Learning interprocess communication with the ACE Framework

Arpan Sen

October 13, 2009

There are several problems with using the native Socket API. The Adaptive Communication Environment Framework defines a set of wrappers that address these problems. This article dissects some of the C++-based, object-oriented classes that ACE provides for IPC between the same or different host computers.

For most programmers, interprocess communication (IPC) is synonymous with using the Socket API. The Socket API was originally developed for the UNIX® platform to provide an application-level interface on top of the TCP/IP protocol. It supports a plethora of functions, some of which are shown in Table 1.

Frequently used acronyms

- **API:** Application programming interface
- **TCP/IP:** Transmission Control Protocol/Internet Protocol

Table 1. Legacy C routines that form part of the Socket API

C function name	Functionality provided
`socket`	Allocates a new socket handle
`bind`	Associates a socket handle to a local or remote address
`listen`	Routine to listen passively to incoming client connection requests
`connect`	The client machine uses the `connect` routine to begin the TCP handshake; the server uses the `accept` routine to accept the connection request
`send`	Routine to transmit data
`recv`	Routine to receive data

There are several problems with using the native Socket API. First, despite their nearly ubiquitous use, the API's legacy `C` functions are not portable. For example, the `socket` method in Windows® returns a handle of type `SOCKET`, while in UNIX, the same function returns an integer. Some methods—like `closesocket`—exist only in Windows, not to mention incompatible headers. Also, a lot of errors with the native API only appear at run time—for example, a mismatch in address or protocol or uninitialized data members.

The Adaptive Communication Environment (ACE) Framework defines a set of wrapper façades that address these problems. This article dissects some of the `C++`-based, object-oriented classes that ACE provides for IPC between the same or different host computers.

ACE classes for network programming

Table 2 shows some of the basic classes that ACE defines for TCP/IP connectivity.

Table 2. Classes in ACE for network programming

ACE class	Functionality provided
`ACE_Addr`	Base class in ACE; used for network addressing
`ACE_INET_Addr`	Derived from `ACE_Addr`; used for Internet domain addressing
`ACE_SOCK`	Base class for the ACE socket wrapper façade hierarchy: `ACE_SOCK_Acceptor`, `ACE_SOCK_Connector`, and so on
`ACE_SOCK_Acceptor`	Used for passive connection establishment; conceptually similar to Berkeley Software Distribution's (BSD's) `accept()` and `listen()` routines
`ACE_SOCK_Connector`	Establishes a connection between a stream object and a remote host; conceptually similar to the BSD `connect()` routine
`ACE_SOCK_Stream`	Class used for handling TCP connection-oriented data transfers

For User Datagram Protocol (UDP) communication, the `ACE_SOCK_Dgram` class and its variants are used. The header files declaring these objects are located in $ACE_ROOT/ace/include area in your installation. The relevant headers are appropriately named *SOCK_Acceptor.h, SOCK_Listener.h, SOCK_Stream.h,* and so on.

Creating a client-side connection

Let's start by dissecting the code in Listing 1.

Listing 1. ACE API to make connections from the client side

```
ACE_INET_Addr server(458, “tintin.cstg.in”);
ACE_SOCK_Stream client_stream;
ACE_SOCK_Connector client_connector;
if (-1 == connector.connect(client_stream, server)) {
  printf (“Failure to establish connection!\n”);
  return;
}
client_stream.send_n(“Hello World”, 12, 0);
client_stream.close( ); // close when done
```

The client needs two pieces of information: the host name and the port number to which the connection will be established. This detail is encapsulated in the `ACE_INET_Addr` class. There are other ways to initialize an `ACE_INET_Addr` class: `hostname: port no` and `hostip: port no` are popular choices. For example, you could declare the server as:

```
ACE_INET_Addr server ("tintin.cstg.in:458:)
```

or:

```
ACE_INET_Addr server ("132.132.2.61:458")
```

The next logical thing would be to establish a connection between the client data stream and the server, which is what the `ACE_SOCK_Connector` class is responsible for. Finally, the `ACE_SOCK_Stream` class does the actual message passing on the client's behalf. The `ACE_SOCK_Stream` class comes with `send_n` and `recv_n` methods meant to transfer data to and receive information from the remote machine. Note that the send and receive methods don't part-deliver messages: It's always done as a whole or not at all, in which case, the routines return **-1**.

Better error handling with ACE_SOCK_Connector

There are various reasons why the `connect` method could return **-1**. Maybe the server was too busy handling multiple clients and was therefore highly loaded; maybe it was just being rebooted and will start accepting requests shortly; or maybe the server runs on a slower operating system or hardware. Whatever the case, it makes good sense to keep trying for a while before giving up. The `connect` method accepts a timeout parameter, and if the connection does not succeed within the stipulated time period, it gives up. Listing 2 shows this process succinctly.

Listing 2. Client-side ACE API with timeout

```
…
ACE_SOCK_Connector client_connector;
ACE_Time_Value timeout(5);
if (-1 == connector.connect(client_stream, server, &timeout)) {
  printf (“Failure to establish connection!\n”);
  return;
}
…
```

Making a server-side connection

Listing 3 shows how the server side of the IPC works.

Listing 3. Server-side connection using the ACE API

```
ACE_INET_Addr server(458);
ACE_SOCK_Acceptor client_responder(server);
ACE_SOCK_Stream client_stream;
ACE_Time_Value timeout(5);
ACE_INET_Addr client;
if (-1 == client_responder.accept(client_stream, &client, &timeout)) {
    printf(“Connection not established with client!\n”);
    return;
} else {
    printf(“Client details: Host Name: %s Port Number: %d\n”,
         client.get_host_name(), client.get_port_number());
}
// Reading 40 bytes of data from the client
char buffer[128];
if (-1 == client_stream.recv_n(buffer, 40, 0)) {
   printf(“Error in reading client data!\n”);
   return;
} else {
   printf(“Client message: %s\n”, buffer);
}
client_stream.close();
```

At the server end, you need a port number on which the service should be run. Once again, an `ACE_INET_Addr` object is created with the port number (the host name is not required: by default, it's the current host). The `ACE_SOCK_Acceptor` class resembles the legacy BSD `listen` routine and is meant to accept client requests for connection to the server. You use `ACE_SOCK_Stream` in a similar manner: message passing to the client on the server's behalf. Observe the usage of the timeout in the call to the `accept` routine—`[client_responder.accept (client_stream, &client)]`: The server will block the call forever. However, using a timeout means that if no client connects to the server within the specified time interval, the server could print a message saying so and hang up its boots.

Finally, the `accept` method accepts a fourth argument not yet mentioned. Listing 4 shows the declaration of the `accept` method from ace/include/SOCK_Acceptor.h.

Listing 4. Declaration of the accept method

```
  /**
   * Accept a new ACE_SOCK_Stream connection using the QoS
   * information in @a qos_params.  A @a timeout of 0 means block
   * forever, a @a timeout of {0, 0} means poll.  @a restart == true means
   * "restart if interrupted," i.e., if errno == EINTR.  Note that
   * @a new_stream inherits the "blocking mode" of @c this
   * ACE_SOCK_Acceptor, i.e., if @c this acceptor factory is in
   * non-blocking mode, the @a new_stream will be in non-blocking mode
   * and vice versa.
   */
  int accept (ACE_SOCK_Stream &new_stream,
              ACE_Accept_QoS_Params qos_params,
              ACE_Addr *remote_addr = 0,
              ACE_Time_Value *timeout = 0,
              bool restart = true,
              bool reset_new_handle = false) const;
```

If the `restart` flag is set to True (the default), then an interrupt to the routine caused from some UNIX signal will cause a restart of the routine. You must decide whether this is a required feature in your application.

UDP-based IPC using ACE

Here's a quick recap of UDP. UDP is a datagram-oriented protocol, which means that if the client sends five 1KB chunks of information to the server, the server may receive anywhere between zero and five chunks. However, any chunk that the server receives will be a complete 1KB chunk—it's either received in full or not at all. UDP does not guarantee the arrival or the order in which the data arrives, so it's possible that the fourth chunk arrives before the second one. Also, datagrams may be lost in transit; however, UDP makes a best effort at delivery. Listing 5 defines a basic UDP-based client communication framework.

Listing 5. Client code for a UDP connection using ACE

```
ACE_INET_Addr server(458, “tintin.cstg.in”);
ACE_INET_Addr client(9000);
ACE_SOCK_Dgram client_data(client);

char* message = “Hello World!\n”;
size_t sent_data_length = client_data.send(message,
                                             strlen(message) + 1, server);
if (sent_data_length == -1)
  printf(“Error in data transmission\n”);

client_data.close();
```

Once again, you create an `ACE_INET_Addr` object for the server-port combination. You also need an `ACE_INET_Addr` object for the client (unlike for TCP) specifying the port that will send and receive the datagrams. The code for UDP does not need any connector or acceptor classes. You must create a class of type `ACE_SOCK_Dgram` and explicitly specify the server address during data transmission, which brings to light another interesting aspect of UDP style connections: The server address does not have to be unique. The same client can actually communicate with multiple remote peers. Although TCP-based connections are 1-to-1 connections, UDP has three different modes: *unicasting* (1-to-1 connection), *broadcasting* (datagram being sent to each remote peer in the network), and *multicasting* (datagram only sent to a subset of the total number of remote peers in the network).

Before delving into the topics of broadcast and multicast, however, look at a simple UDP-based server framework (see Listing 6).

Listing 6. Server code for a UDP connection using ACE

```
ACE_INET_Addr server(458);
ACE_SOCK_Dgram server_data(server);
ACE_INET_Addr client(“haddock.cstg.in”, 9000);

size_t sent_data_length = client_data.send(message,
                                             strlen(message) + 1, server);
if (sent_data_length == -1)
  printf(“Error in data transmission\n”);

server_data.close();
```

Unlike TCP, the client and server code look quite similar for UDP-based communication. Finally, note that the destructor for `ACE_SOCK_Dgram` will not automatically close the underlying UDP socket. An explicit call to the `close` method is a must; otherwise, object leaks result.

Connection-oriented datagram-based communication

For communication that always occurs between two peers, repeatedly providing the server address does not make for good programming practice. ACE provides the `ACE_SOCK_CODgram` class (where *COD* stands for *Connection-oriented Datagram*), which does away with the need for repeating the peer address. The peer address is provided once during the call to the `open` method. Subsequent calls to send are made with only two arguments: the character buffer and its length. Listing 7 shows the code.

Listing 7. Client code for a COD-based connection using ACE

```
ACE_INET_Addr server(458, "tintin.cstg.in");
ACE_INET_Addr client(9000);
ACE_SOCK_CODgram client_data(client);

if (0 != client_data.open(server)) {
   printf("Unable to establish connection with remote host\n");
   return;
}

char* message = "Hello World!\n";
size_t sent_data_length = client_data.send(message,
                                              strlen(message) + 1);
if (sent_data_length == -1)
  printf("Error in data transmission\n");

message = "Initiating";
client_data.send(message, strlen(message) + 1);

client_data.close();
```

Broadcasting using ACE

For broadcast `send` operations to multiple processes, you use the `ACE_SOCK_Dgram_Bcast` class (derived from `ACE_SOCK_Dgram`). The IP broadcast address does not need to be explicitly specified —the `ACE_SOCK_Dgram_Bcast` class takes care of supplying the correct IP address—you need only provide the appropriate remote host port number. The code in Listing 8 is similar to Listing 5, except that the `send` routine now only needs the port number to which it should broadcast.

Listing 8. Client code for broadcasting using ACE

```
ACE_INET_Addr local_host(4158, "tintin.cstg.in");
ACE_SOCK_Dgram_Bcast local_broadcast_dgram(local_host);
int remote_port = 7671;

char* message = "Hello World!\n";

size_t sent_data_length = local_broadcast_dgram.send(
                                         message, strlen(message) + 1, remote_port);

if (sent_data_length == -1)
  printf("Error in data transmission\n");

local_broadcast_dgram.close();
```

All remote machines listening on port 7671 should be able to handle the message using `ACE_SOCK_Dgram::recv`.

Multicasting using ACE

For multicast operations, you use the `ACE_SOCK_Dgram_Mcast` class (again derived from `ACE_SOCK_Dgram`). The `ACE_SOCK_Dgram_Mcast` class provides for sending and receiving messages to or from a specific group of peer computers. Peer computers interested in being a part of this multicast group must explicitly subscribe to the group. Listing 9 shows how to subscribe to or unsubscribe from multicast messages.

Listing 9. Client code that subscribes to receive multicast messages from the peer set

```
#define MULTICAST_ADDR “132.132.2.7”

ACE_INET_Addr multicast_addr(4158, MULTICAST_ADDR);
ACE_SOCK_Dgram_Mcast multicast_dgram;

// Subscribe
if (-1 == multicast_dgram.subscribe(multicast_addr)) {
   printf(“Error subscribing to multicast address\n”);
   return;
}

// Do Processing
…

// Unsubscribe
if (-1 == multicast_dgram.unsubscribe(multicast_addr)) {
   printf(“Error unsubscribing to multicast address\n”);
   return;
}
```

Peer computers that have subscribed to a multicast group receive all messages that any of the component peers sends to the group. However, if any host in the network wishes to send a message to this multicast group and is not interested in receiving a response, then using `ACE_SOCK_Dgram` for sending purposes is also sufficient. Listing 10 shows how a host uses multicast datagrams to receive and send messages. In a real-life application, the receive/send code would typically be part of some loop that keeps sending/receiving messages. Note the use of `ACE_INET_Addr` in the `recv` method: This helps capture the peer computer from which the data was being transmitted.

Listing 10. Client code that subscribes to sending/receiving multicast messages from the peer set

```
#define MULTICAST_ADDR “132.132.2.7”

ACE_INET_Addr multicast_addr(4158, MULTICAST_ADDR);
ACE_SOCK_Dgram_Mcast multicast_dgram;

//.. Subscribe
// Do Processing for a 256 byte message
char buffer[1024];
if (-1 == multicast_dgram.send(buffer, 256, multicast_addr)) {
  printf(“Failure in message transmission\n”);
}

ACE_INET_Addr peer_address;
if (-1 == multicast_dgram.recv(buffer, 256, peer_address)) {
  printf(“Failure in message reception\n”);
} else {
  printf(“Message %s received from Host %s : Port %d\n”,
    buffer, peer_address.get_host_name(),
    peer_address.get_port_number());
}
// .. Unsubscribe
```

Conclusion

This article showed how TCP/IP and UDP-based communication can be achieved using the ACE Framework. There are several other ways of initiating IPC using ACE, like shared memory or UNIX-style socket addressing (the `ACE_LSOCK*` group of classes), which this article does not address. For links to more advanced information, be sure to check out Related topics.

Related topics

- *The ACE Programmer's Guide: Practical Design Patterns for Network and Systems Programming* (Stephen D Huston, James CE Johnson, and Umar Syyid, Addison-Wesley Professional, 2003): Find practical information about programming in ACE.
- *C++ Network Programming, Volume I: Mastering Complexity with ACE and Patterns* (Douglas C. Schmidt and Stephen D Huston, Addison-Wesley Professional, 2001): Learn more about using patterns in ACE.
- ACE site: Check out the comprehensive source for all ACE documentation.
- ACE Framework overview: See an architectural overview of the ACE Framework.
- IBM product evaluation versions: Get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.