao.edu](mailto:rstevens@noao.edu)):

The basic difference is that select()'s `fd_set` is a bit mask and therefore has some fixed size. It would be possible for the kernel to

not limit this size when the kernel is compiled, allowing the application to define `FD_SETSIZE` to whatever it wants (as the comments in the system header imply today) but it takes more work. 4.4BSD's kernel and the Solaris library function both have this limit. But I see that BSD/OS 2.1 has now been coded to avoid this limit, so it's doable, just a small matter of programming. :-) Someone should file a Solaris bug report on this, and see if it ever gets fixed.

With `poll()`, however, the user must allocate an array of `pollfd` structures, and pass the number of entries in this array, so there's no fundamental limit. As Casper notes, fewer systems have `poll()` than `select`, so the latter is more portable. Also, with original implementations (SVR3) you could not set the descriptor to -1 to tell the kernel to ignore an entry in the `pollfd` structure, which made it hard to remove entries from the array; SVR4 gets around this. Personally, I always use `select()` and rarely `poll()`, because I port my code to BSD environments too. Someone could write an implementation of `poll()` that uses `select()`, for these environments, but I've never seen one. Both `select()` and `poll()` are being standardized by POSIX 1003.1g.

#### 2.15. How do I send [this] over a socket?

Anything other than single bytes of data will probably get mangled unless you take care. For integer values you can use `htons()` and friends, and strings are really just a bunch of single bytes, so those should be OK. Be careful not to send a pointer to a string though, since the pointer will be meaningless on another machine. If you need to send a struct, you should write `sendthisstruct()` and `readthisstruct()` functions for it that do all the work of taking the structure apart on one side, and putting it back together on the other. If you need to send floats, you may have a lot of work ahead of you. You should read RFC 1014 which is about portable ways of getting data from one machine to another (thanks to Andrew Gabriel for pointing this out).

#### 2.16. How do I use TCP\_NODELAY?

First off, be sure you really want to use it in the first place. It will disable the Nagle algorithm (see ``2.11 How can I force a socket to send the data in its buffer?"), which will cause network traffic to increase, with smaller than needed packets wasting bandwidth. Also, from what I have been able to tell, the speed increase is very small, so you should probably do it without `TCP_NODELAY` first, and

only turn it on if there is a problem.

Here is a code example, with a warning about using it from Andrew Gierth:

```
int flag = 1;
int result = setsockopt(sock,          /* socket affected */
                       IPPROTO_TCP,   /* set option at TCP level */
                       TCP_NODELAY,   /* name of option */
                       (char *) &flag, /* the cast is historical
                                       cruft */
                       sizeof(int));   /* length of option value */

if (result < 0)
    ... handle the error ...
```

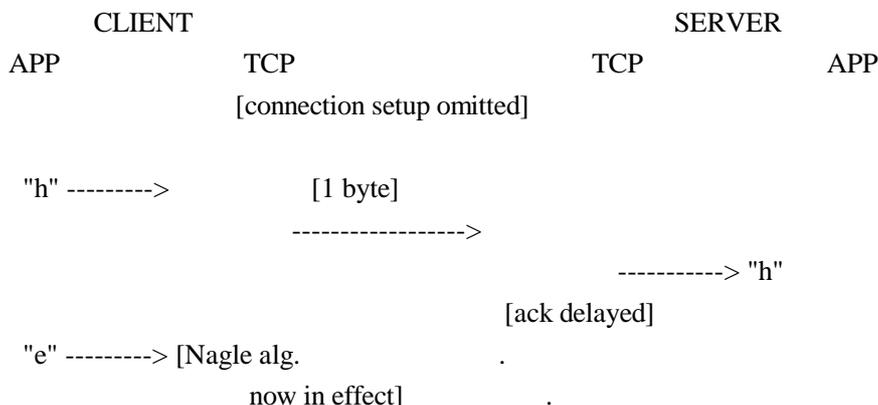
TCP\_NODELAY is for a specific purpose; to disable the Nagle buffering algorithm. It should only be set for applications that send frequent small bursts of information without getting an immediate response, where timely delivery of data is required (the canonical example is mouse movements).

## 2.17. What exactly does the Nagle algorithm do?

It groups together as much data as it can between ACK's from the other end of the connection. I found this really confusing until Andrew Gierth (andrew@erlenstar.demon.co.uk) drew the following diagram, and explained:

This diagram is not intended to be complete, just to illustrate the point better...

Case 1: client writes 1 byte per write() call. The program on host B is tcpserver.c from the FAQ examples.





Total segments: 3.

Time for response = RTT (therefore minimum possible).

Hope this makes things a bit clearer...

Note that in case 2, you don't want the implementation to gratuitously delay sending the data, since that would add straight onto the response time.

2.18. What is the difference between read() and recv()?

From Andrew Gierth (andrew@erlenstar.demon.co.uk):

read() is equivalent to recv() with a flags parameter of 0. Other values for the flags parameter change the behaviour of recv(). Similarly, write() is equivalent to send() with flags == 0.

It is unlikely that send()/recv() would be dropped; perhaps someone with a copy of the POSIX drafts for socket calls can check...

Portability note: non-unix systems may not allow read()/write() on sockets, but recv()/send() are usually ok. This is true on Windows and OS/2, for example.

2.19. I see that send()/write() can generate SIGPIPE. Is there any advantage to handling the signal, rather than just ignoring it and checking for the EPIPE error? Are there any useful parameters passed to the signal catching function?

From Andrew Gierth (andrew@erlenstar.demon.co.uk):

In general, the only parameter passed to a signal handler is the signal number that caused it to be invoked. Some systems have optional additional parameters, but they are no use to you in this case.

My advice is to just ignore SIGPIPE as you suggest. That's what I do in just about all of my socket code; errno values are easier to handle than signals (in fact, the first revision of the FAQ failed to mention SIGPIPE in that context; I'd got so used to ignoring it...)

There is one situation where you should not ignore SIGPIPE; if you are

going to `exec()` another program with `stdout` redirected to a socket. In this case it is probably wise to set `SIGPIPE` to `SIG_DFL` before doing the `exec()`.

2.20. After the `chroot()`, calls to `socket()` are failing. Why?

From Andrew Gierth ([andrew@erlenstar.demon.co.uk](mailto:andrew@erlenstar.demon.co.uk)):

On systems where sockets are implemented on top of Streams (e.g. all SysV-based systems, presumably including Solaris), the `socket()` function will actually be opening certain special files in `/dev`. You will need to create a `/dev` directory under your fake root and populate it with the required device nodes (only).

Your system documentation may or may not specify exactly which device nodes are required; I can't help you there (sorry). (Editors note: Adrian Hall ([adrian@hottub.org](mailto:adrian@hottub.org)) suggested checking the man page for `ftpd`, which should list the files you need to copy and devices you need to create in the `chroot'd` environment.)

A less-obvious issue with `chroot()` is if you call `syslog()`, as many daemons do; `syslog()` opens (depending on the system) either a UDP socket, a FIFO or a Unix-domain socket. So if you use it after a `chroot()` call, make sure that you call `openlog()` *before* the `chroot`.

2.21. Why do I keep getting `EINTR` from the socket calls?

This isn't really so much an error as an exit condition. It means that the call was interrupted by a signal. Any call that might block should be wrapped in a loop that checks for `EINTR`, as is done in the example code (See ``7. Sample Source Code").

2.22. When will my application receive `SIGPIPE`?

From Richard Stevens ([rstevens@noao.edu](mailto:rstevens@noao.edu)):

Very simple: with TCP you get `SIGPIPE` if your end of the connection has received an `RST` from the other end. What this also means is that if you were using `select` instead of `write`, the `select` would have indicated the socket as being readable, since the `RST` is there for you to read (read will return an error with `errno` set to `ECONNRESET`).

Basically an `RST` is TCP's response to some packet that it doesn't expect and has no other way of dealing with. A common case is when

the peer closes the connection (sending you a FIN) but you ignore it because you're writing and not reading. (You should be using select.) So you write to a connection that has been closed by the other end and the other end's TCP responds with an RST.

2.23. What are socket exceptions? What is out-of-band data?

Unlike exceptions in C++, socket exceptions do not indicate that an error has occurred. Socket exceptions usually refer to the notification that out-of-band data has arrived. Out-of-band data (called "urgent data" in TCP) looks to the application like a separate stream of data from the main data stream. This can be useful for separating two different kinds of data. Note that just because it is called "urgent data" does not mean that it will be delivered any faster, or with higher priority than data in the in-band data stream. Also beware that unlike the main data stream, the out-of-band data may be lost if your application can't keep up with it.

2.24. running on? How can I find the full hostname (FQDN) of the system I'm

From Richard Stevens (rstevens@noao.edu):

Some systems set the hostname to the FQDN and others set it to just the unqualified host name. I know the current BIND FAQ recommends the FQDN, but most Solaris systems, for example, tend to use only the unqualified host name.

Regardless, the way around this is to first get the host's name (perhaps an FQDN, perhaps unqualified). Most systems support the Posix way to do this using `uname()`, but older BSD systems only provide `gethostname()`. Call `gethostbyname()` to find your IP address. Then take the IP address and call `gethostbyaddr()`. The `h_name` member of the `hostent{ }` should then be your FQDN.

3. Writing Client Applications (TCP/SOCK\_STREAM)

3.1. How do I convert a string into an internet address?

If you are reading a host's address from the command line, you may not know if you have an `aaa.bbb.ccc.ddd` style address, or a `host.domain.com` style address. What I do with these, is first try to use it as a `aaa.bbb.ccc.ddd` type address, and if that fails, then do a name lookup on it. Here is an example:

```

/* Converts ascii text to in_addr struct.  NULL is returned if the
   address can not be found. */
struct in_addr *atoaddr(char *address) {
    struct hostent *host;
    static struct in_addr saddr;

    /* First try it as aaa.bbb.ccc.ddd. */
    saddr.s_addr = inet_addr(address);
    if (saddr.s_addr != -1) {
        return &saddr;
    }
    host = gethostbyname(address);
    if (host != NULL) {
        return (struct in_addr *) *host->h_addr_list;
    }
    return NULL;
}

```

### 3.2. How can my client work through a firewall/proxy server?

If you are running through separate proxies for each service, you shouldn't need to do anything. If you are working through sockd, you will need to "socksify" your application. Details for doing this can be found in the package itself, which is available at:

<ftp://ftp.net.com/socks.cstc/socks.cstc.4.2.tar.gz>

you can get the socks faq at:

<ftp://coast.cs.purdue.edu/pub/tools/unix/socks/FAQ>

### 3.3. Why does connect() succeed even before my server did an accept()?

From Andrew Gierth ([andrew@erlenstar.demon.co.uk](mailto:andrew@erlenstar.demon.co.uk)):

Once you have done a listen() call on your socket, the kernel is primed to accept connections on it. The usual UNIX implementation of this works by immediately completing the SYN handshake for any incoming valid SYN segments (connection attempts), creating the socket for the new connection, and keeping this new socket on an internal queue ready for the accept() call. So the socket is fully open before the accept is done.

The other factor in this is the 'backlog' parameter for `listen()`; that defines how many of these completed connections can be queued at one time. If the specified number is exceeded, then new incoming connects are simply ignored (which causes them to be retried).

3.4. Why do I sometimes lose a server's address when using more than one server?

From Andrew Gierth ([andrew@erlenstar.demon.co.uk](mailto:andrew@erlenstar.demon.co.uk)):

Take a careful look at `struct hostent`. Notice that almost everything in it is a pointer? All these pointers will refer to statically allocated data.

For example, if you do:

```
struct hostent *host = gethostbyname(hostname);
```

then (as you should know) a subsequent call to `gethostbyname()` will overwrite the structure pointed to by 'host'.

But if you do:

```
struct hostent myhost;
struct hostent *hostptr = gethostbyname(hostname);
if (hostptr) myhost = *host;
```

to make a copy of the `hostent` before it gets overwritten, then it still gets clobbered by a subsequent call to `gethostbyname()`, since although `myhost` won't get overwritten, all the data it is pointing to will be.

You can get round this by doing a proper 'deep copy' of the `hostent` structure, but this is tedious. My recommendation would be to extract the needed fields of the `hostent` and store them in your own way.

Robin Paterson ([etmrpat@etm.ericsson.se](mailto:etmrpat@etm.ericsson.se)) has added:

It might be nice if you mention MT safe libraries provide complimentary functions for multithreaded programming. On the solaris machine I'm typing at, we have `gethostbyname` and `gethostbyname_r` (`_r` for reentrant). The main difference is, you provide the storage for the `hostent` struct so you always have a local copy and not just a

pointer to the static copy.

### 3.5. How can I set the timeout for the connect() system call?

From Richard Stevens (rstevens@noao.edu):

Normally you cannot change this. Solaris does let you do this, on a per-kernel basis with the `ndd tcp_ip_abort_cinterval` parameter.

The easiest way to shorten the connect time is with an `alarm()` around the call to `connect()`. A harder way is to use `select()`, after setting the socket nonblocking. Also notice that you can only shorten the connect time, there's normally no way to lengthen it.

From Andrew Gierth (andrew@erlenstar.demon.co.uk):

First, create the socket and put it into non-blocking mode, then call `connect()`. There are three possibilities:

- o `connect` succeeds: the connection has been successfully made (this usually only happens when connecting to the same machine)
- o `connect` fails: obvious
- o `connect` returns `-1/EINPROGRESS`. The connection attempt has begun, but not yet completed.

If the connection succeeds:

- o the socket will `select()` as writable (and will also select as readable if data arrives)

If the connection fails:

- o the socket will select as readable *and* writable, but either a read or write will return the error code from the connection attempt. Also, you can use `getsockopt(SO_ERROR)` to get the error status - but be careful; some systems return the error code in the result parameter of `getsockopt`, but others (incorrectly) cause the `getsockopt` call *itself* to fail with the stored value as the error.

### 3.6. system choose one for me on the connect() call? Should I bind() a port number in my client program, or let the

From Andrew Gierth (andrew@erlenstar.demon.co.uk):

**\*\* Let the system choose your client's port number \*\***

The exception to this, is if the server has been written to be picky about what client ports it will allow connections from. Rlogind and rshd are the classic examples. This is usually part of a Unix-specific (and rather weak) authentication scheme; the intent is that the server allows connections only from processes with root privilege. (The weakness in the scheme is that many O/Ss (e.g. MS-DOS) allow anyone to bind any port.)

The rresvport() routine exists to help out clients that are using this scheme. It basically does the equivalent of socket() + bind(), choosing a port number in the range 512..1023.

If the server is not fussy about the client's port number, then don't try and assign it yourself in the client, just let connect() pick it for you.

If, in a client, you use the naive scheme of starting at a fixed port number and calling bind() on consecutive values until it works, then you buy yourself a whole lot of trouble:

The problem is if the server end of your connection does an active close. (E.G. client sends 'QUIT' command to server, server responds by closing the connection). That leaves the client end of the connection in CLOSED state, and the server end in TIME\_WAIT state. So after the client exits, there is no trace of the connection on the client end.

Now run the client again. It will pick the same port number, since as far as it can see, it's free. But as soon as it calls connect(), the server finds that you are trying to duplicate an existing connection (although one in TIME\_WAIT). It is perfectly entitled to refuse to do this, so you get, I suspect, ECONNREFUSED from connect(). (Some systems may sometimes allow the connection anyway, but you can't rely on it.)

This problem is especially dangerous because it doesn't show up unless the client and server are on different machines. (If they are the same machine, then the client won't pick the same port number as before). So you can get bitten well into the development cycle (if you do what

I suspect most people do, and test client & server on the same box initially).

Even if your protocol has the client closing first, there are still ways to produce this problem (e.g. kill the server).

### 3.7. Why do I get "connection refused" when the server isn't running?

The connect() call will only block while it is waiting to establish a connection. When there is no server waiting at the other end, it gets notified that the connection can not be established, and gives up with the error message you see. This is a good thing, since if it were not the case clients might wait for ever for a service which just doesn't exist. Users would think that they were only waiting for the connection to be established, and then after a while give up, muttering something about crummy software under their breath.

### 3.8. over the socket ? Is there a way to have a dynamic buffer ? What does one do when one does not know how much information is coming

This question asked by Niranjana Perera (perera@mindspring.com).

When the size of the incoming data is unknown, you can either make the size of the buffer as big as the largest possible (or likely) buffer, or you can re-size the buffer on the fly during your read. When you malloc() a large buffer, most (if not all) variants of unix will only allocate address space, but not physical pages of ram. As more and more of the buffer is used, the kernel allocates physical memory. This means that malloc'ing a large buffer will not waste resources unless that memory is used, and so it is perfectly acceptable to ask for a meg of ram when you expect only a few K.

On the other hand, a more elegant solution that does not depend on the inner workings of the kernel is to use realloc() to expand the buffer as required in say 4K chunks (since 4K is the size of a page of ram on most systems). I may add something like this to sockhelp.c in the example code one day.

## 4. Writing Server Applications (TCP/SOCK\_STREAM)

### 4.1. How come I get "address already in use" from bind()?

You get this when the address is already in use. (Oh, you figured

that much out?) The most common reason for this is that you have stopped your server, and then re-started it right away. The sockets that were used by the first incarnation of the server are still active. This is further explained in ``2.7 Please explain the TIME\_WAIT state.", and ``2.5 How do I properly close a socket?".

#### 4.2. Why don't my sockets close?

When you issue the close() system call, you are closing your interface to the socket, not the socket itself. It is up to the kernel to close the socket. Sometimes, for really technical reasons, the socket is kept alive for a few minutes after you close it. It is normal, for example for the socket to go into a TIME\_WAIT state, on the server side, for a few minutes. People have reported ranges from 20 seconds to 4 minutes to me. The official standard says that it should be 4 minutes. On my Linux system it is about 2 minutes. This is explained in great detail in ``2.7 Please explain the TIME\_WAIT state.".

#### 4.3. How can I make my server a daemon?

There are two approaches you can take here. The first is to use inetd to do all the hard work for you. The second is to do all the hard work yourself.

If you use inetd, you simply use stdin, stdout, or stderr for your socket. (These three are all created with dup() from the real socket) You can use these as you would a socket in your code. The inetd process will even close the socket for you when you are done.

If you wish to write your own server, there is a detailed explanation in "Unix Network Programming" by Richard Stevens (see ``1.6 Where can I get source code for the book [book title]?"). I also picked up this posting from comp.unix.programmer, by Nikhil Nair (nn201@cus.cam.ac.uk). You may want to add code to ignore SIGPIPE, because if this signal is not dealt with, it will cause your application to exit. (Thanks to ingo@milan2.snafu.de for pointing this out).

I worked all this lot out from the GNU C Library Manual (on-line documentation). Here's some code I wrote - you can adapt it as necessary:

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
```

```

#include <unistd.h>
#include <fcntl.h>
#include <signal.h>
#include <sys/wait.h>

/* Global variables */
volatile sig_atomic_t keep_going = 1; /* controls program termination */

/* Function prototypes: */
void termination_handler (int signum); /* clean up before termination */

int
main (void)
{
    ...

    if (chdir (HOME_DIR))          /* change to directory containing data
                                   files */
    {
        fprintf (stderr, "%s: ", HOME_DIR);
        perror (NULL);
        exit (1);
    }

    /* Become a daemon: */
    switch (fork ())
    {
        case -1:                  /* can't fork */
            perror ("fork()");
            exit (3);
        case 0:                   /* child, process becomes a daemon: */
            close (STDIN_FILENO);
            close (STDOUT_FILENO);
            close (STDERR_FILENO);
            if (setsid () == -1)   /* request a new session (job control) */
            {
                exit (4);
            }
            break;
        default:                  /* parent returns to calling process: */
            return 0;
    }

    /* Establish signal handler to clean up before termination: */

```

```

if (signal (SIGTERM, termination_handler) == SIG_IGN)
    signal (SIGTERM, SIG_IGN);
signal (SIGINT, SIG_IGN);
signal (SIGHUP, SIG_IGN);

/* Main program loop */
while (keep_going)
{
    ...
}
return 0;
}

void
termination_handler (int signum)
{
    keep_going = 0;
    signal (signum, termination_handler);
}

```

#### 4.4. How can I listen on more than one port at a time?

The best way to do this is with the `select()` call. This tells the kernel to let you know when a socket is available for use. You can have one process do i/o with multiple sockets with this call. If you want to wait for a connect on sockets 4, 6 and 10 you might execute the following code snippet:

```

fd_set socklist;

FD_ZERO(&socklist); /* Always clear the structure first. */
FD_SET(4, &socklist);
FD_SET(6, &socklist);
FD_SET(10, &socklist);
if (select(11, NULL, &socklist, NULL, NULL) < 0)
    perror("select");

```

The kernel will notify us as soon as a file descriptor which is less than 11 (the first parameter to `select()`), and is a member of our socklist becomes available for writing. See the man page on `select()` for more details.

#### 4.5. What exactly does `SO_REUSEADDR` do?

This socket option tells the kernel that even if this port is busy (in the `TIME_WAIT` state), go ahead and reuse it anyway. If it is busy, but with another state, you will still get an address already in use error. It is useful if your server has been shut down, and then restarted right away while sockets are still active on its port. You should be aware that if any unexpected data comes in, it may confuse your server, but while this is possible, it is not likely.

It has been pointed out that "A socket is a 5 tuple (proto, local addr, local port, remote addr, remote port). `SO_REUSEADDR` just says that you can reuse local addresses. The 5 tuple still must be unique!" by Michael Hunter ([mphunter@qnx.com](mailto:mphunter@qnx.com)). This is true, and this is why it is very unlikely that unexpected data will ever be seen by your server. The danger is that such a 5 tuple is still floating around on the net, and while it is bouncing around, a new connection from the same client, on the same system, happens to get the same remote port. This is explained by Richard Stevens in ``2.7 Please explain the `TIME_WAIT` state.".

#### 4.6. What exactly does `SO_LINGER` do?

On some unices this does nothing. On others, it instructs the kernel to abort tcp connections instead of closing them properly. This can be dangerous. If you are not clear on this, see ``2.7 Please explain the `TIME_WAIT` state.".

#### 4.7. What exactly does `SO_KEEPALIVE` do?

From Andrew Gierth ([andrew@erlenstar.demon.co.uk](mailto:andrew@erlenstar.demon.co.uk)):

The `SO_KEEPALIVE` option causes a packet (called a 'keepalive probe') to be sent to the remote system if a long time (by default, more than 2 hours) passes with no other data being sent or received. This packet is designed to provoke an ACK response from the peer. This enables detection of a peer which has become unreachable (e.g. powered off or disconnected from the net). See ``2.8 Why does it take so long to detect that the peer died?" for further discussion.

Note that the figure of 2 hours comes from RFC1122, "Requirements for Internet Hosts". The precise value should be configurable, but I've often found this to be difficult. The only implementation I know of that allows the keepalive interval to be set per-connection is SVR4.2.

#### 4.8. How can I bind() to a port number < 1024?

From Andrew Gierth (andrew@erlenstar.demon.co.uk):

The restriction on access to ports < 1024 is part of a (fairly weak) security scheme particular to UNIX. The intention is that servers (for example rlogind, rshd) can check the port number of the client, and if it is < 1024, assume the request has been properly authorised at the client end.

The practical upshot of this, is that binding a port number < 1024 is reserved to processes having an effective UID == root.

This can, occasionally, itself present a security problem, e.g. when a server process needs to bind a well-known port, but does not itself need root access (news servers, for example). This is often solved by creating a small program which simply binds the socket, then restores the real userid and exec(s) the real server. This program can then be made setuid root.

4.9. How do I get my server to find out the client's address / host-name?

From Andrew Gierth (andrew@erlenstar.demon.co.uk):

After accept()ing a connection, use getpeername() to get the address of the client. The client's address is of course, also returned on the accept(), but it is essential to initialise the address-length parameter before the accept call for this will work.

Jari Kokko (jkokko@cc.hut.fi) has offered the following code to determine the client address:

```
int t;
int len;
struct sockaddr_in sin;
struct hostent *host;

len = sizeof sin;
if (getpeername(t, (struct sockaddr *) &sin, &len) < 0)
    perror("getpeername");
else {
    if ((host = gethostbyaddr((char *) &sin.sin_addr,
                             sizeof sin.sin_addr,
                             AF_INET)) == NULL)
```

```
        perror("gethostbyaddr");
    else printf("remote host is '%s'\n", host->h_name);
}
```

#### 4.10. How should I choose a port number for my server?

The list of registered port assignments can be found in STD 2 or RFC 1700. Choose one that isn't already registered, and isn't in `/etc/services` on your system. It is also a good idea to let users customize the port number in case of conflicts with other unregistered port numbers in other servers. The best way of doing this is hardcoding a service name, and using `getservbyname()` to lookup the actual port number. This method allows users to change the port your server binds to by simply editing the `/etc/services` file.

#### 4.11. What is the difference between `SO_REUSEADDR` and `SO_REUSEPORT`?

`SO_REUSEADDR` allows your server to bind to an address which is in a `TIME_WAIT` state. It does not allow more than one server to bind to the same address. It was mentioned that use of this flag can create a security risk because another server can bind to a the same port, by binding to a specific address as opposed to `INADDR_ANY`. The `SO_REUSEPORT` flag allows multiple processes to bind to the same address provided all of them use the `SO_REUSEPORT` option.

From Richard Stevens ([rstevens@noao.edu](mailto:rstevens@noao.edu)):

This is a newer flag that appeared in the 4.4BSD multicasting code (although that code was from elsewhere, so I am not sure just who invented the new `SO_REUSEPORT` flag).

What this flag lets you do is rebind a port that is already in use, but only if all users of the port specify the flag. I believe the intent is for multicasting apps, since if you're running the same app on a host, all need to bind the same port. But the flag may have other uses. For example the following is from a post in February:

From Stu Friedberg ([stuartf@sequent.com](mailto:stuartf@sequent.com)):

`SO_REUSEPORT` is also useful for eliminating the try-10-times-to-bind hack in `ftpd`'s data connection setup routine. Without `SO_REUSEPORT`, only one `ftpd` thread can bind to `TCP(lhost, lport, INADDR_ANY, 0)` in preparation for connecting back to the client. Under conditions of heavy

load, there are more threads colliding here than the try-10-times hack can accomodate. With `SO_REUSEPORT`, things work nicely and the hack becomes unnecessary.

I have also heard that DEC OSF supports the flag. Also note that under 4.4BSD, if you are binding a multicast address, then `SO_REUSEADDR` is considered the same as `SO_REUSEPORT` (p. 731 of "TCP/IP Illustrated, Volume 2"). I think under Solaris you just replace `SO_REUSEPORT` with `SO_REUSEADDR`.

From a later Stevens posting, with minor editing:

Basically `SO_REUSEPORT` is a BSD'ism that arose when multicasting was added, even though it was not used in the original Steve Deering code. I believe some BSD-derived systems may also include it (OSF, now Digital Unix, perhaps?). `SO_REUSEPORT` lets you bind the same address \*and\* port, but only if all the binders have specified it. But when binding a multicast address (its main use), `SO_REUSEADDR` is considered identical to `SO_REUSEPORT` (p. 731, "TCP/IP Illustrated, Volume 2"). So for portability of multicasting applications I always use `SO_REUSEADDR`.

#### 4.12. How can I write a multi-homed server?

The original question was actually from Shankar Ramamoorthy ([shankar@viman.com](mailto:shankar@viman.com)):

I want to run a server on a multi-homed host. The host is part of two networks and has two ethernet cards. I want to run a server on this machine, binding to a pre-determined port number. I want clients on either subnet to be able to send broadcast packets to the port and have the server receive them.

And answered by Andrew Gierth ([andrew@erlenstar.demon.co.uk](mailto:andrew@erlenstar.demon.co.uk)):

Your first question in this scenario is, do you need to know which subnet the packet came from? I'm not at all sure that this can be reliably determined in all cases.

If you don't really care, then all you need is one socket bound to `INADDR_ANY`. That simplifies things greatly.

If you do care, then you have to bind multiple sockets. You are

obviously attempting to do this in your code as posted, so I'll assume you do.

I was hoping that something like the following would work.  
Will it? This is on Sparcs running Solaris 2.4/2.5.

I don't have access to Solaris, but I'll comment based on my experience with other Unixes.

[Shankar's original code omitted]

What you are doing is attempting to bind all the current hosts unicast addresses as listed in hosts/NIS/DNS. This may or may not reflect reality, but much more importantly, neglects the broadcast addresses. It seems to be the case in the majority of implementations that a socket bound to a unicast address will not see incoming packets with broadcast addresses as their destinations.

The approach I've taken is to use SIOCGIFCONF to retrieve the list of active network interfaces, and SIOCGIFFLAGS and SIOCGIFBRDADDR to identify broadcastable interfaces and get the broadcast addresses. Then I bind to each unicast address, each broadcast address, and to INADDR\_ANY as well. That last is necessary to catch packets that are on the wire with INADDR\_BROADCAST in the destination. (SO\_REUSEADDR is necessary to bind INADDR\_ANY as well as the specific addresses.)

This gives me very nearly what I want. The wrinkles are:

- o I don't assume that getting a packet through a particular socket necessarily means that it actually arrived on that interface.
- o I can't tell anything about which subnet a packet originated on if its destination was INADDR\_BROADCAST.
- o On some stacks, apparently only those with multicast support, I get duplicate incoming messages on the INADDR\_ANY socket.

#### 4.13. How can I read only one character at a time?

This question is usually asked by people who are testing their server with telnet, and want it to process their keystrokes one character at a time. The correct technique is to use a psuedo terminal (pty). More on that in a minute.

According to Roger Espel Lima (espel@drakkar.ens.fr), you can have your server send a sequence of control characters: 0xff 0xfb 0x01 0xff 0xfb 0x03 0xff 0xfd 0x0f3, which translates to IAC WILL ECHO IAC WILL SUPPRESS-GO-AHEAD IAC DO SUPPRESS-GO-AHEAD. For more information on what this means, check out std8, std28 and std29. Roger also gave the following tips:

- o This code will suppress echo, so you'll have to send the characters the user types back to the client if you want the user to see them.
- o Carriage returns will be followed by a null character, so you'll have to expect them.
- o If you get a 0xff, it will be followed by two more characters. These are telnet escapes.

Use of a pty would also be the correct way to execute a child process and pass the i/o to a socket.

I'll add pty stuff to the list of example source I'd like to add to the faq. If someone has some source they'd like to contribute (without copyright) to the faq which demonstrates use of pty's, please email me!

4.14. I'm trying to exec() a program from my server, and attach my socket's IO to it, but I'm not getting all the data across. Why?

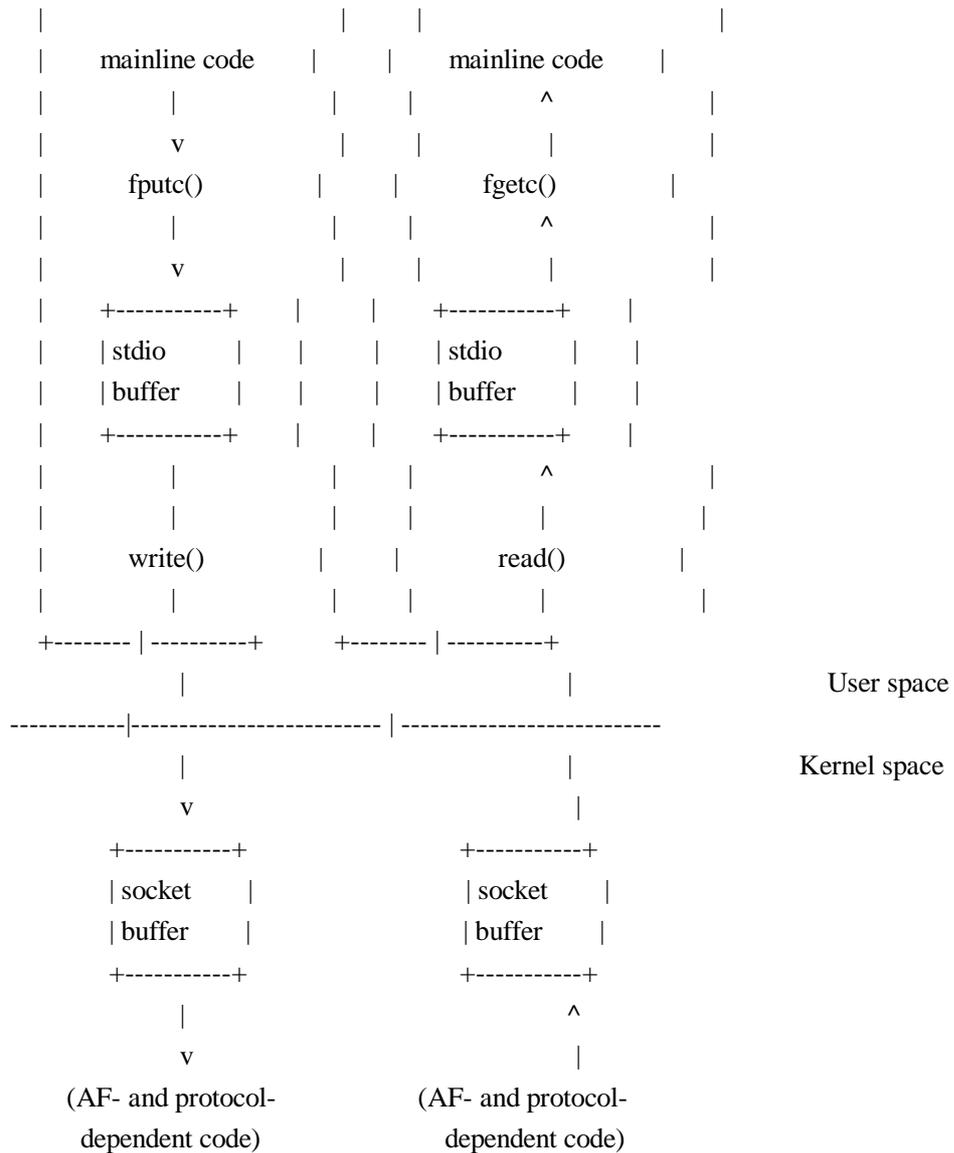
If the program you are running uses printf(), etc (streams from stdio.h) you have to deal with two buffers. The kernel buffers all socket IO, and this is explained in ``section 2.11''. The second buffer is the one that is causing you grief. This is the stdio buffer, and the problem was well explained by Andrew:

(The short answer to this question is that you want to use a pty rather than a socket; the remainder of this article is an attempt to explain why.)

Firstly, the socket buffer controlled by setsockopt() has absolutely nothing to do with stdio buffering. Setting it to 1 is guaranteed to be the Wrong Thing(tm).

Perhaps the following diagram might make things a little clearer:





Assuming these two processes are communicating with each other (I've deliberately omitted the actual comms mechanisms, which aren't really relevant), you can see that data written by process A to its `stdio` buffer is completely inaccessible to process B. Only once the decision is made to flush that buffer to the kernel (via `write()`) can the data actually be delivered to the other process.

The only guaranteed way to affect the buffering within process A is to change the code. However, the default buffering for `stdout` is controlled by whether the underlying FD refers to a terminal or not; generally, output to terminals is line-buffered, and output to non-terminals (including but not limited to files, pipes, sockets, non-ty devices, etc.) is fully buffered. So the desired effect can usually be achieved by using a pty device; this, for example, is what the 'expect' program does.

Since the stdio buffer (and the FILE structure, and everything else related to stdio) is user-level data, it is not preserved across an exec() call, hence trying to use setvbuf() before the exec is ineffective.

A couple of alternate solutions were proposed by Roger Espel Llima (espel@drakkar.ens.fr):

If it's an option, you can use some standalone program that will just run something inside a pty and buffer its input/output. I've seen a package by the name pty.tar.gz that did that; you could search around for it with archie or AltaVista.

Another option (\*\*warning, evil hack\*\*), if you're on a system that supports this (SunOS, Solaris, Linux ELF do; I don't know about others) is to, on your main program, putenv() the name of a shared executable (\*.so) in LD\_PRELOAD, and then in that .so redefine some commonly used libc function that the program you're exec'ing is known to use early. There you can 'get control' on the running program, and the first time you get it, do a setbuf(stdout, NULL) on the program's behalf, and then call the original libc function with a dlopen() + dlsym(). And you keep the dlsym() value on a static var, so you can just call that the following times.

(Editors note: I still haven't done an example for how to do pty's, but I hope I will be able to do one after I finish the non-blocking example code.)

## 5. Writing UDP/SOCK\_DGRAM applications

### 5.1. When should I use UDP instead of TCP?

UDP is good for sending messages from one system to another when the order isn't important and you don't need all of the messages to get to the other machine. This is why I've only used UDP once to write the example code for the faq. Usually TCP is a better solution. It saves you having to write code to ensure that messages make it to the desired destination, or to ensure the message ordering. Keep in mind that every additional line of code you add to your project in another line that could contain a potentially expensive bug.

If you find that TCP is too slow for your needs you may be able to get better performance with UDP so long as you are willing to sacrifice message order and/or reliability.

Philippe Jounin <ph.jounin@computer.org> would like to add...

In chapter 5.1 you say UDP allows more throughput than TCP.  
It is rarely the case if you have to pass several routers.

For instance, if you connect two LANs via X25 (a common way in Europe!), every UDP datagram will :

- o establish a Virtual Channel (VC)
- o send the data
- o close the VC,

whereas the VC remains during a TCP dialog.

UDP must be used to multicast messages to more than one other machine at the same time. With TCP an application would have to open separate connections to each of the destination machines and send the message once to each target machine. This limits your application to only communicate with machines that it already knows about.

5.2. What is the difference between "connected" and "unconnected" sockets?

From Andrew Gierth (andrew@erlenstar.demon.co.uk):

If a UDP socket is unconnected, which is the normal state after a bind() call, then send() or write() are not allowed, since no destination address is available; only sendto() can be used to send data.

Calling connect() on the socket simply records the specified address and port number as being the desired communications partner. That means that send() or write() are now allowed; they use the destination address and port given on the connect call as the destination of the packet.

5.3. of the socket? Does doing a connect() call affect the receive behaviour

From Richard Stevens (rstevens@noao.edu):

Yes, in two ways. First, only datagrams from your "connected peer" are returned. All others arriving at your port are not delivered to you.

But most importantly, a UDP socket must be connected to receive ICMP errors. Pp. 748-749 of "TCP/IP Illustrated, Volume 2" give all the gory details on why this is so.

#### 5.4. How can I read ICMP errors from "connected" UDP sockets?

If the target machine discards the message because there is no process reading on the requested port number, it sends an ICMP message to your machine which will cause the next system call on the socket to return `ECONNREFUSED`. Since delivery of ICMP messages is not guaranteed you may not receive this notification on the first transaction.

Remember that your socket must be "connected" in order to receive the ICMP errors. I've been told, and Alan Cox has verified that Linux will return them on "unconnected" sockets. This may cause porting problems if your application isn't ready for it, so Alan tells me they've added a `SO_BSDCOMPAT` flag which can be set for Linux kernels after 2.0.0.

#### 5.5. How can I be sure that a UDP message is received?

You have to design your protocol to expect a confirmation back from the destination when a message is received. Of course if the confirmation is sent by UDP, then it too is unreliable and may not make it back to the sender. If the sender does not get confirmation back by a certain time, it will have to re-transmit the message, maybe more than once. Now the receiver has a problem because it may have already received the message, so some way of dropping duplicates is required. Most protocols use a message numbering scheme so that the receiver can tell that it has already processed this message and return another confirmation. Confirmations will also have to reference the message number so that the sender can tell which message is being confirmed. Confused? That's why I stick with TCP.

#### 5.6. How can I be sure that UDP messages are received in order?

You can't. What you can do is make sure that messages are processed in order by using a numbering system as mentioned in "5.5 How can I be sure that a UDP message is received?". If you need your messages to be received and be received in order you should really consider

switching to TCP. It is unlikely that you will be able to do a better job implementing this sort of protocol than the TCP people already have, without a significant investment of time.

#### 5.7. How often should I re-transmit un-acknowledged messages?

The simplest thing to do is simply pick a fairly small delay such as one second and stick with it. The problem is that this can congest your network with useless traffic if there is a problem on the lan or on the other machine, and this added traffic may only serve to make the problem worse.

A better technique, described with source code in "UNIX Network Programming" by Richard Stevens (see ``1.6 Where can I get source code for the book [book title]?"), is to use an adaptive timeout with an exponential backoff. This technique keeps statistical information on the time it is taking messages to reach a host and adjusts timeout values accordingly. It also doubles the timeout each time it is reached as to not flood the network with useless datagrams. Richard has been kind enough to post the source code for the book on the web. Check out his home page at <http://www.naoa.edu/~rstevens>.

#### 5.8. How come only the first part of my datagram is getting through?

This has to do with the maximum size of a datagram on the two machines involved. This depends on the systems involved, and the MTU (Maximum Transmission Unit). According to "UNIX Network Programming", all TCP/IP implementations must support a minimum IP datagram size of 576 bytes, regardless of the MTU. Assuming a 20 byte IP header and 8 byte UDP header, this leaves 548 bytes as a safe maximum size for UDP messages. The maximum size is 65516 bytes. Some platforms support IP fragmentation which will allow datagrams to be broken up (because of MTU values) and then re-assembled on the other end, but not all implementations support this.

This information is taken from my reading of "UNIX Network Programming" (see ``1.6 Where can I get source code for the book [book title]?").

Andrew has pointed out the following regarding large UDP messages:

Another issue is fragmentation. If a datagram is sent which is too large for the network interface it is sent through, then the sending host will fragment it into smaller packets which are reassembled by

the receiving host. Also, if there are intervening routers, then they may also need to fragment the packet(s), which greatly increases the chances of losing one or more fragments (which causes the entire datagram to be dropped). Thus, large UDP datagrams should be avoided for applications that are likely to operate over routed nets or the Internet proper.

5.9. Why does the socket's buffer fill up sooner than expected?

From Paul W. Nelson (nelson@thursby.com):

In the traditional BSD socket implementation, sockets that are atomic such as UDP keep received data in lists of mbufs. An mbuf is a fixed size buffer that is shared by various protocol stacks. When you set your receive buffer size, the protocol stack keeps track of how many bytes of mbuf space are on the receive buffer, not the number of actual bytes. This approach is used because the resource you are controlling is really how many mbufs are used, not how many bytes are being held in the socket buffer. (A socket buffer isn't really a buffer in the traditional sense, but a list of mbufs).

For example: Lets assume your UNIX has a small mbuf size of 256 bytes. If your receive socket buffer is set to 4096, you can fit 16 mbufs on the socket buffer. If you receive 16 UDP packets that are 10 bytes each, your socket buffer is full, and you have 160 bytes of data. If you receive 16 UDP packets that are 200 bytes each, your socket buffer is also full, but contains 3200 bytes of data. FIONREAD returns the total number of bytes, not the number of messages or bytes of mbufs. Because of this, it is not a good indicator of how full your receive buffer is.

Additionally, if you receive UDP messages that are 260 bytes, you use up two mbufs, and can only receive 8 packets before your socket buffer is full. In this case, only 2080 bytes of the 4096 are held in the socket buffer.

This example is greatly simplified, and the real socket buffer algorithm also takes into account some other parameters. Note that some older socket implementations use a 128 byte mbuf.

## 6. Advanced Socket Programming

6.1. How would I put my socket in non-blocking mode?

From Andrew Gierth (andrew@erlenstar.demon.co.uk):

Technically, `fcntl(soc, F_SETFL, O_NONBLOCK)` is incorrect since it clobbers all other file flags. Generally one gets away with it since the other flags (`O_APPEND` for example) don't really apply much to sockets. In a similarly rough vein, you would use `fcntl(soc, F_SETFL, 0)` to go back to blocking mode.

To do it right, use `F_GETFL` to get the current flags, set or clear the `O_NONBLOCK` flag, then use `F_SETFL` to set the flags.

And yes, the flag can be changed either way at will.

6.2. How can I put a timeout on `connect()`?

Andrew Gierth (andrew@erlenstar.demon.co.uk) has outlined the following procedure for using `select()` with `connect()`, which will allow you to put a timeout on the `connect()` call:

First, create the socket and put it into non-blocking mode, then call `connect()`. There are three possibilities:

- o `connect` succeeds: the connection has been successfully made (this usually only happens when connecting to the same machine)
- o `connect` fails: obvious
- o `connect` returns `-1/EINPROGRESS`. The connection attempt has begun, but not yet completed.

If the connection succeeds:

- o the socket will `select()` as writable (and will also select as readable if data arrives)

If the connection fails:

- o the socket will select as readable *and* writable, but either a read or write will return the error code from the connection attempt. Also, you can use `getsockopt(SO_ERROR)` to get the error status - but be careful; some systems return the error code in the result parameter of `getsockopt()`, but others (incorrectly) cause the `getsockopt` call itself to fail with the stored value as the error.

Sample code that illustrates this can be found in the socket-faq examples, in the file connect.c.

## 7. Sample Source Code

The sample source code is no longer included in the faq. To get it, please download it from one of the unix-socket-faq www pages:

```
http://www.ibrado.com/sock-faq
http://kipper.york.ac.uk/~vic/sock-faq
http://www.ntua.gr/sock-faq
```

If you don't have web access, you can ftp it with ftpmail by following the following instructions.

To get the sample source by mail, send mail to ftpmail@decwrl.dec.com, with no subject line and a body like this:

```
reply <put your email address here>
connect ftp.zymsys.com
binary
uuencode
get pub/sockets/examples.tar.gz
quit
```

Save the reply as examples.uu, and type:

```
% uudecode examples.uu
% gunzip examples.tar.gz
% tar xf examples.tar
```

This will create a directory called socket-faq-examples which contains the sample code from this faq, plus a sample client and server for both tcp and udp.

Note that this package requires the gnu unzip program to be installed on your system. It is very common, but if you don't have it you can get the source for it from:

```
ftp://prep.ai.mit.edu/pub/gnu/gzip-1.2.4.tar
```

If you don't have ftp access, you can obtain it in a way similar to obtaining the sample source. I'll leave the exact changes to the body

of the message as an exercise for the reader.