HOW TO AVOID THE

TOP TEN SOCKETS PROGRAMMING ERRORS

A WHITE PAPER FOR
SOFTWARE DEVELOPMENT MANAGERS.

RIVERACE

Riverace Corporation
10 Wampanoag Drive
Franklin, MA 02038-1292
888-384-8154 (U.S.)
508-541-9180 (Outside the U.S.)
www.riverace.com

December 2007

Introduction

Today's trade journals are full of stories about the latest hot trends and technologies. SOA. Web services. AJAX. These are cool — if you're a Web programmer. But if you're like most senior software developers or managers we've worked with, the real workhorse in your applications and systems is plain old Sockets — the low-level connections that link applications running on a network.

Your systems move financial data, audio/video, patient health information, billing records — pretty much anything and everything, text or binary. Lots of data must move efficiently with nary a Web server in sight. You may need to run on Windows, Linux, and UNIX across different hardware types and sizes from data center servers and desktops to handheld devices and embedded hardware.

Given the demands of today's business world, everything had better work right, all the time. If a mistake happens or your system crashes, your business not only loses money — its hard-earned reputation for quality also suffers, and you personally are on the hot seat for the mistake.

This white paper will show you the 10 most common errors developers make when programming Sockets — and how you can avoid them.

Part One

The 10 Most Common Sockets Programming Errors

Much of the cost of software development is in the testing and maintenance (fixing the bugs) stages. Although developers know about errors and where they're most likely to occur, it's proven very difficult to avoid certain classes of programming errors — and those commonly encountered when programming Sockets are among the thorniest. Why is this? The reasons Sockets programming is very error-prone can be grouped into three categories:

1. It's too easy to do the wrong thing.
2. The application program interface (API) is too complex.
3. Despite Sockets' ubiquity, it's still not 100% portable.

To top it off, most of the errors can't be caught by the compiler, forcing developers to find and resolve these errors during testing, a far more costly effort than if the errors could be detected at compile time.

As an illustration, consider the following simple function. This code will successfully compile and link on most systems, yet it contains the ten most commonly made Sockets errors. Can you identify them?

// The following example contains bugs! Do not copy this example!

```
#include <sys/types.h>
#include <sys/socket.h>

const int PORT_NUM = 10000;

int echo_server ()
{
  struct sockaddr_in addr;
  int addr_len;
  char buf[BUFSIZ];
11   int n_handle;
12   // Create the local endpoint.
13   int s_handle = socket (PF_UNIX, SOCK_DGRAM, 0);
14   if (s_handle == -1) return -1;
15
16   // Set up the address where the server listens.
17   addr.sin_family = AF_INET;
18   addr.sin_port = PORT_NUM;
19   addr.sin_addr.addr = INADDR_ANY;
20
21   if (bind (s_handle, (struct sockaddr *) &addr,
22              sizeof addr) == -1)
23     return -1;
24
25   // Create a new communication endpoint.
26   if (n_handle = accept (s_handle, (struct sockaddr *)
27                                 &addr, &addr_len) != -1) {
28     int n;
29     while ((n = read (s_handle, buf, sizeof buf)) > 0)
30       write (n_handle, buf, n);
31
32     close (n_handle);
33   }
34   return 0;
35 }
```

Were you able to identify all 10 errors? Even battle-tested developers have had trouble doing so. Here are the explanations:

1. **Forgot to initialize an important variable (lines 8-9).** The addr_len variable must be set to sizeof(addr). Forgetting to initialize this variable will cause the accept() call on line 26 to fail at run time.

2. **Use of nonportable handle datatype (lines 11-14).** Although these lines look harmless enough, they're fraught with peril. This code isn't portable to Windows Sockets (WinSock) platforms, where socket handles are type SOCKET, not type int. Moreover, WinSock failures are indicated via a non-standard macro called INVALID_SOCKET_HANDLE rather than by returning -1. Other bugs exist in this section of the code fragment, but aren't obvious until we examine the rest of the function.

3. **Unused struct members not cleared (lines 17-19).** The entire addr structure should have been initialized to 0 before setting each address member. The Socket API uses one basic addressing structure (sockaddr) with different overlays, depending on the address family, for example, sockaddr_in for IPv4. Without initializing the entire structure to 0, parts of the fields may have indeterminate contents, which can yield random run-time failures.

4. **Address/protocol family mismatch (line 17).** The addr.sin_family field was set to AF_INET, which designates the Internet addressing family. It will be used with a socket (s_handle) that was created with the UNIX protocol family, which is inconsistent. Instead, the protocol family type passed to the socket() function should have been PF_INET.

5. **Wrong byte order (line 18).** The value assigned to addr.sin_port is not in network byte order; that is, the programmer forgot to use htons() to convert the port number from host byte order into network byte order. When run on a computer with little-endian byte order, this code will execute without error; however, clients will be unable to connect to it at the expected port number.

If these network-addressing mistakes were corrected, lines 21 – 23 would actually work! Small consolation, though, since five errors remain in the code, including an interrelated set of errors on lines 25 – 27. These mistakes exemplify the difficulty in locating errors before run-time when programming directly to the C Sockets API.

6. **Missing an important API call (line 25).** The listen() function was omitted accidentally. This function must be called before accept() to set the socket handle into so-called "passive mode."

7. **Wrong socket type for API call (line 26).** The accept() function was called for s_handle, which is exactly what should have been done. The s_handle was created as a SOCK_DGRAM-type socket, however, which is an illegal socket type to use with accept(). The original socket() call on line 13 should therefore have been passed the SOCK_STREAM flag.

8. **Operator precedence error (lines 26-27).** There's one further error related to the accept() call. As written, n_handle will be set to 1 if accept() succeeds and 0 if it fails. If this program runs on UNIX (and the other bugs are fixed), data will be written to either standard output or standard input, respectively, instead of the connected socket. Most of the errors in this example can be avoided by using the ACE programming toolkit (we'll get to that), but this bug is simply an error in operator precedence. It can't be avoided by using ACE, or any other library. It's a common pitfall that illustrates all-too-forgotten wisdom that any good tool still requires the user's experience and skill.

9. **Wrong handle used in API call (line 29).** The read() function was called to receive up to sizeof buf bytes from s_handle, which is the passive-mode listening socket. It should have called read() on n_handle instead. This problem would have manifested itself as an obscure run-time error and could not have been detected at compile time because socket handles are weakly typed.

10. **Possible data loss (line 30).** The return value from write() was not checked to see if all n bytes (or any bytes at all) were written, which is a possible source of data loss. Due to socket buffering and flow control, a write() to a bytestream mode socket may only send part of the requested number of bytes, in which case the rest must be sent later.

If you've been programming Sockets using the C API for a while, you probably picked up a few of these errors right away. However, you may not have caught all of them, and developers without years of battle-earned experience make these mistakes all the time — costing your project precious time and resources.

The more important question, however, is why should you have to worry about mistakes when using the API — it's basic plumbing that shouldn't require your attention. Instead, you and your developers should be using your time and skills to implement the real value of your application.

Part Two

Avoid Sockets Complexities with ACE

To keep pace in our current competitive landscape, today's software developers are under increasing pressure to produce high-quality, high-performance systems, in shorter time-to-market windows. In order to keep pace, more and more project teams are turning to modern software engineering methods and practices as a means for improving their team's productivity and rate of success.

Object-oriented analysis and design, C++, and design patterns are helping today's harried engineering teams put their expertise into domain-specific engineering problems without having to solve many of the same framework issues over and over on each new project.

The ADAPTIVE Communication Environment (ACE) is a modern, high-performance framework and C++ class library that provides a wide variety of classes and patterns that allow you to concentrate on adding the high-value domain-specific knowledge while completing your projects in shorter times.

And, since ACE provides the framework and patterns commonly found in high-performance, networked, multi-threaded systems, development teams have much less code to write (and debug), reducing both your engineering and maintenance costs, now and in the future.

The ACE C++ toolkit offers classes designed to resolve the complexities inherent in Sockets programming. The design makes it easy to implement communications correctly — while making it difficult to do incorrectly. Additionally, the program is easily portable to a very wide range of today's popular computing platforms — from handheld devices to supercomputers.

As an example of how ACE can save you time and frustration, here is the same error-prone program from above rewritten using ACE's socket wrapper classes:

// This example works fine! Copy at will!

```
#include <ace/INET_Addr.h>
#include <ace/SOCK_Acceptor.h>
#include <ace/SOCK_Stream.h>

const u_short PORT_NUM = 10000;

int echo_server ()
{
  ACE_INET_Addr addr (PORT_NUM);
  char buf[BUFSIZ];
  // Create the local listener.
  ACE_SOCK_Acceptor acceptor;
  if (acceptor.open (addr) == -1)
    return -1;
  // Accept a new connection.
  ACE_SOCK_Stream sock;
  if (acceptor.accept (sock) != -1) {
    ssize_t n;
    while ((n = sock.recv (buf, sizeof buf)) > 0)
      sock.send_n (buf, static_cast<size_t> (n));
    sock.close ();
  }
  return 0;
}
```

Result: No errors, fewer lines of code.
In this 100% error-free program code, you'll find three benefits: it's much easier to understand what's going on (even for novices), the program is easily portable, and it requires six fewer lines of code.

Moreover, if the code is written to call an inappropriate method, such as attempting to receive data from the acceptor object, the compiler can quickly flag the error since no data transfer methods have been defined on the ACE_SOCK_Acceptor class. Thus, **the**

error is corrected in seconds — versus spending hours or days trying to debug it at run time.

Try the ACE Source Code – at No Cost

Find out how much time and frustration this type of improvement can save you — download your free, no-obligation ACE source code right now.

DOWNLOAD NOW

http://www.riverace.com/try_ace.htm

Take a look. Kick the tires. It won't cost you anything and may save you lots of grief. (We promise a sales person won't call to bug you — and we guarantee you'll wonder how you ever lived without ACE.)

If you have a particular project in progress and would like to discuss how ACE could be applied to help you, please call Steve Huston at Riverace Corporation today at **888-384-8154 (U.S.) or 508-541-9180 (outside the U.S.) for a no-strings-attached phone consultation.** You'll quickly learn if ACE can be useful to your project and you'll get some fast direction on how to proceed.

RIVERACE

ABOUT RIVERACE CORPORATION

Riverace Corporation, founded in 1994, grew out of a tradition of high-quality consulting services specializing in network system and protocol development, and object-oriented development of distributed object systems. ACE has played a pivotal role in Riverace's successful history.

Riverace delivers superior ACE software development, support, training, and consulting services with enthusiastic customers and their successful projects spread across the United States and around the world.

For further information about any of our services, please call +1.888-384-8154 or send email to info@riverace.com.

RIVERACE

Riverace Corporation
10 Wampanoag Drive
Franklin, MA 02038-1292
888-384-8154 (U.S)
508-541-9180 (Outside the U.S.)

www.riverace.com