Foreword by Douglas C. Schmidt, Inventor of ACE

THE ACE PROGRAMMER'S GUIDE

Practical Design Patterns for Network and Systems Programming

Stephen D. Huston • James CE Johnson
Umar Syyid

The ACE Programmer's Guide

This method creates a basic BSD-style socket. The most common usage will be as shown in the preceding example, where we provide an address at which to listen and the `reuse_addr` flag. The `reuse_addr` flag is generally encouraged so that your server can accept connections on the desired port even if that port was used for a recent connection. If your server is not likely to service new connection requests rapidly, you may also want to adjust the `backlog` parameter.

Once you have an address defined and have opened the acceptor to listen for new connections, you want to wait for those connection requests to arrive. This is done with the `accept()` method, which closely mirrors the `accept()` function:

```
if (acceptor.accept (peer) == -1)
  ACE_ERROR_RETURN ((LM_ERROR,
                     ACE_TEXT ("(%P|%t) Failed to accept ")
                     ACE_TEXT ("client connection\n")),
                    100);
```

This use will block until a connection attempt is made. To limit the wait time, supply a timeout:

```
if (acceptor.accept (peer, &peer_addr, &timeout, 0) == -1)
  {
    if (ACE_OS::last_error() == EINTR)
      ACE_DEBUG ((LM_DEBUG,
                  ACE_TEXT ("(%P|%t) Interrupted while ")
                  ACE_TEXT ("waiting for connection\n")));
    else
      if (ACE_OS::last_error() == ETIMEDOUT)
        ACE_DEBUG ((LM_DEBUG,
                    ACE_TEXT ("(%P|%t) Timeout while ")
                    ACE_TEXT ("waiting for connection\n")));
  }
```

If no client connects in the specified time, you can at least print a message to let the administrator know that your application is still open for business. As we learned in Chapter 3, we can easily turn those things off if they become a nuisance.

Regardless of which approach you take, a successful return will provide you a valid peer object initialized and representing a connection to the client. It is worth noting that by default, the `accept()` method will restart itself if it is interrupted by a UNIX signal, such as `SIGALRM`. That may or may not be appropriate for your application. In the preceding example, we have chosen to pass 0 as the fourth

parameter (`restart`) of the `accept()` method. This will cause `accept()` to return –1 and `ACE_OS::last_error()` to return `EINTR` if the action is interrupted. Because we specified `peer_address` in the preceding example, `accept()` will fill in the address of the peer that connects—if the `accept()` succeeds.

Another handy thing we can do with `ACE_INET_Addr` is extract a string for its address. Using this method, we can easily display the new peer's address:

```
else
  {
    ACE_TCHAR peer_name[MAXHOSTNAMELEN];
    peer_addr.addr_to_string (peer_name, MAXHOSTNAMELEN);
    ACE_DEBUG ((LM_DEBUG,
                ACE_TEXT ("(%P|%t) Connection from %s\n"),
                peer_name));
```

The `addr_to_string()` method requires a buffer in which to place the string and the size of that buffer. This method takes an optional third parameter, specifying the format of the string it creates. Your options are (0) ip-name:port-number and (1) ip-number:port-number. If the buffer is large enough for the result, it will be filled and null terminated appropriately, and the method will return 0. If the buffer is too small, the method will return –1, indicating an error.

Now that we have accepted the client connection, we can begin to work with it. At this point, the distinction between client and server begins to blur because you simply start sending and receiving data. In some applications, the server will send first; in others, the client will do so. How your application behaves depends on your requirements and protocol specification. For our purposes, we will assume that the client is going to send a request that the server will simply echo back:

```
char buffer[4096];
ssize_t bytes_received;

while ((bytes_received =
          peer.recv (buffer, sizeof(buffer))) != -1)
  {
    peer.send_n (buffer, bytes_received);
  }

peer.close ();
```

As our examples become more robust in future chapters, we will begin to process those requests into useful actions.

As the server is written, it will process only one request on one client connection and then exit. If we wrap a simple `while` loop around everything following the `accept()`, we can service multiple clients but only one at a time:

```
#include "ace/INET_Addr.h"
#include "ace/SOCK_Stream.h"
#include "ace/SOCK_Acceptor.h"
#include "ace/Log_Msg.h"

int ACE_TMAIN (int, ACE_TCHAR *[])
{
  ACE_INET_Addr port_to_listen ("HAStatus");
  ACE_SOCK_Acceptor acceptor;

  if (acceptor.open (port_to_listen, 1) == -1)
    ACE_ERROR_RETURN ((LM_ERROR,
                       ACE_TEXT ("%p\n"),
                       ACE_TEXT ("acceptor.open")),
                      100);

  /*
   * The complete open signature:
   *
  int open (const ACE_Addr &local_sap,
            int reuse_addr = 0,
            int protocol_family = PF_INET,
            int backlog = ACE_DEFAULT_BACKLOG,
            int protocol = 0);
   *
   */

  while (1)
    {
      ACE_SOCK_Stream peer;
      ACE_INET_Addr peer_addr;
      ACE_Time_Value timeout (10, 0);

      /*
       * Basic acceptor usage
       */
#if 0
      if (acceptor.accept (peer) == -1)
        ACE_ERROR_RETURN ((LM_ERROR,
```

```
                              ACE_TEXT ("(%P|%t) Failed to accept ")
                              ACE_TEXT ("client connection\n")),
                             100);
#endif /* 0 */

      if (acceptor.accept (peer, &peer_addr, &timeout, 0) == -1)
        {
          if (ACE_OS::last_error() == EINTR)
            ACE_DEBUG ((LM_DEBUG,
                        ACE_TEXT ("(%P|%t) Interrupted while ")
                        ACE_TEXT ("waiting for connection\n")));
          else
            if (ACE_OS::last_error() == ETIMEDOUT)
              ACE_DEBUG ((LM_DEBUG,
                          ACE_TEXT ("(%P|%t) Timeout while ")
                          ACE_TEXT ("waiting for connection\n")));
        }
      else
        {
          ACE_TCHAR peer_name[MAXHOSTNAMELEN];
          peer_addr.addr_to_string (peer_name, MAXHOSTNAMELEN);
          ACE_DEBUG ((LM_DEBUG,
                      ACE_TEXT ("(%P|%t) Connection from %s\n"),
                      peer_name));
          char buffer[4096];
          ssize_t bytes_received;

          while ((bytes_received =
                    peer.recv (buffer, sizeof(buffer))) != -1)
            {
              peer.send_n (buffer, bytes_received);
            }

          peer.close ();
        }
    }

  return (0);
}
```

Such a server isn't very realistic, but it's important that you see the entire example. We will return to this simple server in Chapter 7 to enhance it with the ability to handle multiple, concurrent clients.

If you've worked with the Sockets API directly, you may have noticed something surprising about our examples. We have not once referenced a handle value. If you haven't worked with the Sockets API before, a handle is an opaque chunk of native data representing the socket. Direct handle use is a continual source of accidental complexity and, thus, errors, when using the native system functions. However, ACE's class design properly encapsulates handles, so you will almost never care what the value is, and you will never have to use a handle value directly to perform any I/O operation.

6.4 Summary

The ACE TCP/IP socket wrappers provide you with a powerful yet easy-to-use set of tools for creating client/server applications. Using what you've learned in this chapter, you will be able to convert nearly all your traditionally coded, error prone, nonportable networked applications to true C++ object-oriented, portable implementations.

By using the ACE objects, you can create more maintainable and portable applications. Because you're working at a higher level of abstraction, you no longer have to deal with the mundane details of network programming, such as remembering to zero out those `sockaddr_in` structures and when and what to cast them to. Your application becomes more type safe, which allows you to avoid more errors. Furthermore, when there are errors, they're much more likely to be caught at compile time than at runtime.

Chapter 7
Handling Events and Multiple I/O Streams

Many applications, such as the server example in Chapter 6, can benefit greatly from a simple way to handle multiple events easily and efficiently. Event handling often takes the form of an *event loop* that continually waits for events to occur, decides what actions need to be taken, and dispatches control to other functions or methods appropriate to handle the event(s). In many networked application projects, the event-handling code is often the first piece of the system to be developed, and it's often developed over and over for each new project, greatly adding to the time and cost for many projects.

The ACE Reactor framework was designed to implement a flexible event-handling mechanism in such a way that applications need never write the central, platform-dependent code for their event-handling needs. Using the Reactor framework, applications need do only three things to implement their event handling.

1. Derive one or more classes from `ACE_Event_Handler` and add application-specific event-handling behavior to virtual *callback methods*.
2. Register the application's event-handling objects with the `ACE_Reactor` class and associate each with the event(s) of interest.
3. Run the `ACE_Reactor` event loop.

After we see how easy it is to handle events, we'll look at ways ACE's Acceptor-Connector framework simplifies and enables implementation of services. The examples here are much easier, even, than the ones we saw in Chapter 6.