Five pitfalls of Linux sockets programming

Develop reliable networked applications in heterogeneous environments

M. Tim Jones

September 20, 2005

The Sockets API is the de facto standard API for networking applications development. Although the API is simple, new developers can experience some common problems. This article identifies the most common of these pitfalls and shows you how to overcome them.

First introduced into the 4.2 BSD UNIX® operating system, the Sockets API is now a standard feature of any operating system. In fact, it's hard to find a modern language that doesn't support the Sockets API. The API is a relatively simple one, but new developers can still run into a few common pitfalls.

This article identifies those pitfalls and shows you how to avoid them.

Pitfall 1. Ignoring return status

The first pitfall is an obvious one, but it's an error that new developers make most often. If you ignore the return status of functions, you may miss when they fail or partially succeed. This, in turn, can propagate the error, making it difficult to locate the source of the problem.

Instead of ignoring status returns, capture and check each and every one. Consider the example of a socket `send` function shown in Listing 1.

Listing 1. Ignoring API function status return

```
int status, sock, mode;

/* Create a new stream (TCP) socket */
sock = socket( AF_INET, SOCK_STREAM, 0 );

...

status = send( sock, buffer, buflen, MSG_DONTWAIT );

if (status == -1) {

  /* send failed */
  printf( "send failed: %s\n", strerror(errno) );

} else {

  /* send succeeded -- or did it? */

}
```

Listing 1 explores a function snippet that performs a socket `send` (sending data through a socket). The error status of the function is captured and tested, but this example ignores a feature of `send` in non-blocking mode (enabled by the `MSG_DONTWAIT` flag).

Three classes of return values are possible from the `send` API function:

- If the data has been successfully queued for transmission, a zero is returned.
- If a failure has occurred, a -1 is returned (and that failure can be understood through the use of the `errno` variable).
- If not all characters could be queued in the call, the number of characters sent is the final return value.

Because of the non-blocking nature of the `MSG_DONTWAIT` variant of `send`, the call returns after sending all, some, or none of the data. Ignoring the return status here would result in an incomplete send and subsequent loss of data.

Pitfall 2. Peer socket closure

One of the interesting aspects of UNIX is that you can view almost everything as a file. Files themselves, directories, pipes, devices, and sockets are treated as files. This is a novel abstraction and means that a collective set of APIs can be used over a wide range of device types.

Consider the `read` API function, which reads some number of bytes from a file. The `read` function returns the number of bytes read (up to the maximum that you specify), -1 on error, or zero if the end of the file has been reached.

If you read from a file and reach the end (indicated by a zero-length read), you'd close the file and be done. The same thing applies in a socket, but the semantics are a little different. If you perform a `read` on a socket and get a zero return, this indicates that the peer at the remote end of the socket has called the `close` API function. The indication is the same as the file read -- no more data can be read through the descriptor (see Listing 2).

Listing 2. Proper handling of the read API function return value

```
int sock, status;

sock = socket( AF_INET, SOCK_STREAM, 0 );

...

status = read( sock, buffer, buflen );

if (status > 0) {

  /* Data read from the socket */

} else if (status == -1) {

  /* Error, check errno, take action... */

} else if (status == 0) {

  /* Peer closed the socket, finish the close */
  close( sock );

  /* Further processing... */

}
```

The closure of a peer socket can also be detected with the `write` API function. In this case, you'll receive a `SIGPIPE` signal or, if this signal is blocked, the `write` function will return a -1 and set `errno` to `EPIPE`.

Pitfall 3. Address in use error (EADDRINUSE)

You can use the `bind` API function to bind an address (an interface and a port) to a socket endpoint. You can use this function in a server setting to restrict the interfaces from which incoming connections are possible. You can also use this function from a client setting to restrict the interface that should be used for an outgoing connection. The most common use of `bind` is to associate a port number with a server and use the wildcard address (`INADDR_ANY`), which allows any interface to be used for incoming connections.

The problem commonly encountered with `bind` is attempting to bind a port that's already in use. The pitfall is that no active socket may exist, but binding to the port is still disallowed (`bind` returns `EADDRINUSE`), which is caused by the TCP socket `TIME_WAIT` state. This state keeps a socket around for two to four minutes after its close. After the `TIME_WAIT` state has exited, the socket is removed, and the address can be rebound without issue.

Waiting for `TIME_WAIT` to finish can be annoying, especially if you're developing a socket server and you need to stop the server to make changes and then restart it. Luckily, there's a way to get around the `TIME_WAIT` state. You can apply the `SO_REUSEADDR` socket option to the socket, such that the port can be reused immediately.

Consider the example in Listing 3. Prior to binding an address, I call `setsockopt` with the `SO_REUSEADDR` option. To enable address reuse, I set the integer argument (`on`) to 1 (otherwise, you can set it to 0 to disable address reuse).

Listing 3. Avoiding the "Address In Use" error using the SO_REUSEADDR socket option

```
int sock, ret, on;
struct sockaddr_in servaddr;

/* Create a new stream (TCP) socket */
sock = socket( AF_INET, SOCK_STREAM, 0 ):

/* Enable address reuse */
on = 1;
ret = ( sock, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on) );

/* Allow connections to port 8080 from any available interface */
memset( &servaddr, 0, sizeof(servaddr) );
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl( INADDR_ANY );
servaddr.sin_port = htons( 45000 );

/* Bind to the address (interface/port) */
ret = bind( sock, (struct sockaddr *)&servaddr, sizeof(servaddr) );
```

After you have applied the `SO_REUSEADDR` socket option, the `bind` API function will always permit immediate reuse of the address.

Pitfall 4. Sending structured data

Sockets are a perfect vehicle for sending unstructured binary byte-streams or ASCII streams of data (such as HTML pages over HTTP, or e-mail over SMTP). But if you try to send binary data over a socket, it becomes much more complicated.

Let's say you want to send an integer through a socket: can you be certain that the receiver will interpret the integer in the same way? Applications running on similar architectures can rely on their common platforms to interpret the type identically. But what happens if a client running on a big endian IBM PowerPC attempts to send a 32-bit integer to a little endian Intel x86? Byte ordering will cause the value to be interpreted incorrectly.

To byte-swap or not?

Endianness refers to the byte ordering in memory. *Big endian* orders by most-significant byte first, whereas *little endian* orders by least-significant byte first.

Big endian architectures, such as the PowerPC®, have an advantage over little endian architectures such as the Intel® Pentium® series in that network byte order is big endian. This means that control data within the TCP/IP stack is naturally in order for big endian machines. Little endian architectures require byte-swapping -- a slight performance disadvantage for networked applications.

What about sending a C structure through a socket? You can run into trouble here as well, because not all compilers align the elements of a structure in the same way. The structure could also be packed to minimize wasted space, further misaligning the elements in the structure.

Fortunately, there are solutions to this problem that ensure consistent interpretation of data by both endpoints. In the old days, the Remote Procedure Call (RPC) toolkit provided what was called

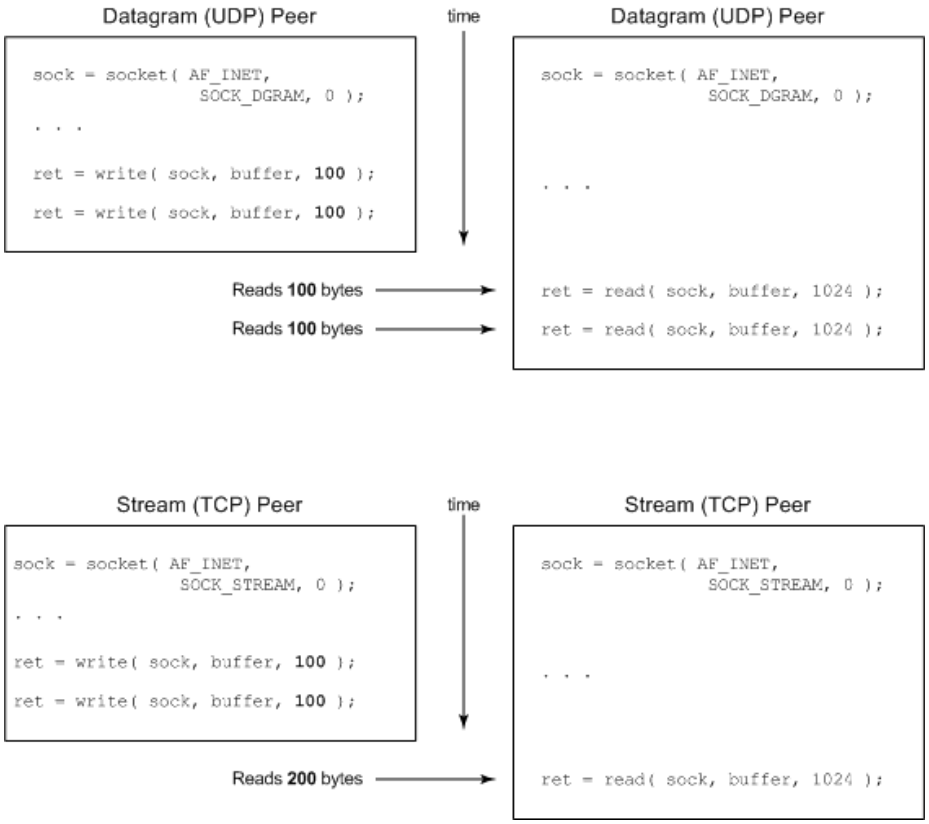
External Data Representation (XDR). XDR defined a standard representation for data to support the development of communicating heterogeneous network applications.

Today, a couple of newer protocols provide a similar capability. The Extensible-Markup-Language/ Remote Procedure Call protocol (XML/RPC) marshals procedure calls over HTTP in an XML format. Data and metadata are encoded within XML and transported as ASCII strings, disassociating the values from their physical representation by the host architecture. SOAP followed XML-RPC and extends its ideas with greater features and functionality. See the Related topics section for more information on each of these protocols.

Pitfall 5. Framing assumptions in TCP

TCP provides no framing, which makes it perfect for byte-stream-oriented protocols. This is one of the key differences between TCP and the User Datagram Protocol (UDP). UDP is a message-oriented protocol that preserves the boundaries of messages between the sender and receiver. TCP is a stream-based protocol that assumes the data being communicated is unstructured, as shown in Figure 1.

Figure 1. Framing capabilities of UDP and the lack of framing in TCP



The top of Figure 1 illustrates a UDP client and server. The peer on the left performs two socket writes of 100 bytes each. The UDP layer of the stack keeps track of the quantities of the writes and ensures that when the receiver on the right gets the data through the socket, it arrives in the same quantities. In other words, the boundaries of the messages that the writer provides are preserved for the reader.

Now, look at the bottom of Figure 1. It demonstrate the same granularity of writes for the TCP layer. Two independent writes to the stream socket of 100 bytes each are performed. But in this case, the reader of the stream socket gets 200 bytes. The TCP layer of the stack has aggregated the two writes. This aggregation can occur in either the sender or receiver TCP/IP stacks. It's important to note that the aggregation may not occur -- TCP guarantees only ordered delivery of the data.

This pitfall causes a quandary for most developers. You want the reliability of TCP but the framing aspects of UDP. Other than switching to a different transport protocol, such as the Stream Transmission Control Protocol (STCP), it's up to the application layer developer to implement the buffering and segmenting functionality.

Tools for debugging sockets applications

GNU/Linux provides several debugging tools that can help you uncover problems in sockets applications. Further, using these tools can also be educational and help explain the behavior of your application and the TCP/IP stack. You'll get a quick overview of a few tools here, but check out the Related topics below to learn more.

Viewing details of the networking subsystem

The `netstat` tool provides visibility into the GNU/Linux networking subsystem. With `netstat`, you can view currently active connections (on a per-protocol basis), view connections in a particular state (such as server sockets in the listening state), and many others. Listing 4 shows some of the options that `netstat` provides and the features they enable.

Listing 4. Usage patterns for the netstat utility

```
View all TCP sockets currently active
$ netstat --tcp

View all UDP sockets
$ netstat --udp

View all TCP sockets in the listening state
$ netstat --listening

View the multicast group membership information
$ netstat --groups

Display the list of masqueraded connections
$ netstat --masquerade

View statistics for each protocol
$ netstat --statistics
```

A lot of other utilities exist, but `netstat` tends to be a one-stop shop that covers the capabilities of `route`, `ifconfig`, and other standard GNU/Linux tools.

Watching the traffic go by

With GNU/Linux, you can use several tools to inspect the low-level traffic on a network. The `tcpdump` tool is an older tool that "sniffs" network packets from a network and either prints them to

`stdout` or logs them to a file. This functionality allows you to see the traffic that your application generates and also the low-level flow-control mechanisms that TCP generates. A newer tool called `tcpflow` complements `tcpdump` and provides a way to do protocol flow analysis and to properly reconstruct data streams, regardless of packet order or retransmissions. A couple of usage patterns for `tcpdump` are shown in Listing 5.

Listing 5. Usage patterns for the tcpdump tool

```
Display all traffic on the eth0 interface for the local host
$ tcpdump -l -i eth0

Show all traffic on the network coming from or going to host plato
$ tcpdump host plato

Show all HTTP traffic for host camus
$ tcpdump host camus and (port http)

View traffic coming from or going to TCP port 45000 on the local host
$ tcpdump tcp port 45000
```

The `tcpdump` and `tcpflow` tools give you a huge number of options, including the ability to create complex filter expressions. Check the Related topics below for more information on these tools.

Both `tcpdump` and `tcpflow` are text-based command-line tools. If you prefer a graphical user interface (GUI), an open source tool called `Ethereal` may fit the bill. `Ethereal` is a professional protocol analyzer that can help debug application layer protocols. Its plug-in architecture can decompose protocols such as HTTP or any other protocol you can think of (637 protocols at the time of this writing).

Wrapup

Sockets programming is easy and enjoyable, especially if you avoid introducing bugs or at least make them easy to find by considering the five common pitfalls described in this article, in addition to standard defensive programming practices. GNU/Linux tools and utilities can also help bring to light problems in your programs. Remember: when checking out the man page of a utility, follow the related or "see also" tools. You might find a new tool that you can't do without.

Related topics

- Explore the history and implications of Endianness at Wikipedia.
- Read an introduction to RPC/XDR from the Programming in C courseware.
- SOAP builds on the features of XML-RPC. Find specifications, implementations, and tutorials and articles at SoapWare.Org.
- The tutorial "Programming Linux sockets, Part 1" (developerWorks, October 2003) shows how to begin programming with sockets and how to build an echo server and client that connect over TCP/IP. "Programming Linux sockets, Part 2" (developerWorks, January 2004) focuses on UDP and shows how to write UDP sockets applications in C and in Python (although the code will translate well to other languages).
- The tcpdump and tcpflow utilities can be used to monitor network traffic.
- The Ethereal network protocol analyzer provides the functionality of tcpdump but with a graphical UI and scalable plug-in architecture.
- Build your next development project on Linux with IBM trial software, available for download directly from developerWorks.