HUGHES NETWORK SYSTEMS

The Adaptive Communication Environment : "ACE"

A Tutorial

Umar Syyid
(usyyid@hns.com)
11717, Exploration Lane
Germantown MD, 20876.

Acknowledgments

I would like to thank the following people for their assistance in making this tutorial possible,

Ambreen Ilyas ambreen@bitsmart.com
James CE Johnson jcej@lads.com
Thomas Jordan ace@programmer.net
Erik Koerber erik.koerber@siemens.at
Martin Krumpolec krumpo@pobox.sk
Fred Kuhns fredk@tango.cs.wustl.edu
Susan Liebeskind shl@cc.gatech.edu
Douglas C. Schmidt schmidt@cs.wustl.edu
Aaron Valdivia avaldivia@hns.com

TABLE OF CONTENTS

Chapter

1

The Adaptive Communication Environment

An introduction

The Adaptive Communication Environment (ACE) is an object oriented framework and toolkit that implements core concurrency and distribution patterns for communication software. ACE includes several components to help in the development of communication software and to achieve better flexibility, efficiency, reliability and portability. Components in ACE can be used for the following purposes:

- Concurrency and Synchronization.
- Interprocess communication (IPC)
- Memory Management.
- Timers
- Signals
- File System management
- Thread Management
- Event demultiplexing and handler dispatching.
- Connection establishment and service initialization.
- Static and dynamic configuration and reconfiguration of software.
- Layered protocol construction and stream-based frameworks.
- Distributed communication services – naming, logging, time synchronization, event routing and network locking. etc.

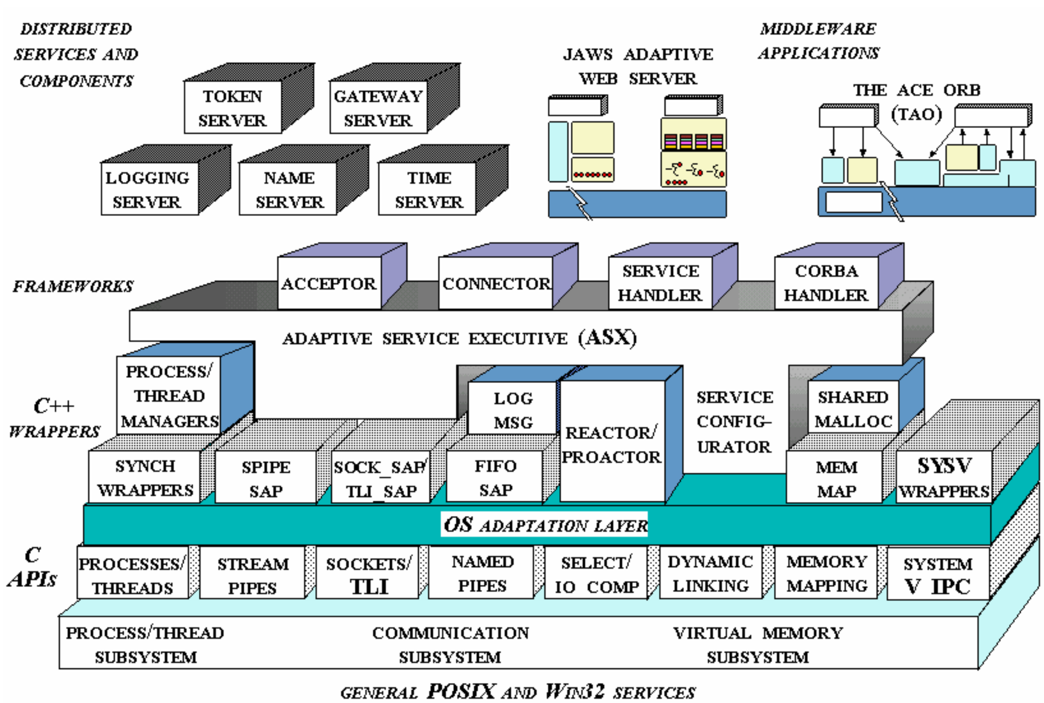
The ACE Architecture

ACE has a layered architecture as shown in the figure below. There are three basic layers in the ACE framework.

- The Operating Systems (OS)Adaptation Layer
- The C++ wrapper layer
- The Frameworks and patterns layer

The OS Adaptation Layer

The OS Adaptation is a thin layer of code that sits between the native OS APIs and ACE. The OS Adaptation layer therefore shields the higher layers of ACE from the platform and makes any code written by a developer using ACE platform independent. With a minimal amount of effort, an application developer can move his application to a different supported OS platform.



The OS adaptation layer is also the reason why the ACE framework is available on so many platforms. A few of the platforms on which ACE is currently available include real-time operating systems, (VxWorks, Chorus, LynxOS and pSoS) most versions of UNIX (SunOS 4.x and 5.x; SGI IRIX 5.x and 6.x; HP-UX 9.x, 10.x and 11.x; DEC UNIX 3.x and 4.x; AIX 3.x and 4.x; DG/UX; Linux; SCO; UnixWare; NetBSD and FreeBSD), Win32 (WinNT 3.5.x, 4.x, Win95 and WinCE using MSVC++ and Borland C++) and MVS OpenEdition.

The C++ wrappers layer

The C++ wrappers layer includes C++ wrapper classes which can be used to build highly portable and typesafe C++ applications. This is the largest part of the ACE toolkit and includes approximately 50% of the total source code. C++ wrapper classes are available for:

- **Concurrency and Synchronization** – ACE provides several concurrency and synchronization wrapper classes which abstract the native OS multi-threading and multi-processing API. These wrappers encapsulate primitives such as Semaphores, Locks, Barriers and Condition Variables for threads and for processes. Higher level primitives such as Guards are also available. All of these primitives share similar interfaces and thus are easy to use and substitute for one another.

- **IPC –** ACE provides several C++ wrapper classes which encapsulate different inter process communication (IPC) interfaces that are available on different OSs, For example, wrapper classes are provided around IPC mechanisms such as BSD Sockets, TLI, UNIX FIFOs, STREAM Pipes, Win32 Named Pipes. ACE also provides for Message Queue classes, and also includes wrapper classes for certain real-time OS-specific message queues.

- **Memory Management Components –** ACE includes classes to help in dynamic allocation and de-allocation of memory. This includes classes that allow for pre-allocation of all dynamic memory. This memory is then managed locally with the help of management classes provided in ACE. Such fine grain management is extremely necessary in most real-time and embedded systems. There are also classes to flexibly manage interprocess shared memory.

- **Timer Classes –** Various different classes are available to handle scheduling and canceling of timers. The different varieties of timers available in ACE use different underlying mechanisms (heaps, timer wheels or simple lists) to provide different performance characteristics. However, no matter what underlying mechanism is used, the interface to these classes remains the same, which makes it easy to use any of the timer classes. In addition to these timer classes, wrapper classes for high-resolution timers (which are available on some platforms, such as VxWorks, Win32/Pentium, AIX and Solaris) and Profile Timers are also available.

- **Container Classes –** ACE also includes several portable STL-type container classes, such as Map, Hash_Map, Set, List, etc.

- **Signal Handling –** ACE provides wrapper classes over the OS-specific signal handling interface. These classes allow the easy installation and removal of signal handlers and allow the installation of several handlers for one signal. Also available are signal guard classes that can be used to disable all signals in the scope of the guard.

- **Filesystem Components –** ACE also contains classes that wrap the filesystem API. These classes include wrappers for file I/O, asynchronous file I/O, file locking, file streams, file connection etc.
- **Thread Management –** ACE provides wrappers classes to create and manage threads. These wrappers also encapsulate the OS specific threading API and can be used to provide functionality such as thread-specific storage.

The ACE Framework Components

The ACE framework components are the highest level building blocks available in ACE. These arebased on several design patterns that have been recognized as being particularly useful in the communication software domain. A designer can use these components to help him think about the system at a higher level. These components are thus not only useful in the implementation stage of development, but also at the design stage, since they provide a mini-architecture for the system that is to be built. This layer contains the following large-scale components:

- **Event Handling –** Most communication software includes a large amount of code to handle the efficient de-multiplexing, dispatching and handling of various events which occur as the system is running. These events can be of various types, such as I/O based events, timer-based events, signal-based events and synchronization-based events. ACE includes a framework component called the **Reactor** to solve this problem. The Reactor provides code for efficient event de-multiplexing and dispatching.
- **Connection or Service Initialization Components** – ACE includes **Connector** and **Acceptor** components to decouple the initialization of a connection from the actual service that is to be performed by the application once the connection has been established. This is useful in application servers, which receive many connection requests that must be initialized and then handled by different handling routines. This decoupling allows the developer to worry about the handling and initialization as separate concerns. Thus, if the number of connection requests increases, he may chose to use a different set of policies for initialization (ACE includes several policies from which he can pick and choose) to achieve the required performance level.
- **Stream Components –** The ACE **Streams** component can be used to simplify the development of software that is layered or hierarchic in nature. A good example is the development of user-level protocol stacks that are composed of several interconnected layers, which can be developed more or less independently of each other. These layers then process and change "data" as it passes through them and pass it on to the next layer for further processing. Since each layer is independent of the other, they can be easily re-used or replaced.
- **Service Configuration components –** Another problem faced by communication software developers is that software services may have to be configured at installation time or reconfigured at run-time. The implementation of a certain

service in an application may require *change* and thus the application may have to be reconfigured with the new changed service. The ACE **Service Configurator** provides for dynamic start, suspend and configuration of services being provided by an application.

Although there have been rapid advances in the field of computer networks, the writing of communication software has become more difficult. Much of the effort expended on developing communication software is of the “re-inventing the wheel” variety, where components which are known to be common across applications are re-written rather then re-used. ACE provides a solution to this problem by collecting together common components and architecture which are known to be reused again and again in the network and systems programming domains. Thus an application developer can pick up ACE, pick and choose the components that he needs to use in his application and start building his application with the ACE tool box at his side. Besides capturing simple building blocks in the C++ wrappers layer, ACE also includes large architectural blocks which capture proven “patterns” and “software architecture” that is useful in this realm of software development.

Chapter

2

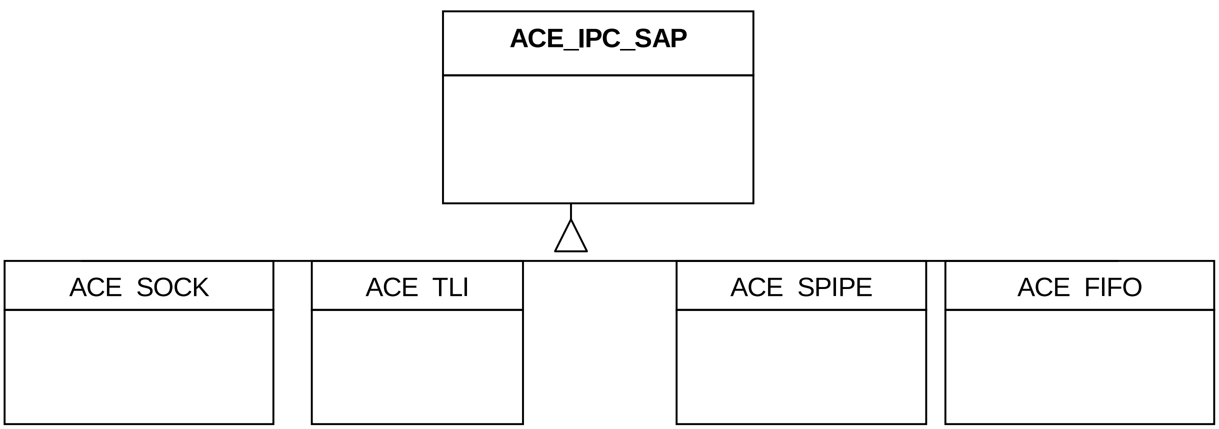
IPC SAP

Interprocess communication Service Access Point wrappers

Sockets, TLI, STREAM pipes and FIFOs provide a wide range of interfaces for accessing both local and global IPC mechanisms. However, there are many problems associated with these non-uniform interfaces. Problems such as lack of type safety and multiple dimensions of complexity lead to problematic and error-prone programming.

The IPC SAP class category in ACE provides a uniform hierarchic category of classes that encapsulate these tedious and error-prone interfaces. IPC is designed to improve the *correctness*, *ease of learning*, *portability* and *reusability* of communication software while maintaining high performance.

Categories of classes in IPC SAP



The IPC SAP classes are divided into four major categories, based on the different underlying IPC interface they are using. This division is illustrated by the class diagram above. The *ACE_IPC_SAP* class provides a few functions that are common to all IPC interfaces. From this class, four different classes are derived for each category of wrapper classes that ACE contains. These classes encapsulate functionality that is common to a particular interface. For example, the *ACE_SOCK* class contains functions that are common to the entire BSD sockets programming interface.

Underneath each of these four classes lie a whole hierarchy of wrapper classes which completely wrap the underlying interface and provide highly reusable, modular, safe and easy-to-use wrapper classes.

The Sockets Class Category (ACE_SOCK)

The classes in this category all lie under the *ACE_SOCK* class. This category provides an interface to the Internet domain and UNIX domain protocol families using the BSD sockets programming interface. The family of classes in this category can be further subdivided as:

- *Dgram* Classes and *Stream* Classes: The *Dgram* classes are based on the UDP datagram protocol and thus provide unreliable connectionless messaging functionality. The *Stream* Classes, on the other hand, are based on the TCP protocol and thus provide connection-oriented messaging.
- *Acceptor*, *Connector* Classes and *Stream* Classes: The *Acceptor* and *Connector* classes are used to passively and actively establish connections, respectively. The *Acceptor* classes encapsulates the BSD accept() call and the *Connector* encapsulates the BSD connect() call. The *Stream* classes are used AFTER a connection has been established to provide bi-directional data flow and encapsulate send and receive functions.

The Table below details the classes in this category and what their responsibilities are:

Class Name	Responsibility
`ACE_SOCK_Acceptor`	Used for passive connection establishment based on the BSD accept() and listen() calls.
`ACE_SOCK_Connector`	Used for active connection establishment based on the BSD connect() call.
`ACE_SOCK_Dgram`	Used to provide UDP (User Datagram Protocol) based connectionless messaging services. Encapsulates calls such as sendto() and receivefrom() and provides a simple send() and recv() interface.
`ACE_SOCK_IO`	Used to provide a connection-oriented messaging service. Encapsulates calls such as send(), recv() and write(). This class is the base class for the ACE_SOCK_Stream and ACE_SOCK_CODgram classes.
`ACE_SOCK_Stream`	Used to provide TCP (Transmission Control Protocol) -based connection-oriented messaging service. Derives from ACE_SOCK_IO and provides further wrapper methods.
`ACE_SOCK_CODgram`	Used to provide a connected datagram abstraction. Derives from ACE_SOCK_IO and includes an open() method, which causes a bind() to the local address specified and connects to the remote address using UDP.
`ACE_SOCK_Dgram_Mcast`	Used to provide a datagram-based multicast abstraction.

	Includes methods for subscribing to a multicast group as well as sending and receiving messages.
ACE_SOCK_Dgram_Bcast	Used to provide a datagram-based broadcast abstraction. Includes methods to broadcast datagram message to all interfaces in a subnet.

In the following sections, we will illustrate how the IPC_SAP wrappers can be used directly to handle interprocess communication. Remember that this is just the tip of the iceberg in ACE. All the good pattern-oriented tools come in later chapters of this tutorial.

Using Streams in ACE

The stream wrappers in ACE are used to provide connection-oriented communication. The Streams data transfer wrapper classes include *ACE_SOCK_Stream* and *ACE_LSOCK_Stream*, which wrap the TCP/IP and UNIX domain sockets protocols data transfer functionality, respectively. The Connection establishment classes include *ACE_SOCK_Connector and ACE_SOCK_Acceptor* for TCP/IP, and *ACE_LSOCK_Connector* and *ACE_LSOCK_Acceptor* for UNIX domain sockets.

The Acceptor class is used to passively accept connections (using the BSD *accept()* call) and the Connector class is used to actively establish connections (using the BSD *connect()* call).

The following examples should help illustrate how the acceptors and connectors are used to establish a connection, which is then used for data transfer using the stream classes.

Example 1

```
#include "ace/SOCK_Acceptor.h"
#include "ace/SOCK_Stream.h"
#define SIZE_DATA 18
#define SIZE_BUF 1024

class Server{

public:
Server (int port):
   server_addr_(port),peer_acceptor_(server_addr_){
   data_buf_= new char[SIZE_BUF];
   }

//Handle the connection once it has been established. Here the
//connection is handled by reading SIZE_DATA amount of data from the
//remote and then closing the connection stream down.
int handle_connection(){
    // Read data from client
   if(new_stream_.recv_n (data_buf_, SIZE_DATA, 0)==-1)
      ACE_ERROR ((LM_ERROR, "%p\n", "Error in recv"));


   ACE_DEBUG((LM_DEBUG,"Server recieved %s \n",data_buf_));
```

```
   // Close new endpoint
   if (new_stream_.close () == -1)
      ACE_ERROR ((LM_ERROR, "%p\n", "close"));
   return 0;
}
//Use the acceptor component peer_acceptor_ to accept the connection
//into the underlying stream new_stream_. After the connection has been
//established call the handle_connection() method.
int accept_connections (){
   if (peer_acceptor_.get_local_addr (server_addr_) == -1)
      ACE_ERROR_RETURN ((LM_ERROR,"%p\n","Error in get_local_addr"),1);

   ACE_DEBUG ((LM_DEBUG,"Starting server at port %d\n",
      server_addr_.get_port_number ()));


   // Performs the iterative server activities.
   while(1){
      ACE_Time_Value timeout (ACE_DEFAULT_TIMEOUT);
      if (peer_acceptor_.accept
            (new_stream_, &client_addr_, &timeout)== -1){
        ACE_ERROR ((LM_ERROR, "%p\n", "accept"));
        continue;
        }
      else
        ACE_DEBUG((LM_DEBUG,
        "Connection established with remote %s:%d\n",
        client_addr_.get_host_name(),client_addr_.get_port_number()));
     //Handle the connection
        handle_connection();
      }
   }



private:
   char *data_buf_;
   ACE_INET_Addr server_addr_;
   ACE_INET_Addr client_addr_;
   ACE_SOCK_Acceptor peer_acceptor_;
   ACE_SOCK_Stream new_stream_;
};

int main (int argc, char *argv[]){
   if(argc<2){
      ACE_ERROR((LM_ERROR,"Usage egX <port_num>"));
      ACE_OS::exit(1);
      }
   Server server(ACE_OS::atoi(argv[1]));
   server.accept_connections();
}
```

In the above example, a passive server is created which waits for incoming client connections. When a connection has been established, it calls the `handle_connection()` method to handle the incoming data on the newly created connection.

The *Server* class contains a method *accept_connections()* which uses an acceptor, i.e. *ACE_SOCK_Acceptor*, to accept the connection into the *ACE_SOCK_Stream* `new_stream_`. This is done by calling *accept()* on the acceptor and passing in the stream which we want it to accept the connection “nto. Once a connection has been established into a stream, then the stream wrappers *send()* and *recv()* methods can be used to send and receive data over the newly established link. The *accept()* method for the acceptor is also passed in a blank *ACE_INET_Addr*, which it sets to the address of the remote machine that had initiated the connection.

After the connection has been established, the server calls the *handle_connection()* method, which proceeds to read a pre-known word from the client and then closes down the stream. This is very non-realistic and what would probably happen in a real world situation is that the connection would be handled in either a separate thread or in a separate process. How such multi-threading and multi-process type handling is done will be illustrated in subsequent chapters.

The connection is closed down by calling the *close()* method on the stream. The method will release all socket resources and terminate the connection.

The next example illustrates how to use a Connector in conjunction with the Acceptor shown in the previous example.

Example 2

```
#include "ace/SOCK_Connector.h"
#include "ace/INET_Addr.h"
#define SIZE_BUF 128
#define NO_ITERATIONS 5

class Client{
public:
Client(char *hostname, int port):remote_addr_(hostname){
   remote_addr_.set_port_number(port);
   data_buf_=new char[SIZE_BUF];
   }

//Uses a connector component connector_ to connect to a remote machine
//and pass the connection into a stream component client_stream_
int connect_to_server(){
  // Initiate blocking connection with server.
  ACE_DEBUG ((LM_DEBUG, "(%P|%t) Starting connect to %s:%d\n",
      remote_addr_.get_host_name(),remote_addr_.get_port_number()));
  if (connector_.connect (client_stream_, remote_addr_) == -1)
    ACE_ERROR_RETURN ((LM_ERROR,"(%P|%t) %p\n","connection failed"),-1);
  else
    ACE_DEBUG ((LM_DEBUG,"(%P|%t) connected to %s\n",
       remote_addr_.get_host_name ()));
  return 0;
      }
```

```
//Uses a stream component to send data to the remote host.
int send_to_server(){
  // Send data to server
 ACE_OS::sprintf(data_buf_,"Hello from Client");
 for(int i=0;i<NO_ITERATIONS; i++){
    if (client_stream_.send_n
            (data_buf_, ACE_OS::strlen(data_buf_), 0) == -1){
      ACE_ERROR_RETURN ((LM_ERROR,"(%P|%t) %p\n","send_n"),0);
      break;
      }
   }
   //Close down the connection
   close();
}

//Close down the connection properly.
int close(){
   if (client_stream_.close () == -1)
      ACE_ERROR_RETURN ((LM_ERROR,"(%P|%t) %p\n","close"),-1);
   else
      return 0;
      }

private:
   ACE_SOCK_Stream client_stream_;
   ACE_INET_Addr remote_addr_;
   ACE_SOCK_Connector connector_;
   char *data_buf_;
};

int main (int argc, char *argv[]){
   if(argc<3){
      ACE_DEBUG((LM_DEBUG,"Usage egX <hostname> <port_number>\n"));
      ACE_OS::exit(1);
      }
   Client client(argv[1],ACE_OS::atoi(argv[2]));
   client.connect_to_server();
   client.send_to_server();
}
```

The above example illustrates a client that actively connects to the server which was described in Example 1. After establishing a connection, it sends a single string of data to the server `NO_ITERATIONS` times and closes down the connection. The example consists of a single *Client* class which contains a *connect_to_server()* and a *send_to_server()* method.

The *connect_to_server()* method uses a Connector, i.e. *ACE_SOCK_Connector*, to actively establish a connection to the remote machine specified in the runtime arguments to the application. This is done by calling the *connect()* method on the Connector `connector_` , passing it the `remote_addr_` of the machine we wish to connect to and an empty *ACE_SOCK_Stream* to establish the connection into. Once the connect method returns successfully, this stream can be used to send and receive data over the newly established link. This is accomplished by using the *send()* and *recv()* family of methods available in the *ACE_SOCK_Stream* wrapper class.

In the example, once the connection has been established, the *send_to_server()* method is called to send a single string to the sever `NO_ITERATIONS` times. As mentioned before, this is done by using the send() methods of the stream wrapper class.

Using Datagrams in ACE

The Datagrams wrapper classes in ACE are *ACE_SOCK_Dgram* and *ACE_LSOCK_Dgram*. These wrappers include methods to send and receive datagrams and wrap the non-connection-oriented UDP protocol and UNIX domain sockets protocol. Unlike the Streams wrapper, since these wrappers wrap a non-connection-oriented protocol, there is no concept of having acceptors and connectors to "set up" a connection. Instead, in this case, communication will be through a series of sends and receives, where the sends will indicate which remote address the data is to be sent to. The following example will help to illustrate how datagrams can be used in ACE. The examples use *ACE_SOCK_Dgram* (i.e., the UDP wrappers). The *ACE_LSOCK_Dgram* is used for UNIX domain datagrams. The usage of both wrappers is very similar, the only difference being that local datagrams would use the *ACE_UNIX_Addr* class for addresses instead of *ACE_INET_Addr*.

Example 3

```
//Server
#include "ace/OS.h"
#include "ace/SOCK_Dgram.h"
#include "ace/INET_Addr.h"

#define DATA_BUFFER_SIZE 1024
#define SIZE_DATA 18

class Server{
public:
Server(int local_port)
   :local_addr_(local_port),local_(local_addr_){
    data_buf = new char[DATA_BUFFER_SIZE];
    }
//Expect data to arrive from the remote machine. Accept it and display
//it. After receiving data immediately send some data back to the
//remote.
int accept_data(){
   while(local_.recv(data_buf,SIZE_DATA,remote_addr_)!=-1){
      ACE_DEBUG((LM_DEBUG, "Data received from remote %s was %s \n"
                  ,remote_addr_.get_host_name(), data_buf));
      ACE_OS::sleep(1);
      if(send_data()==-1) break;
      }
   return -1;
   }
```

```
//Method used to send data to the remote using the datagram component
//local_
int send_data(){
   ACE_DEBUG((LM_DEBUG,"Preparing to send reply to client %s:%d\n",
      remote_addr_.get_host_name(),remote_addr_.get_port_number()));
   ACE_OS::sprintf(data_buf,"Server says hello to you too");
   if(
   local_.send(data_buf, ACE_OS::strlen(data_buf),remote_addr_)==-1)
      return -1;
   else
      return 0;
   }

private:
   char *data_buf;
   ACE_INET_Addr remote_addr_;
   ACE_INET_Addr local_addr_;
   ACE_SOCK_Dgram local_;
};

int main(int argc, char *argv[]){
   Server server(ACE_OS::atoi(argv[1]));
   server.accept_data();
}
```

The above code is for a simple server that expects a client application to send it a datagram on a known port, with a fixed amount of data in it. The server, on reception of this data, then proceeds to send a reply back to the client that originally sent the data.

The single class *Server* contains an *ACE_SOCK_Dgram* named `local_` as a private member which it uses both to receive and send data. The *Server* instantiates `local_` in its constructor, with a known *ACE_INET_Addr* (local host with known port) so the client can locate it and send messages.

The class contains two methods: *accept_data()*,used to receive data from the client (uses the `local_` wrappers *recv()* call) and *send_data()* (uses the `local_` wrappers *send()* call) used to send data to the remote client. Notice that the underlying calls for both the *send()* and *receive()*of the `local_` wrapper class wrap the BSD *sendto()* and *recvfrom()* calls and have a similar interface.

The main function just instantiates an object of type server and calls the *accept_data()* method on it which waits for data from the client. When it gets the data it is expecting it calls *send_data()* to send a reply message back to the client. This goes on forever until the client dies.

The corresponding client code is very similar:

Example 4

```
//Client
#include "ace/OS.h"
#include "ace/SOCK_Dgram.h"
```

```
#include "ace/INET_Addr.h"

#define DATA_BUFFER_SIZE 1024
#define SIZE_DATA 28
class Client{
public:
Client(char * remote_addr,int port)
   :remote_addr_(remote_addr),
    local_addr_((u_short)0),local_(local_addr_){
       data_buf = new char[DATA_BUFFER_SIZE];
    remote_addr_.set_port_number(port);
    }

//Accept data from the remote host using the datgram component local_
int accept_data(){
   if(local_.recv(data_buf,SIZE_DATA,remote_addr_)!=-1){
      ACE_DEBUG((LM_DEBUG, "Data received from remote server %s
                  was: %s \n" ,remote_addr_.get_host_name(), data_buf));
      return 0;
      }
   else
      return -1;
   }
//Send data to the remote. Once data has been sent wait for a reply from
//the server.
int send_data(){
   ACE_DEBUG((LM_DEBUG,"Preparing to send data to server %s:%d\n",
       remote_addr_.get_host_name(),remote_addr_.get_port_number()));
   ACE_OS::sprintf(data_buf,"Client says hello");

   while(local_.send
            (data_buf,ACE_OS::strlen(data_buf),remote_addr_)!=-1){
      ACE_OS::sleep(1);
      if(accept_data()==-1)
        break;
      }
      return -1;
      }

private:
   char *data_buf;
   ACE_INET_Addr remote_addr_;
   ACE_INET_Addr local_addr_;
   ACE_SOCK_Dgram local_;
};

int main(int argc, char *argv[]){
if(argc<3){
   ACE_OS::printf("Usage: Client hostname port_number \n");
   ACE_OS::exit(1);

   }
Client client(argv[1],ACE_OS::atoi(argv[2]));
client.send_data();
```

```
}
```

Using Multicast with ACE

ACE provides for an unreliable multicast wrapper *ACE_SOCK_Dgram_Mcast* to send datagram messages to a multicast group address.

The intent of multicasting is to address the problem of broadcasting, not to all terminals, but to a certain subset or group of terminals. Thus, it can be thought of as a controlled broadcast.

This is achieved by using multicast addresses. Each terminal that is interested in receiving these controlled broadcasts must subscribe to the multicast address (also called subscribing to the multicast group). Any datagram message sent to the group will then be received by all terminals that have subscribed to the multicast group. An application that only wishes to send messages to the multicast group does not have to subscribe to the multicast group. In fact, such a sender may just use the *ACE_SOCK_Dgram* wrapper to send messages to the multicast address. This message will than be received by the entire multicast group.

In ACE, the multicast functionality is encapsulated in *ACE_SOCK_Dgram_Mcast.* This includes functions to subscribe, unsubscribe and receive on the multicast group.

The following examples illustrate how multicasting can be used in ACE.

Example 5

```
#include "ace/SOCK_Dgram_Mcast.h"
#include "ace/OS.h"
#define DEFAULT_MULTICAST_ADDR "224.9.9.2"
#define TIMEOUT 5

class Receiver_Multicast{

public:
Reciever_Multicast(int port):
   mcast_addr_(port,DEFAULT_MULTICAST_ADDR),remote_addr_((u_short)0){
  // Subscribe to multicast address.
  if (mcast_dgram_.subscribe (mcast_addr_) == -1){
   ACE_DEBUG((LM_DEBUG,"Error in subscribing to Multicast address \n"));
   exit(-1);
   }
}

~Receiver_Multicast(){
   if(mcast_dgram_.unsubscribe()==-1)
      ACE_DEBUG((LM_ERROR,”Error in unsubscribing from Mcast group\n”));
   }
```

```
//Receive data from someone who is sending data on the multicast group
//address. To do so it must use the multicast datagram component
//mcast_dgram_.
int recv_multicast(){
   //get ready to receive data from the sender.
   if(mcast_dgram_.recv
            (&mcast_info,sizeof (mcast_info),remote_addr_)==-1)
      return -1;
   ACE_DEBUG ((LM_DEBUG, "(%P|%t) Received multicast from %s:%d.\n",
   remote_addr_.get_host_name(), remote_addr_.get_port_number()));
   ACE_DEBUG((LM_DEBUG,"Successfully received %d\n", mcast_info));
   return 0;
   }

private:
   ACE_INET_Addr mcast_addr_;
   ACE_INET_Addr remote_addr_;
   ACE_SOCK_Dgram_Mcast mcast_dgram_;
   int mcast_info;
};


int main(int argc, char*argv[]){
   Reciever_Multicast m(2000);
   //Will run forever
   while(m.recv_multicast()!=-1) {
      ACE_DEBUG((LM_DEBUG,"Multicaster successful \n"));
   }

   ACE_DEBUG((LM_ERROR,"Multicaster failed \n"));
   exit(-1);
}
```

The above example shows how an application can use *ACE_SOCK_Dgram_Mcast* to subscribe to and receive messages from a multicast group.

The constructor of the *Reciever_Multicast* class subscribes the object to the multicast group and the destructor of the object unsubscribes. Once subscribed, the application waits forever for any data that is sent to the multicast address.

The next example shows how an application can send datagram messages to the multicast address or group using the simple *ACE_SOCK_Dgram* wrapper class.

Example 6

```
#define DEFAULT_MULTICAST_ADDR "224.9.9.2"
#define TIMEOUT 5
#include "ace/SOCK_Dgram_Mcast.h"
#include "ace/OS.h"

class Sender_Multicast{
public:
Sender_Multicast(int port):
```

```
    local_addr_((u_short)0),dgram_(local_addr_),
   multicast_addr_(port,DEFAULT_MULTICAST_ADDR){}

//Method which uses a simple datagram component to send data to the
//multicast group.
int send_to_multicast_group(){
   //Convert the information we wish to send into network byte order
   mcast_info= htons (1000);

   // Send multicast
   if(dgram_.send
            (&mcast_info, sizeof (mcast_info), multicast_addr_)==-1)
      return -1;

   ACE_DEBUG ((LM_DEBUG,
              "%s; Sent multicast to group.  Number sent is %d.\n",
              __FILE__,
              mcast_info));
  return 0;
}

private:
   ACE_INET_Addr multicast_addr_;
   ACE_INET_Addr local_addr_;
   ACE_SOCK_Dgram dgram_;
   int mcast_info;
};


int main(int argc, char*argv[]){
Sender_Multicast m(2000);
if(m.send_to_multicast_group()==-1) {
   ACE_DEBUG((LM_ERROR,"Send to Multicast group failed \n"));
   exit(-1);
   }
else
   ACE_DEBUG((LM_DEBUG,"Send to Multicast group successful \n"));
}
```

In this example, the client application creates a datagram component that is used to send data to the multicast group.

The Sender_Multicast object contains a simple `send_to_multicast_group()` method, which uses the datagram component `dgram_` to send messages to the multicast group.

Chapter

3

Memory Management

An introduction to Memory Management in ACE

ACE contains two sets of classes which are used to manage local and shared memory. The first set are those classes which are based on *ACE_Allocator* and use dynamic binding to provide for flexibility and extensibility. These can only be used to provide for local dynamic memory allocation. The second set of classes is based on the *ACE_Malloc* template class. This set uses templates to provide for flexibility in memory allocation mechanisms. The classes in this set not only include classes for local dynamic memory management, but also include classes to manage shared memory between processes, using the operating systems (OS) shared memory interface.

The tradeoff between these two sets is efficiency and flexibility. The *ACE_Allocator* classes are more flexible as the actual allocation object and thus the mechanism that is used can be changed at runtime using dynamic binding. This flexibility, however, does not come without a cost. The indirection of virtual function calls makes this alternative the more expensive option. The *ACE_Malloc* classes are configured at compile time with the memory allocation mechanism that they use. Thus, although *ACE_Malloc* is more efficient, it is not as flexible as *ACE_Allocator.*

Allocators

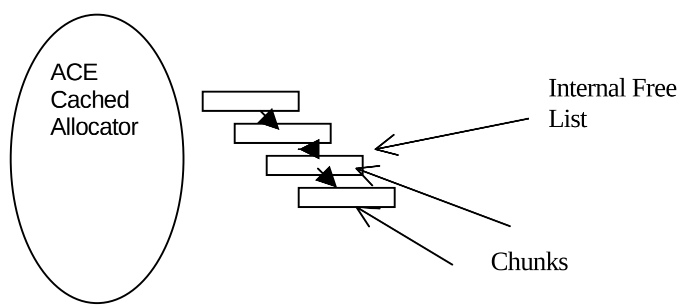
Allocators are used in ACE to provide a dynamic memory management mechanism. Several Allocators are available in ACE which work using different policies, although they provide for the same type of functionality. For example, in real-time systems it may be necessary for an application to pre-allocate all the dynamic memory it will need from the OS and then control the allocation and de-allocation internally, so that the performance for these routines is predictable.

All Allocators support the *ACE_Allocator* interface and therefore can be easily replaced with one another, either at runtime or compile time. The Table below gives a brief description of the different allocators that are available in ACE. It also describes the kind of policies they use for memory allocation.

Allocator	Description
`ACE_Allocator`	Interface class for the set of Allocator classes in ACE. These classes use inheritance and dynamic binding to provide flexibility.
`ACE_Static_Allocator`	This Allocator manages a fixed size of memory. Every time a request is received for more memory, it moves an internal pointer to return the chunk. This allocator assumes that memory, once allocated, will never be freed.
`ACE_Cached_Allocator`	This Allocator preallocates pool of memory that contains a specific number of size-specified chunks. These chunks are maintained on an internal free list and returned when a memory request (*malloc()*) is received. The chunks can be returned to the internal free list (When applications call *free()*, the chunk is returned back to the free list).
`ACE_New_Allocator`	An allocator which provides a wrapper over the C++ new and delete operators, i.e. it internally uses the new and delete operators to satisfy dynamic memory requests.

Using the Cached Allocator

As mentioned in the table above, the *ACE_Cached_Allocator* preallocates memory which it manages itself on a free list. This kind of mechanism can be very useful, especially in real-time systems in which memory allocation must be predictable.



The Cached Allocator is illustrated in the diagram above. The free list maintains several "chunks" of memory on a free list. These chunks may be any complex data type. In the case of memory used for general allocation in a real-time system, internal fragmentation of chunks is a concern.

The following example illustrates how the *ACE_Cached_Allocator* can be used to pre-allocate memory and then handle requests for memory.

```
Example 1
#include "ace/Malloc.h"
//A chunk of size 1K is created
```

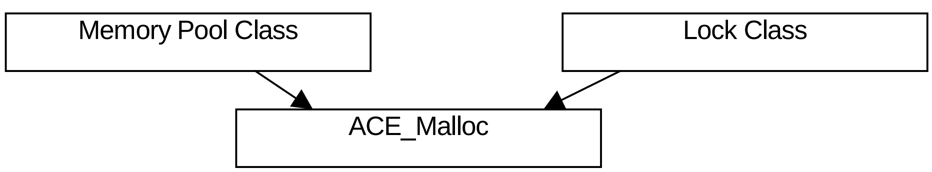
```
typedef char MEMORY_BLOCK[1024];



//Create an ACE_Cached_Allocator which is passed in the type of the
//“chunk” that it must pre-allocate and assign on the free list
typedef ACE_Cached_Allocator<MEMORY_BLOCK,ACE_SYNCH_MUTEX> Allocator;

class MessageManager{
public:
//The constructor is passed the number of chunks that the allocator
//should pre-allocate and maintain on its free list.
MessageManager(int n_blocks):
   allocator_(n_blocks),message_count_(0){}

//Allocate memory for a message using the Allocator
void allocate_msg(const char *msg){
   mesg_array_[message_count_]=
            allocator_.malloc(ACE_OS::strlen(msg)+1);
   ACE_OS::strcpy(mesg_array_[message_count_],msg);
   message_count_++;
   }

//Free all memory allocated. This will cause the chunks to be returned
//to the allocator’s internal free list and NOT to the OS.
void free_all_msg(){
   for(int i=0;i<message_count_;i++)
      allocator_.free(mesg_array_[i]);
   message_count_=0;
   }
void display_all_msg(){
   for(int i=0;i<message_count_;i++)
      ACE_OS::printf("%s\n",mesg_array_[i]);
   }

private:
   char *mesg_array_[20];
   Allocator allocator_;
   int message_count_;
};


int main(int argc, char* argv[]){

if(argc<2){
   ACE_OS::printf("Usage: egXX <Number of blocks>\n");
   exit(1);
   }

//Instantiate the Memory Manager class
int n_blocks=ACE_OS::atoi(argv[1]);
MessageManager mm(n_blocks);
```

```
//Use the Memory Manager class to assign messages and free them.
//Run this in your debug environment and you will notice that the
//amount of memory your program uses after Memory Manager has been
//instantiated remains the same. That means the Cached Allocator
//controls or manages all the memory for the application.

//Do forever.
while(1){
   //allocate the messages somewhere
   for(int i=0; i<n_blocks;i++)
      mm.allocate_msg("Hi there");
   //show the messages
   mm.display_all_msg();
   //free up the memory for the messages.
   mm.free_all_msg();
   }
return 0;
}
```

This simple example contains a message manager class which instantiates a cached allocator. This allocator is then used to allocate, display and free messages forever. The memory usage of the application, however, does not change. You can check this out with the debugging tool of your choice.

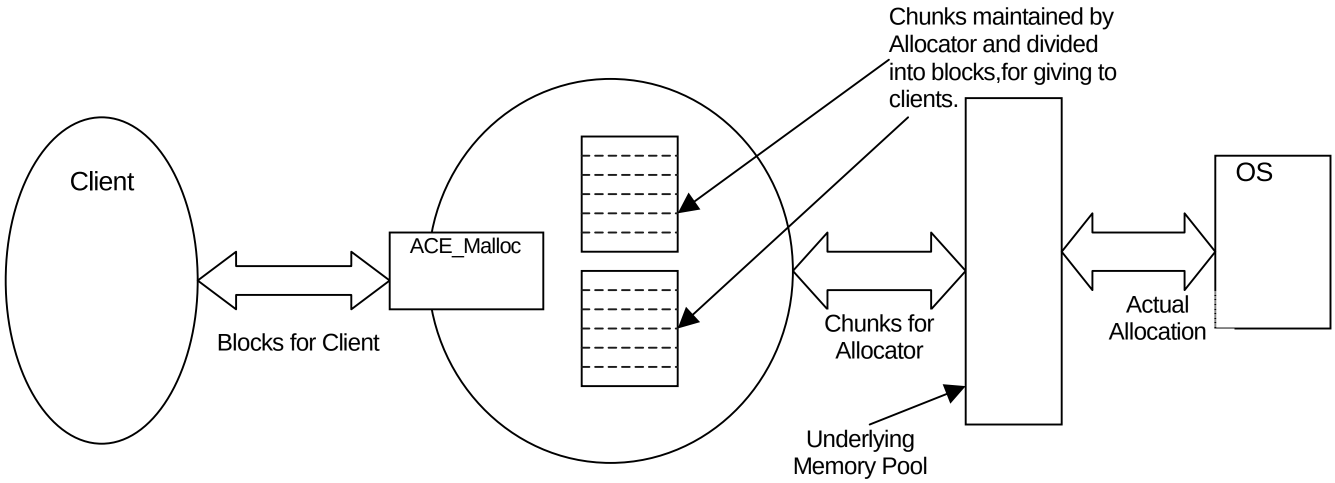
ACE_Malloc

As mentioned earlier, the Malloc set of classes use the template class *ACE_Malloc* to provide for memory management. The ACE_Malloc template takes two arguments, a memory pool and a lock for the pool, which gives us our allocator class, as is shown in the figure below.



How ACE_Malloc works

The idea here is that the *ACE_Malloc* class will "acquire" memory from the memory pool that was passed in and the application then "malloc()s" memory using the *ACE_Malloc* classes interface. The memory returned by the underlying memory pool is returned internally to the *ACE_Malloc* class in what are known in ACE as "chunks". The *ACE_Malloc* class uses these chunks of memory to allocate smaller "blocks" of memory to an application developer. This is illustrated in the diagram below.



When an application requests a block of memory, the *ACE_Malloc* class will check if there is enough space to allocate the block from one of the chunks it has already acquired from the memory pool. If it cannot find a chunk with enough space on it, then it asks the underlying memory pool to return a larger chunk, so it can satisfy the application's request for a block of memory. When an application issues a *free()* call, *ACE_Malloc* will not return the memory that was freed back to the memory pool, but will maintain it on its free list. When ACE_Malloc receives subsequent requests for memory, it will use this free list to search for empty blocks that can be returned. Thus, when the *ACE_Malloc* class is used, the amount of memory allocated from the OS will only go up and not down, if simple *malloc()* and *free()* calls are the only calls issued. The *ACE_Malloc* class also includes a *remove()* method, which issues a request to the memory pool for it to return the memory to the OS. This method also returns the lock to the OS.

Using ACE_Malloc

Using the *ACE_Malloc* class is simple. First, instantiate *ACE_Malloc* with a memory pool and locking mechanism of your choice, to create an allocator class. This allocator class is subsequently used to instantiate an object, which is the allocator your application will use. When you instantiate an allocator object, the first parameter to the constructor is a string, which is the "name" of the underlying memory pool you want the allocator object to use. It is **VERY** important that the correct name is passed in to the constructor, especially if you are using shared memory. Otherwise the allocator will create a new memory pool for you. This, of course, is not what you want if you are using a shared memory pool, since then you get no sharing.

To facilitate the sharing of the underlying memory pool (again, if you are using shared memory this is important), the *ACE_Malloc* class also includes a map type interface. Each block of memory that is allocated can be given a name, and can thus be easily found by another process looking through the memory pool. This includes *bind()* and *find()*

calls. The *bind()* call is used to give names to the blocks that are returned by the *malloc()* call to *ACE_Malloc*. The *find()* call, as you probably expect, is then used to find the memory previously associated with the name.

There are several different memory pool classes that are available (shown in table below) to be used when instantiating the *ACE_Malloc* template class. These pools cannot only be used to allocate memory that are used within a process, but can also be used to allocate memory pools that are shared between processes. This also makes it clearer why the *ACE_Malloc* template needs to be instantiated with a locking mechanism. The lock ensures that when multiple processes access the shared memory pool, it doesn't get corrupted. Note that even when multiple threads are using the allocator, it will be necessary to provide a locking mechanism.

The different memory pools available are listed in the table below:

Name of Pool	Macro	Description
`ACE_MMAP_ Memory_Pool`	`ACE_MMAP_ MEMORY_POOL`	Uses the <*mmap(2)*> to create the pool. Thus memory can be shared between processes. Memory is updated to the backing store on every update.
`ACE_Lite_MMAP_ Memory_Pool`	`ACE_LITE_MMAP_MEMORY_POOL`	Uses the <*mmap(2)*> to create the pool. Unlike the previous map, this does not update to the backing store. The tradeoff is lowered reliability.
`ACE_Sbrk_ Memory_Pool`	`ACE_SBRK_ MEMORY_POOL`	Uses the <*sbrk(2)*> call to create the pool.
`ACE_Shared_ Memory_Pool`	`ACE_SHARED_ MEMORY_POOL`	Uses the System V <*shmget(2)*> call to create the memory pool. Memory can be shared between processes.
`ACE_Local_ Memory_Pool`	`ACE__LOCAL_ MEMORY_POOL`	Creates a local memory pool based on the C++ *new* and *delete* operators. This pool can't be shared between processes.

The following example uses the *ACE_Malloc* class with a shared memory pool (the example shows it using *ACE_SHARED_MEMORY_POOL*, but any memory pool, from the table above, that supports shared memory may be used).

It creates a server process, which creates a memory pool and then allocates memory from the pool. The server then creates messages it wants the client process to "pick up" using the memory it allocated from the pool. Next, it *bind*s names to these messages so that the client can use the corresponding *find* operation to find the messages the server inserted into the pool.

The client starts up and creates its own allocator, but uses the *SAME* memory pool. This is done by passing the same **name** to the constructor for the allocator, after which it uses the *find()* call to find the messages inserted by the server and print them out for the user to see.

Example 2

```
#include "ace/Shared_Memory_MM.h"
#include "ace/Malloc.h"
#include "ace/Malloc_T.h"
#define DATA_SIZE 100
#define MESSAGE1 "Hiya over there client process"
#define MESSAGE2  "Did you hear me the first time?"
LPCTSTR poolname="My_Pool";

typedef ACE_Malloc<ACE_SHARED_MEMORY_POOL,ACE_Null_Mutex>
Malloc_Allocator;

static void
server (void){
   //Create the memory allocator passing it the shared memory
   //pool that you want to use
   Malloc_Allocator shm_allocator(poolname);

   //Create a message, allocate memory for it and bind it with
   //a name so that the client can the find it in the memory
   //pool
   char* Message1=(char*)shm_allocator.malloc(strlen(MESSAGE1));
   ACE_OS::strcpy(Message1,MESSAGE1);
   shm_allocator.bind("FirstMessage",Message1);
   ACE_DEBUG((LM_DEBUG,"<<%s\n",Message1));

   //How about a second message
   char* Message2=(char*)shm_allocator.malloc(strlen(MESSAGE2));
   ACE_OS::strcpy(Message2,MESSAGE2);
   shm_allocator.bind("SecondMessage",Message2);
   ACE_DEBUG((LM_DEBUG,"<<%s\n",Message2));

   //Set the Server to go to sleep for a while so that the client has
   //a chance to do its stuff
   ACE_DEBUG((LM_DEBUG,
            "Server done writing.. going to sleep zzz..\n\n\n"));
   ACE_OS::sleep(2);

   //Get rid of all resources allocated by the server. In other
   //words get rid of the shared memory pool that had been
   //previously allocated
   shm_allocator.remove();

}
```

```
static void
client(void){
   //Create a memory allocator. Be sure that the client passes
   // in the "right" name here so that both the client and the
   //server use the same memory pool. We wouldn't want them to
   // BOTH create different underlying pools.
   Malloc_Allocator shm_allocator(poolname);

   //Get that first message. Notice that the find is looking up the
   //memory based on the "name" that was bound to it by the server.
   void *Message1;
   if(shm_allocator.find("FirstMessage",Message1)==-1){
      ACE_ERROR((LM_ERROR,
            "Client: Problem cant find data that server has sent\n"));
      ACE_OS::exit(1);
      }
   ACE_OS::printf(">>%s\n",(char*) Message1);
   ACE_OS::fflush(stdout);

   //Lets get that second message now.
   void *Message2;
   if(shm_allocator.find("SecondMessage",Message2)==-1){
      ACE_ERROR((LM_ERROR,
            "Client: Problem cant find data that server has sent\n"));
      ACE_OS::exit(1);
      }
   ACE_OS::printf(">>%s\n",(char*)Message2);
   ACE_OS::fflush(stdout);

   ACE_DEBUG((LM_DEBUG,"Client done reading! BYE NOW\n"));
   ACE_OS::fflush(stdout);
}


int main (int, char *[]){
switch (ACE_OS::fork ())
    {
    case -1:
      ACE_ERROR_RETURN ((LM_ERROR, "%p\n", "fork"), 1);
    case 0:
      // Make sure the server starts up first.
      ACE_OS::sleep (1);
      client ();
      break;
    default:
      server ();
        break;
   }
  return 0;
}
```

Using the Malloc classes with the Allocator interface

Most container classes in ACE allow for an Allocator object to be passed in for managing memory used in the container. Since certain memory allocation schemes are only available with the *ACE_Malloc* set of classes, ACE includes an adapter template class *ACE_Allocator_Adapter*, which adapts the *ACE_Malloc* class to the *ACE_Allocator* interface. What this means is that the new class created after instantiating the template can be used in place of any *ACE_Allocator*. For example:

```
typedef
ACE_Allocator_Adapter<ACE_Malloc<ACE_SHARED_MEMORY_POOL,ACE_Null_Mutex>>
Allocator;
```

Thus this newly created *Allocator* class can be used wherever the Allocator interface is required, but it will use the underlying functionality of *ACE_Malloc* with a *ACE_Shared_Memory_Pool*. Thus the adapter "adapts" the Malloc class to the Allocator class.

This allows one to use the functionality associated with the *ACE_Malloc* set of classes with the dynamic binding flexibility available with *ACE_Allocator*. It is, however, important to remember that this flexibility comes at the price of performance.

Chapter

4

Thread Management

Synchronization and thread management mechanisms in ACE

ACE contains many different classes for creating and managing multi-threaded programs. In this chapter, we will go over a few of the mechanisms that are available in ACE to provide for thread management. In the beginning, we will go over the simple thread wrapper classes, which contain minimal management functionality. As the chapter progresses, however, we will go over the more powerful management mechanisms available in *ACE_Thread_Manager*. ACE also contains a very comprehensive set of classes that deal with synchronization of threads. These classes will also be covered here.

Creating and canceling threads

There are several different interfaces that are available for thread management on different platforms. These include the POSIX pthreads interface, Solaris threads, Win32 threads, VxWorks threads, pSoS threads, etc. Each of these interfaces provides the same or similar functionality but with APIs that are vastly different. This leads to difficult, tedious and error-prone programming, since the application programmer must make himself familiar with several interfaces to write on different platforms. Furthermore, such programs, once written, are non-portable and inflexible.

ACE_Thread provides a simple wrapper around the OS thread calls that deal with issues such as creation, suspension, cancellation and deletion of threads. This gives the application programmer a simple and easy-to-use interface which is portable across different threading APIs. *ACE_Thread* is a very thin wrapper, with minimal overhead. Most methods are inlined and thus are equivalent to a direct call to the underlying OS-specific threads interface. All methods in *ACE_Thread* are static and the class is not meant to be instantiated.

The following example illustrates how the *ACE_Thread* wrapper class can be used to create, yield and join with threads.

Example 1

```
#include "ace/Thread.h"
#include "ace/OS.h"
#include "ace/Synch_T.h"
#include "ace/Synch.h"
static int number=0;
static int seed=0;

static void*
worker(void *arg){
   ACE_UNUSED_ARG(arg);
   ACE_DEBUG((LM_DEBUG,"Thread (%t) Created to do some work"));
   ::number++;
   ACE_DEBUG((LM_DEBUG," and number is %d\n",::number));

   //Let the other guy go while I fall asleep for a random period
   //of time
   ACE_Thread::yield();
   ACE_OS::sleep(ACE_OS::rand()%2);

   //Exiting now
   ACE_DEBUG((LM_DEBUG,
        "\t\t Thread (%t) Done! \t The number is now: %d\n",number));
   ACE_OS::fflush(stdout);
   return 0;
   }


int main(int argc, char *argv[]){
if(argc<2)
   ACE_DEBUG((LM_DEBUG,"Usage: <program_name> <number of threads>\n"));

int n_threads=ACE_OS::atoi(argv[1]);
//Setup the random number generator
ACE_OS::srand(::seed);

//Spawn off n_threads number of threads. Note that this can also be done
//using the spawn_n() call which spawns off the required number of
//threads on its own without the loop.
for(int i=0; i<n_threads; i++){
   if(ACE_Thread::spawn((ACE_THR_FUNC)worker)==-1)
      ACE_DEBUG((LM_DEBUG,"Error in spawning thread\n"));
   }

//Wait for all the threads to exit before you let the main fall through
//and have the process exit. This way of using join is non-portable
//and may not work on a system using pthreads.
int check_count=0;
while(ACE_Thread::join(NULL,NULL,NULL)==0) check_count++;
ACE_ASSERT(check_count==n_threads);
```

```
}
```

This program is a simple example in which first `n_thread` number of worker threads are created to perform the *worker()* function defined in the program. The threads are created by using the *ACE_Thread::spawn()* call (Note that there is another call, *spawn_n()*, which allows you to spawn *n* threads at the same time). This call is passed a pointer to the function which is to be called as the starting point of execution for the thread. (In this case the *worker()* function). One important point to note is that *ACE_Thread::spawn()* requires all starting functions (methods) for threads to be static or global (as is required when using the OS threads API directly).

Once the worker function starts, it increments the global `number` variable, reports its present value and then calls *ACE_Thread::yield()* to yield the processor to another thread. This is done because the current thread knows it will be going to sleep and thus won't need the CPU. The *sleep()* is for a random period of time. After the thread wakes up it, informs the user of the present value of `number` and falls out of the *worker()* function.

Once a thread returns from its starting function it implicitly issues a thread *exit()* call on the thread library and exits. The worker thread thus exits once it "falls out" of the *worker()* function. The main thread, which was responsible for creating the worker thread, is "waiting" for all other threads to complete their execution and exit before it exits. When the main thread does exit (by falling out of the *main()*function), the entire process will be destroyed. (If the main thread is not forced to wait for the other threads then when it dies, the process will automatically be destroyed, destroying all the worker threads along with it, **before** they complete their job!)

This wait is performed using the *ACE_Thread::join()* call. You can pass, as the first parameter to this call, the thread id of the thread on which you wish to wait. In the example, a NULL parameter is passed in so that join waits for **any** thread. However, if the platform only supports pthreads, this call may not work, since on this platform it is necessary for the actual id of the thread which you are waiting for to be specified as the first parameter of the join call. To do this, we would have to, of course, remember the thread ids of each of the threads that were spawned somewhere and then call join on them, requiring extra management effort.

There are several facts worth noting in this example. First, there is no management functionality available that we can call upon to take care of issuing the join or remembering the ids of the threads that the application has spawned. Second, no synchronization primitives were used in the program to protect the global data. In this case, they were not necessary, since all threads were only performing addition operations on the global. In real life, however, "locks" would be required to protect all shared mutable data (global or static variables).

Synchronization primitives in ACE

ACE has several classes that can be used for synchronization purposes. These classes can be divided into the following categories

- The ACE Locks Class Category
- The ACE Guards Class Category
- The ACE Conditions Class Category
- Miscellaneous ACE Synchronization classes

The ACE Locks Category

The locks category includes classes that wrap simple locking mechanisms, such as mutexes, semaphores, read/write mutexes and tokens. The classes that are available under this category are shown in the table below. Each class name is followed by a brief description of how it can be used:

Name	Description
`ACE_Mutex`	Wrapper class around the mutual exclusion mechanism (which, depending on the platform, may be `mutex_t`, `pthread_mutex_t`, etc.) and are used to provide a simple and efficient mechanism to serialize access to a shared resource. Similar in functionality to a binary sempahore. Can be used for mutual exclusion among both threads and processes.
`ACE_Thread_Mutex`	Can be used in place of `ACE_Mutex` and is specific for synchronization of threads.
`ACE_Process_Mutex`	Can be used in place of `ACE_Mutex` and is specific for synchronization of processes.
`ACE_Null_Mutex`	Provides a do-nothing implementation of the `ACE_Mutex` interface, and can be replaced with it when no synchronization is required.
`ACE_RW_Mutex`	Wrapper classes which encapsulate readers/writers locks. These are locks that are acquired differently for reading and writing, thus enabling multiple readers to read while no one is writing.
`ACE_RW_Thread_Mutex`	Can be used in place of `ACE_RW_Mutex` and is specific for synchronization of threads.
`ACE_RW_Process_Mutex`	Can be used in place of `ACE_RW_Mutex` and is specific for synchronization of processes.
`ACE_Semaphore`	These classes implement a counting semaphore, which is

	useful when a fixed number of threads can access a resource at the same time. In the case where the OS does not provide this kind of synchronization mechanism, it is simulated with mutexes.
`ACE_Thread_Semaphore`	Should be used in place of `ACE_Semaphore` and is specific for synchronization of threads.
`ACE_Process_Semaphore`	Should be used in place of `ACE_Semaphore` and is specific for synchronization of processes.
`ACE_Token`	This provides for a "recursive mutex", i.e. the thread that currently holds the token can reacquire it multiple times without blocking. Also, when the token is released, it makes sure that the next thread which was blocked and waiting for the token is the next one to go.
`ACE_Null_Token`	A do-nothing implementation of the token interface used when you know that multiple threads won't be present.
`ACE_Lock`	An interface class which defines the locking interface. A pure virtual class, which, if used, will entail virtual function call overhead.
`ACE_Lock_Adapter`	A template-based adapter which allows any of the previously mentioned locking mechanisms to adapt to the ACE_Lock interface.

The classes described in the table above all support the same interface. However, these classes are NOT related to each other in any inheritance hierarchy. In ACE, locks are usually parameterized using templates since the overhead of having virtual function calls is, in most cases, unacceptable. The usage of templates allows the programmer a certain degree of flexibility. He can choose the type of locking mechanism he wishes to use at compile time, but not at runtime. Nevertheless, in some places the programmer may need to use dynamic binding and substitution, and for these cases, ACE includes the `ACE_Lock` and `ACE_Lock_Adapter` classes.

Using the Mutex classes

A mutex implements a simple form of synchronization called *"mutual exclusion"* (hence the name mutex). Mutexes prohibit multiple threads from entering a protected or *"critical section"* of code. Consequently, at any moment of time , only one thread is allowed into such a protected section of code.

Before any thread can enter a defined critical section, it must *acquire* ownership of the mutex associated with that section. If another thread already owns the mutex for the critical section then other threads cant enter. These other threads will be forced to wait until the current owner *releases* the mutex.

When do you use mutexes? Mutexes are used to protect shared mutable code, i.e. data that is either global or static. Such data must be protected by mutexes to prevent its corruption when multiple threads access it at the same time.

The following example illustrates usage of the `ACE_Thread_Mutex` class. Notice that `ACE_Mutex` could easily replace the use of `ACE_Thread_Mutex` class here since they both have the same interface.

Example 2

```
#include "ace/OS.h"
#include "ace/Synch.h"

//Arguments that are to be passed to the worker thread are passed
//through this class.
class Args{
public:
Args(int iterations):
   mutex_(),iterations_(iterations){}
ACE_Thread_Mutex mutex_;
int iterations_;
};

//The starting point for the worker threads
static void*
worker(void*arguments){
Args *arg= (Args*) arguments;
for(int i=0;i<arg->iterations_;i++){
   ACE_DEBUG((LM_DEBUG,
            "(%t) Trying to get a hold of this iteration\n"));
   //This is our critical section
   arg->mutex_.acquire();
   ACE_DEBUG((LM_DEBUG,"(%t) This is iteration number %d\n",i));
   //work
   ACE_OS::sleep(2);
   arg->mutex_.release();
   }
return 0;
}

int main(int argc, char*argv[]){
if(argc<2){
   ACE_OS::printf("Usage: egx <number_of_threads>
                                    <number_of_iterations>\n");
   ACE_OS::exit(1);
   }
//Setup the arguments
Args arg(ACE_OS::atoi(argv[2]));

ACE_Thread::spawn_n
       (ACE_OS::atoi(argv[1]),(ACE_THR_FUNC)worker,(void*)&arg);
```

```
//Spawn the worker threads
while(ACE_Thread::join(NULL,NULL,NULL)==0);
}
```

In the above example, the *ACE_Thread* wrapper class is used to spawn off multiple threads to perform the *worker()* function, as in the previous example. Each thread is passed in an *Arg* object which contains the number of iterations it is supposed to perform and the mutex that it is going to use.

In this example, on startup, each thread immediately enters a *for* loop. Once inside the loop, the thread enters a critical section. The work done within this critical section is protected by using an *ACE_Thread_Mutex* mutex object. This object was passed in as an argument to the worker thread from the main thread. Control of the critical section is achieved by acquiring ownership of the mutex. This is done by issuing an *acquire()* call on the *ACE_Thread_Mutex* object. Once the mutex is acquired, no other thread can enter this section of code. Control of the critical section is released by using the *release()* call. Once ownership of the mutex is relinquished, any other waiting threads are awakened. These threads then compete to obtain ownership of the mutex. Whichever thread manages to acquire ownership first enters the critical section.

Using the Lock and Lock Adapter for dynamic binding

As mentioned earlier, the mutex variety of locks is meant to be used either directly in your code, or, if flexibility is desired, as a template parameter. However, if you need to change the type of lock that is used with your code dynamically, i.e. at run-time, these locks cannot be used.

To counter this problem, ACE includes the *ACE_Lock* and *ACE_Lock_Adapter* classes, which allow for such run-time substitution.

The following example illustrates how the *ACE_Lock* class and *ACE_Lock_Adapter* provide the application programmer with the facility to use dynamic binding and substitution with the locking mechanisms.

Example 3

```
#include "ace/OS.h"
#include "ace/Synch.h"
#include "ace/Synch_T.h"

//Arguments that are to be passed to the worker thread are passed
//through this class.
class Args{
public:
Args(ACE_Lock* lock,int iterations):
   mutex_(lock),iterations_(iterations){}
ACE_Lock* mutex_;
int iterations_;
};
```

```
//The starting point for the worker threads
static void*
worker(void*arguments){
Args *arg= (Args*) arguments;
for(int i=0;i<arg->iterations_;i++){
   ACE_DEBUG((LM_DEBUG,
            "(%t) Trying to get a hold of this iteration\n"));
   //This is our critical section
   arg->mutex_->acquire();
   ACE_DEBUG((LM_DEBUG,"(%t) This is iteration number %d\n",i));
   //work
   ACE_OS::sleep(2);
   arg->mutex_->release();
   }
return 0;
}

int main(int argc, char*argv[]){
if(argc<4){
   ACE_OS::printf("Usage: egx <number_of_threads>
                  <number_of_iterations> <lock_type>\n");
   ACE_OS::exit(1);
   }
//Lock used by application
ACE_Lock *lock;

//Decide which lock you want to use at run time. Possible due to
//ACE_Lock.
if(ACE_OS::strcmp(argv[3],"Thread"))
   lock=new ACE_Lock_Adapter<ACE_Thread_Mutex>;
else
   lock=new ACE_Lock_Adapter<ACE_Mutex>

//Setup the arguments
Args arg(lock,ACE_OS::atoi(argv[2]));
//Spawn the worker threads
ACE_Thread::spawn_n
            (ACE_OS::atoi(argv[1]),(ACE_THR_FUNC)worker,(void*)&arg);
while(ACE_Thread::join(NULL,NULL,NULL)==0);
}
```

In this example, the only difference from the previous example is that the `ACE_Lock` class is used with `ACE_Lock_Adapter` to provide dynamic binding. The decision as to whether the underlying locking mechanism will be `ACE_Thread_Mutex` or `ACE_Mutex` is made from command line arguments while the program is running. The advantage of using dynamic binding, again, is that the actual locking mechanism *can be substituted* at run time. The disadvantage is that each call to the lock now entails an extra level of indirection through the virtual function table.

As mentioned in the table the ACE_Token class provides for a “recursive mutex”, which can be reacquired multiple times by the same thread that had initially acquired it. This means that if a thread calls a recursive routine which acquires a lock again and again the thread won't block on itself. However, if any other thread tries to acquire ownership of the lock, it will be blocked. Another important fact about the ACE_Token type of lock is that, once it releases the lock, it passes ownership to the next thread which had asked for it. This is not how locks are usually released. In the usual case, once the lock is released, any of the threads that was previously blocked on it, or any thread that is currently trying to acquire it, may take the lock.

When I ran the previous example (example 3) on SunOS 5.x I found that the thread that released the lock was the one that managed to reacquire it too! (in around 90% of the cases.) However if you run the example with the ACE_Token class as the locking mechanism each thread has its turn and then gives up to the next thread in line.

Example 4

```
#include "ace/OS.h"
#include "ace/Token.h"

//Arguments that are to be passed to the worker thread are passed
//through this class.
class Args{
public:
Args(int iterations):
   mutex_(),iterations_(iterations){}
ACE_Token mutex_;
int iterations_;
};

//The starting point for the worker threads
static void*
worker(void*arguments){
Args *arg= (Args*) arguments;
for(int i=0;i<arg->iterations_;i++){
   ACE_DEBUG((LM_DEBUG,"(%t) Trying to get a hold of this
iteration\n"));
   //This is our critical section
   arg->mutex_.acquire();
   ACE_DEBUG((LM_DEBUG,"(%t) This is iteration number %d\n",i));
   //work
   ACE_OS::sleep(2);
   arg->mutex_.release();
   }
return 0;
}


int main(int argc, char*argv[]){
if(argc<4){
```

```
   ACE_OS::printf("Usage: egx <number_of_threads>
                 <number_of_iterations> <lock_type>\n");
   ACE_OS::exit(1);
   }

//Setup the arguments
Args arg(ACE_OS::atoi(argv[2]));
//Spawn the worker threads
ACE_Thread::spawn_n(ACE_OS::atoi(argv[1]),(ACE_THR_FUNC)worker,(void*)&a
rg);

while(ACE_Thread::join(NULL,NULL,NULL)==0);

}
```

The ACE Guards Category

The Guards in ACE are used to automatically acquire and release locks. An object of the Guard class defines a "block" of code over which a lock is acquired. The lock is released automatically when the block is exited.

The Guard classes in ACE are templates which are parameterized with the type of locking mechanism required. The underlying lock could be any of the classes that were described in the ACE Locks Category, i.e. any of the mutex or lock classes. This works by using the constructor of the object to acquire the lock and the destructor to release the lock. The following Guards are available in ACE:

Name	Description
`ACE_Guard`	Automatically calls *acquire()* and *release()* on the underlying lock. Can be passed any of the locks in the ACE locks class category as its template parameter.
`ACE_Read_Guard`	Automatically calls *acquire_read()* and *release()* on the underlying lock.
`ACE_Write_Guard`	Automatically calls *acquire_write()* and *release()* on the underlying lock.

The following example illustrates how guards can be used:

```
Example 5
#include "ace/OS.h"
#include "ace/Synch.h"
#include "ace/Synch_T.h"
#include "ace/Token.h"

//Arguments that are to be passed to the worker thread are passed
//through this class.
class Args{
public:
```

```
Args(int iterations):
   mutex_(),iterations_(iterations){}
ACE_Token mutex_;
int iterations_;
};

//The starting point for the worker threads
static void*
worker(void*arguments){
Args *arg= (Args*) arguments;
for(int i=0;i<arg->iterations_;i++){
   ACE_DEBUG((LM_DEBUG,"(%t) Trying to get a hold of this
iteration\n"));
      //begin critical section
      ACE_Guard<ACE_Token> guard(arg->mutex_);
      //This is our critical section
      ACE_DEBUG((LM_DEBUG,"(%t) This is iteration number %d\n",i));
      //work
      ACE_OS::sleep(2);
      //end critical section
   }
return 0;
}


int main(int argc, char*argv[]){
   if(argc<3){
      ACE_OS::printf("Usage: egx <number_of_threads>
                      <number_of_iterations> \n");
      ACE_OS::exit(1);
      }

   //Setup the arguments
   Args arg(ACE_OS::atoi(argv[2]));
   //Spawn the worker threads
   ACE_Thread::spawn_n(ACE_OS::atoi(argv[1]),
            (ACE_THR_FUNC)worker,(void*)&arg);
   //Wait for all work threads to exit before exiting the process.
   while(ACE_Thread::join(NULL,NULL,NULL)==0);
   }
```

In the above example, a guard manages the critical section in the worker thread. The Guard object is created from the *ACE_Guard* template class. The template is passed the type of lock the guard should use. The guard object is created by passing the actual lock object it needs to acquire through its constructor. The lock is internally and automatically acquired by *ACE_Guard* and the section in the *for* loop is thus a protected critical section. Once it comes out of scope, the guard object is automatically destroyed, which causes the lock to be released.

The ACE Conditions Category

The `ACE_Condition` class is a wrapper class around the condition variable primitive of the OS. So what are condition variables anyway?

Oftentimes a thread needs a certain condition to be satisfied before it can continue with its operation. For example, consider a thread which needs to insert messages onto a global message queue. Before inserting any messages, it must check to see if there is space available on the message queue. If the message queue is in the full state, then it cannot do anything, and must go to sleep and try again at some later time. That is, before accessing the global resource, a *condition* must be true. Later, when another thread empties the message queue, there should be a way to inform or signal the original thread that there is now room on the message queue and that it should now try to insert the message again. This can be done using condition variables. Condition variables are used not as mutual exclusions primitives, but as indicators that a certain condition has been reached, and thus a thread which was blocked because the condition was not true should try to continue.

Your program should go through the following steps when using condition variables:

- Acquire the lock (mutex) to the global resource (e.g. the message queue).
- Check the condition. (e.g. Does the message queue have space on it?).
- If condition fails then call the condition variables *wait()* method. Wait for some future point in time when the condition may be true.
- When another thread performs some action on the global resource, it *signal()*s all the other threads that have tried some condition on the resource. (For example, another thread dequeues a message from the message queue and then *signal()*s on the condition variable, so that the threads that are blocked on the *wait()* can try to insert their message into the queue.)
- After waking up, re-check to see if the condition is now true. If it is, then perform some action on the global resource. (For example, enqueue a message on the global message queue.)

One important thing you should notice is that the condition variable mechanism, i.e. *ACE_Cond*, takes care of releasing the mutex on the global resource before blocking internally in the wait call. If it did not do this, then no other thread could work on the resource (which is the cause of the change in the condition). Also, once the blocked thread is *signal*ed to wake up again, it internally re-acquires the lock before checking the condition.

The example below is a rehash of the first example in this chapter. If you remember, we had said that using the *ACE_Thread::join()* call was non-portable (in fact some OSs, such as VxWorks, don't even have a *join()* method) and that it was generally not such a good idea to use it with all null parameters, as we have been using in all the previous examples. However, we needed to do this since otherwise the main thread would terminate without waiting for the child threads. This would result in not only its own self destruction, but also the destruction of the child threads and the process. One simple solution to this problem (using condition variables) would be for the main thread to wait for the condition “all threads are done” to be true before exiting. The last thread could signal the waiting main thread through a condition variable that all threads are done and it is the last. The main thread would then go ahead and exit the application and destroy the process. This is illustrated below:

Example 6

```
#include "ace/Thread.h"
#include "ace/OS.h"
#include "ace/Synch_T.h"
#include "ace/Synch.h"

static int number=0;
static int seed=0;

class Args{
public:
Args(ACE_Condition<ACE_Thread_Mutex> *cond, int threads):
   cond_(cond), threads_(threads){}
ACE_Condition<ACE_Thread_Mutex> *cond_;
int threads_;
};

static void* worker(void *arguments){
   Args *arg= (Args*)arguments;
   ACE_DEBUG((LM_DEBUG,"Thread (%t) Created to do some work\n"));
   ::number++;
   //Work
   ACE_OS::sleep(ACE_OS::rand()%2);

   //Exiting now
   ACE_DEBUG((LM_DEBUG,
            "\tThread (%t) Done! \n\tThe number is now: %d\n",number));
   //If all threads are done signal main thread that
   //program can now exit
   if(number==arg->threads_){
      ACE_DEBUG((LM_DEBUG,
                  "(%t) Last Thread!\n All threads have done their job!
                  Signal main thread\n"));
      arg->cond_->signal();
      }
return 0;
}


int main(int argc, char *argv[]){
   if(argc<2){
      ACE_DEBUG((LM_DEBUG,
            "Usage: <program_name> <number of threads>\n"));
      ACE_OS::exit(1);
      }
```

```
   int n_threads=ACE_OS::atoi(argv[1]);

   //Setup the random number generator
   ACE_OS::srand(::seed);

   //Setup arguments for threads
   ACE_Thread_Mutex mutex;
   ACE_Condition<ACE_Thread_Mutex> cond(mutex);
   Args arg(&cond,n_threads);

   //Spawn off n_threads number of threads
   for(int i=0; i<n_threads; i++){
      if(ACE_Thread::spawn((ACE_THR_FUNC)worker,(void*)&arg,
                           THR_DETACHED|THR_NEW_LWP)==-1)
        ACE_DEBUG((LM_DEBUG,"Error in spawning thread\n"));
        }
   //Wait for signal indicating that all threads are done and program
   //can exit. The global resource here is “number” and the condition
   //that the condition variable is waiting for is number==n_threads.
   mutex.acquire();
   while(number!=n_threads)
      cond.wait();
   ACE_DEBUG((LM_DEBUG,"(%t) Main Thread got signal. Program
                exiting..\n"));
   mutex.release();
   ACE_OS::exit(0);
}
```

Notice that before evaluating the condition, a mutex is acquired by the main thread. The condition is then evaluated. If the condition is not true, then the main thread calls a wait on the condition variable. The condition variable in turn releases the mutex *automatically* and causes the main thread to fall asleep. Condition variables are always used in conjunction with a mutex like this. This is a general pattern [I] that can be described as

while(expression NOT TRUE) wait on condition variable;

Remember that condition variables are not used for mutual exclusion, but are used for the signaling functionality we have been describing.

Besides the *ACE_Condition* class, ACE also includes an *ACE_Condition_Thread_Mutex* class, which uses an *ACE_Thread_Mutex* as the underlying locking mechanism for the global resource..

Miscellaneous Synchronization Classes

Besides the synchronization classes described above ACE also includes other synchronization classes such as *ACE_Barrier* and *ACE_Atomic_Op.*

Barriers have a good name. The name pretty much describes what barriers are supposed to do. A group of threads can use a barrier to collectively synchronize with each other. Each thread in the group executes until it arrives at the barrier, after which it blocks. After all the involved threads have reached their respective barriers, they all continue with their execution. That is, they all block one by one waiting for the others to reach the barrier. Once all threads reach the "barrier point" in their execution paths, they all restart together.

In ACE, the barrier is implemented in the *ACE_Barrier* class. When the barrier object is instantiated, it is passed the number of threads it is going to be waiting on. Each thread issues a *wait()* call on the barrier object once they reach the "barrier point" in their execution path. They block at this point and wait for the other threads to reach their respective "barrier" points before they all continue together. When the barrier has received the appropriate number of *wait()* calls from the involved threads, it wakes up all the blocked threads together.

The following example illustrates how barriers can be used with ACE

Example 7

```
#include "ace/Thread.h"
#include "ace/OS.h"
#include "ace/Synch_T.h"
#include "ace/Synch.h"

static int number=0;
static int seed=0;

class Args{
public:
Args(ACE_Barrier *barrier):
   barrier_(barrier){}
ACE_Barrier *barrier_;
};

static void*
worker(void *arguments){
   Args *arg= (Args*)arguments;
   ACE_DEBUG((LM_DEBUG,"Thread (%t) Created to do some work\n"));
   ::number++;

   //Work
   ACE_OS::sleep(ACE_OS::rand()%2);

   //Exiting now
   ACE_DEBUG((LM_DEBUG,
            "\tThread (%t) Done! \n\tThe number is now: %d\n",number));
   //Let the barrier know we are done.
   arg->barrier_->wait();
   ACE_DEBUG((LM_DEBUG,"Thread (%t) is exiting \n"));
```

```
   return 0;
}


int main(int argc, char *argv[]){
if(argc<2){
   ACE_DEBUG((LM_DEBUG,"Usage: <program_name> <number of threads>\n"));
   ACE_OS::exit(1);
   }

int n_threads=ACE_OS::atoi(argv[1]);
ACE_DEBUG((LM_DEBUG,"Preparing to spawn %d threads",n_threads));
//Setup the random number generator
ACE_OS::srand(::seed);

//Setup arguments for threads
ACE_Barrier barrier(n_threads);
Args arg(&barrier);

//Spawn off n_threads number of threads
for(int i=0; i<n_threads; i++){
   if(ACE_Thread::spawn((ACE_THR_FUNC)worker,
                  (void*)&arg,THR_DETACHED|THR_NEW_LWP)==-1)
      ACE_DEBUG((LM_DEBUG,"Error in spawning thread\n"));
   }

//Wait for all the other threads to let the main thread
// know that they are done using the barrier
barrier.wait();
ACE_DEBUG((LM_DEBUG,"(%t)Other threads are finished. Program
exiting..\n"));
ACE_OS::sleep(2);
}
```

In this example, a barrier is created and then passed to the worker threads. Each worker thread calls *wait()* on the barrier just before exiting, causing all threads to be blocked after they have completed their work and right before they exit. The main thread also blocks just before exiting. Once all threads (including main) have reached the end of their execution, they all continue and then exit together.

Atomic Op

The *ACE_Atomic_Op* class is used to transparently parameterize synchronization into basic arithmetic operations. *ACE_Atomic_Op* is a template class that is passed in a locking mechanism and the type which is be parameterized. ACE achieves this by overloading all arithmetic operators and ensuring that a lock is acquired before the operation and then released after the operation. The operation itself is delegated to the class passed in through the template.

The following example illustrates the usage of this class.

Example 8

```
#include "ace/Synch_T.h"
#include "ace/Synch.h"

//Global mutable and shared data on which we will perform simple
//arithmetic operations which will be protected.
ACE_Atomic_Op<ACE_Thread_Mutex,int> foo;

//The worker threads will start from here.
static void* worker(void *arg){
  ACE_UNUSED_ARG(arg);
  foo=5;
  ACE_ASSERT (foo == 5);

  ++foo;
  ACE_ASSERT (foo == 6);

   --foo;
  ACE_ASSERT (foo == 5);

  foo += 10;
  ACE_ASSERT (foo == 15);

  foo -= 10;
  ACE_ASSERT (foo == 5);

  foo = 5L;
  ACE_ASSERT (foo == 5);
   return 0;
}

int main(int argc, char *argv[]){
if(argc<2){
   ACE_DEBUG((LM_DEBUG,"Usage: <program_name> <number of threads>\n"));
   ACE_OS::exit(1);
   }

int n_threads=ACE_OS::atoi(argv[1]);
ACE_DEBUG((LM_DEBUG,"Preparing to spawn %d threads\n",n_threads));

//Spawn off n_threads number of threads
for(int i=0; i<n_threads; i++){
   if(ACE_Thread::spawn(
        (ACE_THR_FUNC)worker,0,THR_DETACHED|THR_NEW_LWP)==-1)
   ACE_DEBUG((LM_DEBUG,"Error in spawning thread\n"));
   }
//Wait for all the other threads to let the main thread know
//when it is time to exit the program
while(ACE_Thread::join(NULL,NULL,NULL)==0);

ACE_DEBUG((LM_DEBUG,"(%t)Other threads are finished. Program
      exiting..\n"));
}
```

In the above program, several simple arithmetic operations are performed on the `foo` variable. After the operation, an assertion check is performed to make sure that the value of the variable is what it is "supposed" to be.

You may be wondering why synchronization of such arithmetic primitives (such as in the program above) is necessary. You must think that increment and decrement operations should be atomic.

However, these operations are usually NOT atomic. The CPU would probably divide the instructions into three steps: a read of the variable, an increment or decrement and then a write back. In such a case, if atomic operations are not used, then the following scenario may be developed.

- Thread one reads the variable, increments and gets swapped out without writing the new value back.
- Thread two reads the old value of the variable, increments it and writes back a new incremented value.
- Thread one overwrites thread two's increment with its own.

The above example program *may* not break even if there are no synchronization primitives in place. The reason is that the thread in this case is compute-bound and the OS may not pre-empt such a thread. However, writing such code would be unsafe, since you cannot rely on the OS scheduler to act this way. In any case, in most environments, timing relationships are non-deterministic (because of real-time effects like page faults, or the use of timers or because of actually having multiple physical processors).

Thread Management with the *ACE_Thread_Manager*

In all the previous examples, we have been using the *ACE_Thread* wrapper class to create and destroy threads. However, the functionality of this wrapper class is somewhat limited. The *ACE_Thread_Manager* provides a superset of the facilities that are available in *ACE_Thread*. In particular, it adds management functionality to make it easier to start, cancel, suspend and resume a group of related threads. It provides for creating and destroying groups of threads and tasks (*ACE_Task* is a higher level construct then threads and can be used in ACE for doing multi-threaded programming. We will talk more about tasks later). It also provides functionality such as sending signals to a group of threads or waiting on a group of threads instead of calling join in the non-portable fashion that we have seen in the previous examples.

The following example illustrates how *ACE_Thread_Manager* can be used to create, and then wait for, the completion of a group of threads.

Example 9

```
#include "ace/Thread_Manager.h"
```

```
#include "ace/OS.h"
#include "ace/Get_Opt.h"

static void* taskone(void*){
   ACE_DEBUG((LM_DEBUG,"Thread:(%t)started Task one! \n"));
   ACE_OS::sleep(2);
   ACE_DEBUG((LM_DEBUG,"Thread:(%t)finished Task one!\n"));
   return 0;
   }

static void* tasktwo(void*){
   ACE_DEBUG((LM_DEBUG,"Thread:(%t)started Task two!\n"));
   ACE_OS::sleep(1);
   ACE_DEBUG((LM_DEBUG,"Thread:(%t)finished Task two!\n"));
   return 0;
   }

static void print_usage_and_die(){
   ACE_DEBUG((LM_DEBUG,"Usage program_name
            -a<num threads for pool1> -b<num threads for pool2>"));
   ACE_OS::exit(1);
   }
int main(int argc, char* argv[]){
   int num_task_1;
   int num_task_2;
   if(argc<3)
      print_usage_and_die();

   ACE_Get_Opt get_opt(argc,argv,"a:b:");
   char c;
   while( (c=get_opt())!=EOF){
   switch(c){
   case 'a':
      num_task_1=ACE_OS::atoi(get_opt.optarg);
      break;
   case 'b':
      num_task_2=ACE_OS::atoi(get_opt.optarg);
      break;
   default:
      ACE_ERROR((LM_ERROR,"Unknown option\n"));
      ACE_OS::exit(1);
      }
   }


   //Spawn the first set of threads that work on task 1.
   if(ACE_Thread_Manager::instance()->spawn_n(num_task_1,
                     (ACE_THR_FUNC)taskone,//Execute task one
                     0, //No arguments
                     THR_NEW_LWP, //New Light Weight Process
                     ACE_DEFAULT_THREAD_PRIORITY,
                     1)==-1) //Group ID is 1
      ACE_ERROR((LM_ERROR,
                  Failure to spawn first group of threads: %p \n"));
```

```
   //Spawn second set of threads that work on task 2.
   if(ACE_Thread_Manager::instance()->spawn_n(num_task_2,
                        (ACE_THR_FUNC)tasktwo,//Execute task one
                        0, //No arguments
                        THR_NEW_LWP, //New Light Weight Process
                        ACE_DEFAULT_THREAD_PRIORITY,
                        2)==-1)//Group ID is 2
      ACE_ERROR((LM_ERROR,
                  "Failure to spawn second group of threads: %p \n"));


   //Wait for all tasks in grp 1 to exit
   ACE_Thread_Manager::instance()->wait_grp(1);
   ACE_DEBUG((LM_DEBUG,"Tasks in group 1 have exited! Continuing \n"));
   //Wait for all tasks in grp 2 to exit
   ACE_Thread_Manager::instance()->wait_grp(2);
   ACE_DEBUG((LM_DEBUG,"Tasks in group 2 have exited! Continuing \n"));
}
```

This next example illustrates the suspension, resumption and co-operative cancellation mechanisms that are available in the *ACE_Thread_Manager.*

Example 10

```
// Test out the group management mechanisms provided by the
// ACE_Thread_Manager, including the group suspension and resumption,
//and cooperative thread cancellation mechanisms.
#include "ace/Thread_Manager.h"
static const int DEFAULT_THREADS = ACE_DEFAULT_THREADS;
static const int DEFAULT_ITERATIONS = 100000;

static void *
worker (int iterations)
{
  for (int i = 0; i < iterations; i++){
      if ((i % 1000) == 0){
     ACE_DEBUG ((LM_DEBUG,
            "(%t) checking cancellation before iteration %d!\n",
            i));
      //Before doing work check if you have been canceled. If so don't
      //do any more work.

     if (ACE_Thread_Manager::instance ()->testcancel
                              (ACE_Thread::self ()) != 0){
         ACE_DEBUG ((LM_DEBUG,
              "(%t) has been canceled before iteration %d!\n",i));
         break;
       }
     }
   }

  return 0;
```

```
}


int main (int argc, char *argv[]){
  int n_threads = argc > 1 ? ACE_OS::atoi (argv[1]) : DEFAULT_THREADS;
  int n_iterations = argc > 2 ? ACE_OS::atoi (argv[2]) :
                                               DEFAULT_ITERATIONS;

   ACE_Thread_Manager *thr_mgr = ACE_Thread_Manager::instance ();
   //Create a group of threads n_threads that will execute the worker
   //function the spawn_n method returns the group ID for the group of
   //threads that are spawned. The argument n_iterations is passed back
   //to the worker. Notice that all threads are created detached.
   int grp_id = thr_mgr->spawn_n (n_threads, ACE_THR_FUNC (worker),
                          (void *) n_iterations,
                          THR_NEW_LWP | THR_DETACHED);

  // Wait for 1 second and then suspend every thread in the group.
  ACE_OS::sleep (1);
  ACE_DEBUG ((LM_DEBUG, "(%t) suspending group\n"));
  if (thr_mgr->suspend_grp (grp_id) == -1)
    ACE_ERROR ((LM_DEBUG, "(%t) %p\n", "Could not suspend_grp"));

  // Wait for 1 more second and then resume every thread in the
  // group.
  ACE_OS::sleep (1);
  ACE_DEBUG ((LM_DEBUG, "(%t) resuming group\n"));
  if (thr_mgr->resume_grp (grp_id) == -1)
    ACE_ERROR ((LM_DEBUG, "(%t) %p\n", "resume_grp"));

  // Wait for 1 more second and then cancel all the threads.
  ACE_OS::sleep (ACE_Time_Value (1));
  ACE_DEBUG ((LM_DEBUG, "(%t) canceling group\n"));
  if (thr_mgr->cancel_grp (grp_id) == -1)
    ACE_ERROR ((LM_DEBUG, "(%t) %p\n", "cancel_grp"));

  // Perform a barrier wait until all the threads have shut down.
  thr_mgr->wait ();
  return 0;
}
```

In this example `n_threads` are created to execute the `worker` function. Each thread loops in the `worker` function for `n_iterations`. While these threads loop in the `worker` function, the main thread will *suspend()* them, then *resume()* them and lastly will cancel them. Each thread in `worker` will check for cancellation using the *testcancel()* method of *ACE_Thread_Manager.*

Thread Specific Storage

When a single threaded program wishes to create a variable whose value persists across multiple function calls, it allocates that data statically or globally. When such a program is made multi-threaded, this global or static data is the same for all the threads. This may or may not be desirable. For example, a pseudo-random generator may need a static or global integer seed variable which is not affected by its value being changed by multiple threads at the same time. However, in other cases the global or static data element may need to be different for each thread that executes. For example, consider a multi-threaded GUI application in which each window runs in a separate thread and has an input port from which it receives event input. Such an input port must remain "persistent" across function calls in the window but also must be window-specific or private. To achieve this, Thread Specific Storage is used. A structure such as the input port can be put into thread specific storage and is logically accessed as if it is static or global when it is actually private to the thread.

Traditionally, thread-specific storage was achieved using a confusing low-level operating system API. In ACE, TSS is achieved using the *ACE_TSS* template class. The class which is to be thread-specific is passed into the *ACE_TSS* template and then all its public methods may be invoked using the C++ `->` operator.

The following example illustrates how simple it is to use thread-specific storage in ACE.

Example 11

```
#include "ace/OS.h"
#include "ace/Synch_T.h"
#include "ace/Thread_Manager.h"

class DataType{
public:
   DataType():data(0){}
   void increment(){ data++;}
   void set(int new_data){ data=new_data;}
   void decrement(){ data--;}
   int get(){return data;}
private:
   int data;
};

ACE_TSS<DataType> data;

static void* thread1(void*){
   data->set(10);
   ACE_DEBUG((LM_DEBUG,"(%t)The value of data is %d \n",data->get()));
   for(int i=0;i<5;i++)
      data->increment();
   ACE_DEBUG((LM_DEBUG,"(%t)The value of data is %d \n",data->get()));
```

```
   return 0;
   }

static void * thread2(void*){
   data->set(100);
   ACE_DEBUG((LM_DEBUG,"(%t)The value of data is %d \n",data->get()));
   for(int i=0; i<5;i++)
      data->increment();
   ACE_DEBUG((LM_DEBUG,"(%t)The value of data is %d \n",data->get()));
   return 0;
   }

int main(int argc, char*argv[]){
//Spawn off the first thread
ACE_Thread_Manager::instance()-
>spawn((ACE_THR_FUNC)thread1,0,THR_NEW_LWP| THR_DETACHED);

//Spawn off the second thread
ACE_Thread_Manager::instance()-
>spawn((ACE_THR_FUNC)thread2,0,THR_NEW_LWP| THR_DETACHED);

//Wait for all threads in the manager to complete.
ACE_Thread_Manager::instance()->wait();
ACE_DEBUG((LM_DEBUG,"Both threads done.Exiting.. \n"));
}
```

In the above example, the class `DataType` was created in thread-specific storage.

The methods of this class are then accessed using the `->` operator from the functions `thread1` and `thread2`, which are executed in two separate threads. The first thread sets the private data variable to 10 and then increments it by 5 to take it to 15. The second thread takes its private data variable, sets its value to 100 and increments it by 5 to 105. Although the data ***looks*** global it is actually thread specific, and each thread prints out 15 and 105 respectively, indicating the same.

There are several advantages in using thread specific storage where possible. If global or data can be kept in thread specific storage, then the overhead due to synchronization can be minimized. This is the major advantage of using TSS.

Chapter

5

Tasks and Active Objects

Patterns for Concurrent Programming

This chapter introduces the *ACE_Task* class, which was mentioned in the previous chapter, and also presents the Active Object pattern. This chapter will basically cover two topics. First, it will cover how to use the *ACE_Task* construct as a high-level object-oriented mechanism for writing multi-threaded programs. Second, it will discuss how the *ACE_Task* is used in the *Active Object Pattern* [II].

Active Objects

So what is an active object anyway? Traditionally, all objects are passive pieces of code. Code in an object is executed within the thread that has issued method calls on it. That is, the calling thread is "borrowed" to execute methods on the passive object.

Active objects, however, act differently. These objects retain their own thread (or even multiple threads) and use this thread for execution of any methods invoked on them. Thus if you think of a traditional object with a thread (or multiple threads) encapsulated within it, you get an active object.

For example, consider an object "**A"** that has been instantiated in the *main()* function of your program. When your program starts up, a single thread is created by the OS, to execute starting from the *main()* function. If you call any methods on object **A,** this thread will "flow" through and execute the code in that method. Once completed, this thread returns back to the point from which the method had been invoked and continues with its execution. However, if "**A"** was an Active Object, this is not what happens. In this case, the main thread is not borrowed by the Active Object. Instead, when a method is invoked on "**A",** the execution of the method occurs within the thread that is retained by the Active Object. Another way to think about this is: if the method is invoked on a passive object (a regular object) then the call will be blocking or synchronous; if, on the other hand, the method is invoked on an Active Object, the call will be non-blocking or asynchronous.

ACE_Task

ACE_Task is the base class for the *Task* or *Active Object* “processing construct” that is available in ACE. This class is one of the classes that is used to implement the Active Object pattern in ACE. All objects that wish to be “active” must derive from this class. You can also think of *ACE_Task* as being a higher level, more OO, thread class.

You must have noticed something “bad” when we used the *ACE_Thread* wrapper, in the previous chapter. Most of the examples in that chapter were programs which were decomposed into functions, instead of objects. Why did this happen? Right, the *ACE_Thread* wrapper needs to be passed a global function name or a static method as an argument. This function (static method) is then used as the “start point” for the thread that is spawned. This naturally led to writing a function for each thread. As we saw this may result in non-object-oriented decomposition of programs.

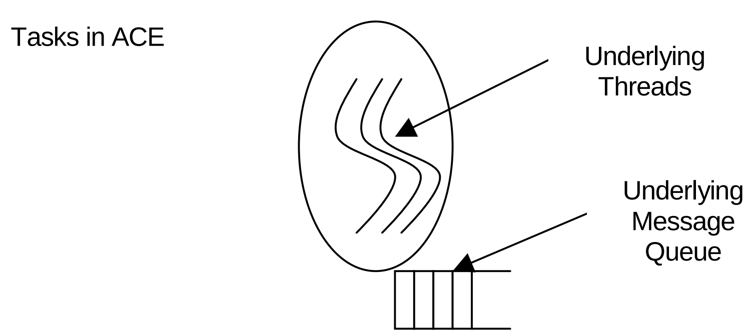
In contrast, *ACE_Task* deals with objects, and is thus easier to think about when building OO programs. Therefore, in most cases, it is better to use a subclass of *ACE_Task* when you need to build multi-threaded programs. There are several advantages in doing this. Foremost is what I just mentioned, i.e. this leads to better OO software. Second, you don’t have to worry about your thread entry point being static, since the entry point for *ACE_Task* is a regular member function. Furthermore, we will see that *ACE_Task* also includes an easy-to-use mechanism for communicating with other tasks.

To reiterate what I just said, *ACE_Task* can be used as

- A Higher Level Thread (which we call a *Task*).
- As an Active Object in the Active Object Pattern.

Structure of a Task

An *ACE_Task* has a structure similar in nature to the structure of “Actors” in Actor Based Systems [III]. This structure is illustrated below:



The above diagram shows that each Task contains one or more threads and an underlying message queue. Tasks communicate with each other through these message queues. However, the message queues are not entities that the programmer needs to be aware of. A sending task can just use the *putq()* call to insert a message into the message queue of another task. The receiving task can then extract this message from its own message queue by using the *getq()* call.

Thus, you can think of a system of more or less autonomous tasks (or active objects) communicating with each other through their message queues. Such an architecture helps considerably in simplifying the programming model for multi-threaded programs.

Creating and using a Task

As mentioned above, to create a task or active object, you must subclass from the *ACE_Task* class. After subclassing, the following steps must be taken.

- **Implementing Service Initialization and Termination Methods**: The *open()* method should contain all task-specific initialization code. This may include resources such as connection control blocks, locks and memory. The *close()* method is the corresponding termination method.
- **Calling the Activation method**: After an Active Object is instantiated, you must activate it by calling *activate()*. The *activate()* method is passed, among other things, the number of threads that are to be created within the Active Object. The *activate()* method will cause the *svc()* method to be the starting point of all threads that are spawned by it.
- **Implementing the Service Specific Processing Methods**: As mentioned above, after an active object is *activate*d, each new thread is started in the *svc()* method. This method must be defined in the subclass by the application developer.

The following example illustrates how you would go about creating a task

Example 1

```
#include "ace/OS.h"
#include "ace/Task.h"

class TaskOne: public ACE_Task<ACE_MT_SYNCH>{

public:
//Implement the Service Initialization and Termination methods
int open(void*){
   ACE_DEBUG((LM_DEBUG,"(%t) Active Object opened \n"));

   //Activate the object with a thread in it.
   activate();
   return 0;
   }
```

```
int close(u_long){
   ACE_DEBUG((LM_DEBUG, "(%t) Active Object being closed down \n"));
   return 0;
   }

int svc(void){
   ACE_DEBUG((LM_DEBUG,
                  "(%t) This is being done in a separate thread \n"));
   // do thread specific work here
   //.......
   //.......

   return 0;
   }
   };

int main(int argc, char *argv[]){
   //Create the task
   TaskOne *one=new TaskOne;

   //Start up the task
   one->open(0);

   //wait for all the tasks to exit
   ACE_Thread_Manager::instance()->wait();
   ACE_DEBUG((LM_DEBUG,"(%t) Main Task ends \n"));
   }
```

The above example illustrates how *ACE_Task* can be used as a higher-level thread class. In the example, the class `TaskOne` derives from *ACE_Task* and implements the *open(), close()* and *svc()* methods. After the task object is instantiated, the *open()* method is invoked on it. This method in turn calls the *activate()* method, which causes a new thread to be spawned and started. The entry point for this thread is the *svc()* routine. The main thread waits for the active object thread to expire and then exits the process.

Communication between tasks

As mentioned earlier, each task in ACE has an underlying message queue (see illustration above). This message queue is used as a means of communication between tasks. When one tasks wants to “talk” to another task, it creates a message and enqueues that message onto the message queue of the task that it wishes to talk to. The receiving task is usually on a *getq()* from its message queue. If no data is available in the queue, it falls asleep. If some other task has inserted something into its queue, it will wake up, pick up the data from its queue and process it. Thus, in this case, the receiving task will receive the message from the sending task and respond to it in some application specific manner.

This next example illustrates how two tasks communicate with each other using their underlying message queue. This example involves an implementation for the classic producer-consumer problem. The Producer Task creates data, which it sends to the Consumer Task. The Consumer Task in turn consumes this data. Using the *ACE_Task* construct, we think of both the Producer and Consumers as separate objects of type *ACE_Task.* These tasks communicate with each other using the underlying message queue.

Example 2

```
#include "ace/OS.h"
#include "ace/Task.h"
#include "ace/Message_Block.h"

//The Consumer Task.
class Consumer:
   public ACE_Task<ACE_MT_SYNCH>{
public:
int open(void*){
   ACE_DEBUG((LM_DEBUG, "(%t) Producer task opened \n"));
   //Activate the Task
   activate(THR_NEW_LWP,1);
   return 0;
   }

//The Service Processing routine
int svc(void){
   //Get ready to receive message from Producer
      ACE_Message_Block * mb =0;
   do{
      mb=0;
      //Get message from underlying queue
      getq(mb);
      ACE_DEBUG((LM_DEBUG,
            "(%t)Got message: %d from remote task\n",*mb->rd_ptr()));
      }while(*mb->rd_ptr()<10);
   return 0;
   }

int close(u_long){
   ACE_DEBUG((LM_DEBUG,"Consumer closes down \n"));
   return 0;
   }

};

class Producer:
   public ACE_Task<ACE_MT_SYNCH>{
public:
Producer(Consumer * consumer):
   consumer_(consumer), data_(0){
   mb_=new ACE_Message_Block((char*)&data_,sizeof(data_));
   }

int open(void*){
```

```
   ACE_DEBUG((LM_DEBUG, "(%t) Producer task opened \n"));

   //Activate the Task
   activate(THR_NEW_LWP,1);
   return 0;
   }

//The Service Processing routine
int svc(void){
   while(data_<11){
      //Send message to consumer
      ACE_DEBUG((LM_DEBUG,
            "(%t)Sending message: %d to remote task\n",data_));
      consumer_->putq(mb_);
      //Go to sleep for a sec.
      ACE_OS::sleep(1);
      data_++;
      }
   return 0;
   }

int close(u_long){
   ACE_DEBUG((LM_DEBUG,"Producer closes down \n"));
   return 0;
   }
private:
   char data_;
   Consumer * consumer_;
   ACE_Message_Block * mb_;
};


int main(int argc, char * argv[]){

   Consumer *consumer = new Consumer;
   Producer * producer = new Producer(consumer);

   producer->open(0);
   consumer->open(0);
   //Wait for all the tasks to exit.
   ACE_Thread_Manager::instance()->wait();
}
```

Here, each of the Producer and Consumer tasks are very similar. Neither has any service initialization or termination code. The *svc()* method for both classes is different, however. After the Producer is activated in the *open()* method, the *svc()* method is called. In this method, the Producer creates a message, which it inserts onto the consumer's queue. The message is created using the *ACE_Message_Block* class. (To read more about how to use *ACE_Message_Block*, please read the chapter on message queues in this guide and in the online ACE tutorials). The producer maintains a pointer to the consumer task (object). It needs this pointer so it can enqueue a message onto the consumer's message queue. The pointer is set up in the *main()* function, through its constructor.

The consumer sits in a loop in its *svc()* method and waits for data to come into its message queue. If there is no data on the message queue, the consumer blocks and falls asleep. (This is done auto-magically by the *ACE_Task* class). Once data does arrive on the consumer's queue, it wakes up and consumes the data.

Here, the data that is sent by the producer consists of one integer. This integer is incremented by the producer each time, before being sent to the consumer.

As you can see, the solution for the producer-consumer problem is simple and object-oriented. Using the *ACE_Thread* wrapper, there is a good chance that a programmer would create a solution which would have a producer and consumer function for each of the threads. *ACE_Task* is the better way to go when writing object-oriented multi-threaded programs.

The Active Object Pattern

The Active Object pattern is used to decouple method execution from method invocation. This pattern facilitates another, more transparent method for inter-task communication.

This pattern uses the *ACE_Task* class as an active object. Methods are invoked on this object as if it were a regular object. That is, method invocation is done through the same old `->` operator, the difference being that the **execution** of these methods occurs in the thread which is encapsulated within *ACE_Task.* The client programmer will see no difference, or only a minimal difference, when programming with passive or active objects. This is highly desirable for a framework developer, where you want to shield the client of the framework from the innards of how the framework is doing its work. Thus a framework USER does not have to worry about threads, synchronization, rendezvous, etc.

How the Active Object Pattern Works

The Active Object pattern is one of the more complicated patterns that has been implemented in ACE and has several participants:

1. An Active Object (based on an *ACE_Task*).
2. An *ACE_Activation_Queue.*
3. Several *ACE_Method_Object*s. (One method object is needed for each of the methods that the active object supports).
4. Several *ACE_Future* Objects. (One is needed for each of the methods that returns a result).

We have already seen how an *ACE_Task* creates and encapsulates a thread. To make an *ACE_Task* an active object, a few additional things have to be done.

A method object must be written for all the methods that are going to be called asynchronously from the client. Each Method Object that is written will derive from the

ACE_Method_Object and will implement the *call()* method. Each Method Object also maintains context information (such as parameters, which are needed to execute the method and an *ACE_Future* Object, which is used to recover the return value. These values are maintained as private attributes). You can consider the method object to be the **closure** of the method call. When a client issues a method call, this causes the corresponding method object to be instantiated and then enqueued on the activation queue. The Method Object is a form of the **Command** pattern. (See the references on Design Patterns).

The *ACE_Activation_Queue* is a queue on which the method objects are enqueued as they wait to be executed. Thus the activation queue contains all of the pending method invocations on it (in the form of method objects). The thread encapsulated in *ACE_Task* stays blocked, waiting for any method objects to be enqueued on the Activation Queue. Once a method object is enqueued, the task dequeues the method object and issues the *call()* method on it. The *call()* method should, in turn, call the corresponding implementation of that method in the *ACE_Task.* After the implementation method returns, the *call()* method *set()*s the result that is obtained in an *ACE_Future* object.

The *ACE_Future* object is used by the client to obtain results for any asynchronous operations it may have issued on the active object. Once the client issues an asynchronous call, it is immediately returned an *ACE_Future* object. The client is then free to try to obtain the results from this future object whenever it pleases. If the client tries to extract the result from the future object before it has been *set()*, the client will block. If the client does not wish to block, it can poll the future object by using the *ready()* call. This method returns 1 if the result has been set and 0 otherwise. The *ACE_Future* object is based on the idea of "polymorphic futures" [[IV]]

The *call()* method should be implemented such that it sets the internal value of the returned *ACE_Future* object to the result obtained from calling the actual implementation method (this actual implementation method is written in *ACE_Task*).

The following example illustrates how the active object pattern can be implemented. In this example, the Active Object is a "Logger" object. The Logger is sent messages which it is to log using a slow I/O system. Since the I/O system is slow, we do not want the main application tasks to be slowed down because of relatively non-time-critical logging. To prevent this, and to allow the programmer to issue the log calls as if they are normal method calls, we use the Active Object Pattern.

The declaration for the Logger class is shown below:

Example 3a

```
//The worker thread with which the client will interact
class Logger: public ACE_Task<ACE_MT_SYNCH>{
public:
   //Initialization and termination methods
   Logger();
   virtual ~Logger(void);
   virtual int open (void *);
   virtual int close (u_long flags = 0);
```

```
   //The entry point for all threads created in the Logger
   virtual int svc (void);

   ///////////////////////////////////////////////////////
   //Methods which can be invoked by client asynchronously.
   ///////////////////////////////////////////////////////

   //Log message
   ACE_Future<u_long> logMsg(const char* msg);

   //Return the name of the Task
   ACE_Future<const char*> name (void);

   ///////////////////////////////////////////////////////
   //Actual implementation methods for the Logger
   ///////////////////////////////////////////////////////
   u_long logMsg_i(const char *msg);
   const char * name_i();
private:
  char *name_;
  ACE_Activation_Queue activation_queue_;
};
```

As we can see, the Logger Active Object derives from *ACE_Task* and contains an *ACE_Activation_Queue.* The Logger supports two asynchronous methods, i.e. `logMsg()` and `name()`. These methods should be implemented such that when the client calls them, they instantiate the corresponding method object type and enqueue it onto the task's private activation queue. The actual implementation for these two methods (which means the methods that "really" contain the code that does the requested job) are `logMsg_i()` and `name_i()`.

The next segment shows the interfaces to the two method objects that we need, one for each of the two asynchronous methods in the Logger Active Object.

Example 3b

```
//Method Object which implements the logMsg() method of the active
//Logger active object class
class logMsg_MO: public ACE_Method_Object{
public:
   //Constructor which is passed a reference to the active object, the
   //parameters for the method, and a reference to the future which
   //contains the result.
  logMsg_MO(Logger * logger, const char * msg,
                      ACE_Future<u_long> &future_result);
  virtual ~logMsg_MO();

   //The call() method will be called by the Logger Active Object
   //class, once this method object is dequeued from the activation
   //queue.  This is implemented so that it does two things. First it
   //must execute the actual implementation method (which is specified
   //in the Logger class. Second, it must set the result it obtains from
```

```
   //that call in the future object that it has returned to the client.
   //Note that the method object always keeps a reference to the same
   //future object that it returned to the client so that it can set the
   //result value in it.
   virtual int call (void);
private:
   Logger * logger_;
   const char* msg_;
   ACE_Future<u_long> future_result_;
};

//Method Object which implements the name() method of the active Logger
//active object class
class name_MO: public ACE_Method_Object{
public:
   //Constructor which is passed a reference to the active object, the
   //parameters for the method, and a reference to the future which
   //contains the result.
   name_MO(Logger * logger, ACE_Future<const char*> &future_result);
   virtual ~name_MO();

   //The call() method will be called by the Logger Active Object
   //class, once this method object is dequeued from the activation
   //queue.  This is implemented so that it does two things. First it
   //must execute the actual implementation method (which is specified
   //in the Logger class. Second, it must set the result it obtains from
   //that call in the future object that it has returned to the client.
   //Note that the method object always keeps a reference to the same
   //future object that it returned to the client so that it can set the
   //result value in it.
   virtual int call (void);
private:
   Logger * logger_;
   ACE_Future<const char*> future_result_;
};
```

Each of the method objects contains a constructor, which is used to create a closure for the method call. This means that the constructor ensures that the parameters and return values for the call are "remembered" by the object by recording them as private member data in the method object. The call method contains code that will delegate the actual implementation methods specified in the Logger Active Object (i.e. `logMsg_i()` and `name_i()`).

This next segment of the example contains the implementation for the two Method Objects.

Example 3c

```
//Implementation for the logMsg_MO method object.
//Constructor
logMsg_MO::logMsg_MO(Logger * logger, const char * msg,
ACE_Future<u_long> &future_result)
   :logger_(logger), msg_(msg), future_result_(future_result){
      ACE_DEBUG((LM_DEBUG, "(%t) logMsg invoked \n"));
```

```
      }
//Destructor
logMsg_MO::~logMsg_MO(){
      ACE_DEBUG ((LM_DEBUG, "(%t) logMsg object deleted.\n"));
      }
//Invoke the logMsg() method
int logMsg_MO::call (void){
   return this->future_result_.set (
                  this->logger_->logMsg_i (this->msg_));
      }
```

```
Example 3c
//Implementation for the name_MO method object.
//Constructor
name_MO::name_MO(Logger * logger, ACE_Future<const char*>
&future_result):
   logger_(logger), future_result_(future_result){
      ACE_DEBUG((LM_DEBUG, "(%t) name() invoked \n"));
      }
//Destructor
name_MO::~name_MO(){
      ACE_DEBUG ((LM_DEBUG, "(%t) name object deleted.\n"));
      }
//Invoke the name() method
int name_MO::call (void){
   return this->future_result_.set (this->logger_->name_i ());
      }
```

The implementation for these two methods object is quite straightforward. As was explained above, the constructor for the method object is responsible for creating a **closure** (capturing the input parameters and the result). The *call()* method calls the actual implementation methods and then sets the value in the future object by using its *ACE_Future::set()* method.

This next segment of code shows the implementation for the Logger Active Object itself. Most of the code is in the *svc()* method. It is in this method that it dequeues method objects from the activation queue and invokes the *call()* method on them.

```
Example 3d
//Constructor for the Logger
Logger::Logger(){
      this->name_= new char[sizeof("Worker")];
      ACE_OS:strcpy(name_,"Worker");
      }
//Destructor
Logger::~Logger(void){
      delete this->name_;
      }

//The open method where the active object is activated
```

```
int Logger::open (void *){
      ACE_DEBUG ((LM_DEBUG, "(%t) Logger %s open\n", this->name_));
      return this->activate (THR_NEW_LWP);
      }

//Called then the Logger task is destroyed.
int Logger::close (u_long flags = 0){
      ACE_DEBUG((LM_DEBUG, "Closing Logger \n"));
      return 0;
      }
//The svc() method is the starting point for the thread created in the
//Logger active object. The thread created will run in an infinite loop
//waiting for method objects to be enqueued on the private activation
//queue. Once a method object is inserted onto the activation queue the
//thread wakes up, dequeues the method object and then invokes the
//call() method on the method object it just dequeued. If there are no
//method objects on the activation queue, the task blocks and falls
//asleep.
int Logger::svc (void){
      while(1){
        // Dequeue the next method object (we use an auto pointer in
        // case an exception is thrown in the <call>).
        auto_ptr<ACE_Method_Object> mo
                   (this->activation_queue_.dequeue ());

        ACE_DEBUG ((LM_DEBUG, "(%t) calling method object\n"));
        // Call it.
        if (mo->call () == -1)
             break;
        // Destructor automatically deletes it.
        }
      return 0;
      }

   //////////////////////////////////////////////////////////////
   //Methods which are invoked by client and execute asynchronously.
   //////////////////////////////////////////////////////////////

   //Log this message
   ACE_Future<u_long> Logger::logMsg(const char* msg){
      ACE_Future<u_long> resultant_future;

      //Create and enqueue method object onto the activation queue
      this->activation_queue_.enqueue
            (new logMsg_MO(this,msg,resultant_future));
      return resultant_future;
      }

   //Return the name of the Task
   ACE_Future<const char*> Logger::name (void){
      ACE_Future<const char*> resultant_future;

      //Create and enqueue onto the activation queue
      this->activation_queue_.enqueue
```

```
            (new name_MO(this, resultant_future));
      return resultant_future;
      }

   ////////////////////////////////////////////////////////
   //Actual implementation methods for the Logger
   ////////////////////////////////////////////////////////

   u_long Logger::logMsg_i(const char *msg){
      ACE_DEBUG((LM_DEBUG,"Logged: %s\n",msg));

      //Go to sleep for a while to simulate slow I/O device
      ACE_OS::sleep(2);
      return 10;
      }

   const char * Logger::name_i(){
      //Go to sleep for a while to simulate slow I/O device
      ACE_OS::sleep(2);
      return name_;
```

The last segment of code illustrates the application code, which instantiates the Logger Active Object and uses it for logging purposes.

Example 3e

```
//Client or application code.
int main (int, char *[]){
   //Create a new instance of the Logger task
   Logger *logger = new Logger;

   //The Futures or IOUs for the calls that are made to the logger.
   ACE_Future<u_long> logresult;
   ACE_Future<const char *> name;

   //Activate the logger
   logger->open(0);

   //Log a few messages on the logger
   for (size_t i = 0; i < n_loops; i++){
      char *msg= new char[50];
      ACE_DEBUG ((LM_DEBUG,
                  Issuing a non-blocking logging call\n"));
      ACE_OS::sprintf(msg, "This is iteration %d", i);
      logresult= logger->logMsg(msg);
      //Don’t use the log result here as it isn't that important...
      }
   ACE_DEBUG((LM_DEBUG,
            "(%t)Invoked all the log calls \
            and can now continue with other work \n"));
   //Do some work over here...
```

```
   // ...
   // ...

   //Find out the name of the logging task
   name = logger->name ();

   //Check to "see" if the result of the name() call is available
   if(name.ready())
      ACE_DEBUG((LM_DEBUG,"Name is ready! \n"));
   else
      ACE_DEBUG((LM_DEBUG,
                  "Blocking till I get the result of that call \n"));
   //obtain the underlying result from the future object.
   const char* task_name;
   name.get(task_name);
           ACE_DEBUG ((LM_DEBUG,
            "(%t)==> The name of the task is: %s\n\n\n", task_name));

   //Wait for all threads to exit.
   ACE_Thread_Manager::instance()->wait();
}
```

The client code issues several non-blocking asynchronous calls on the *Logger* active object. Notice that the calls appear as if they are being made on a regular passive object. In fact, the calls are being executed in a separate thread of control. After issuing the calls to log multiple messages, the client then issues a call to determine the name of the task. This call returns a future to the client. The client then proceeds to check whether the result is set in the future object or not, using the *ready()* method. It then determines the underlying value in the future by using the *get()* method. Notice how elegant the client code is, with no mention of threads, synchronization, etc. Therefore, the active object pattern can be used to help make the lives of your clients a little bit easier.

Chapter

6

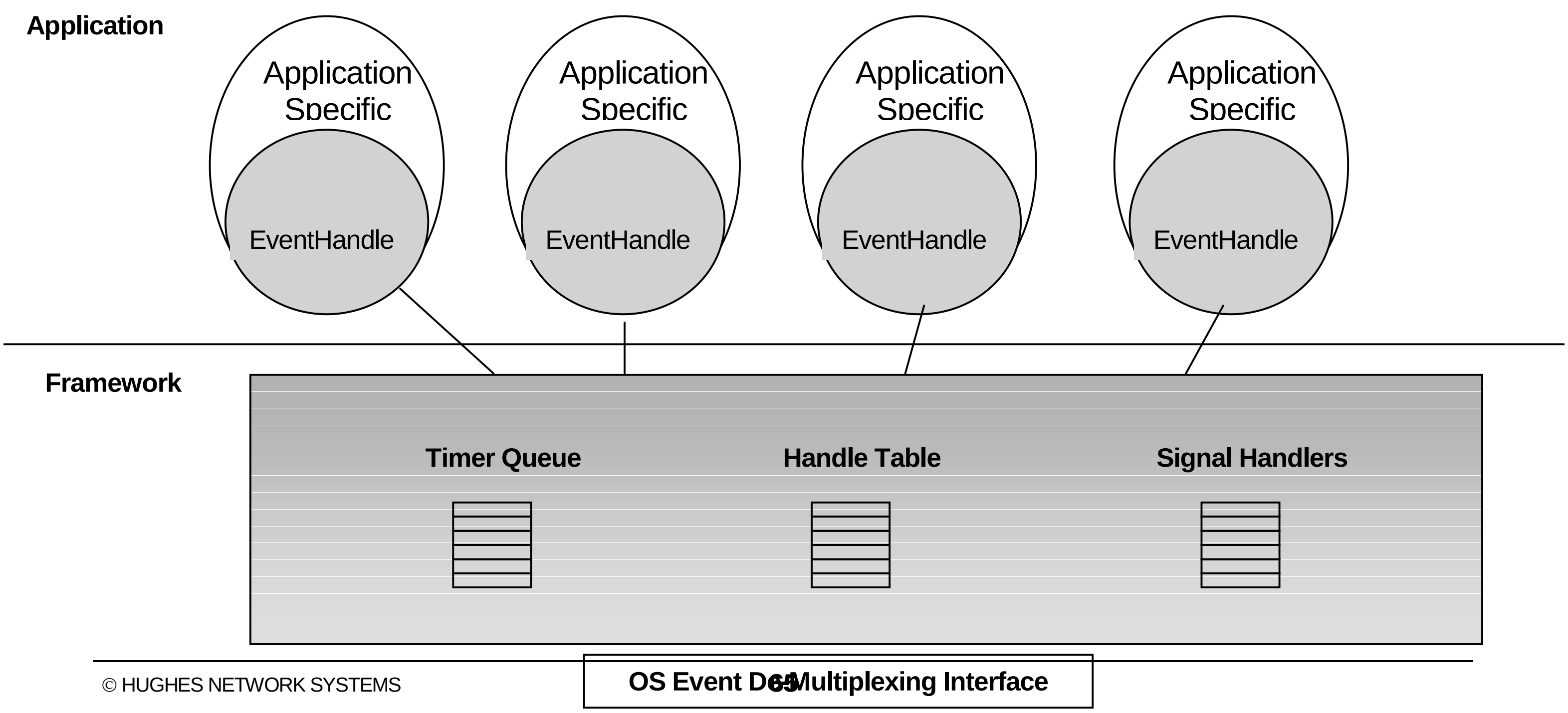
The Reactor

An Architectural Pattern for Event De-multiplexing and Dispatching

The Reactor Pattern has been developed to provide an extensible OO framework for efficient event de-multiplexing and dispatching. Current OS abstractions that are used for event de-multiplexing are difficult and complicated to use, and are therefore error-prone. The Reactor pattern essentially provides for a set of higher-level programming abstractions that simplify the design and implementation of event-driven distributed applications. Besides this, the Reactor integrates together the de-multiplexing of several different kinds of events to one easy-to-use API. In particular, the Reactor handles timer-based events, signal events, I/O-based port monitoring events and user-defined notifications uniformly.

In this chapter, we describe how the Reactor is used to de-multiplex all of these different event types.

Reactor Components



As shown in the above figure, the Reactor in ACE works in conjunction with several components, both internal and external to itself. The basic idea is that the Reactor framework determines that an event has occurred (by listening on the OS Event De-multiplexing Interface) and issues a "*callback*" to a method in a pre-registered *event handler* object. This object is implemented by the application developer and contains application specific code to handle the event.

The **user** (i.e., the application developer) must thus :

1) Create an Event Handler to handle an event he is interested in.

2) Register with the Reactor, informing it that he is interested in handling an event and at this time also passing a pointer to the event handler he wishes to handle the event.

The **Reactor** framework then automatically will:

1) The Reactor maintains tables internally, which associate different event types with event handler objects

2) When an event occurs that the user had registered for, it issues a call back to the appropriate method in the handler.

Event Handlers

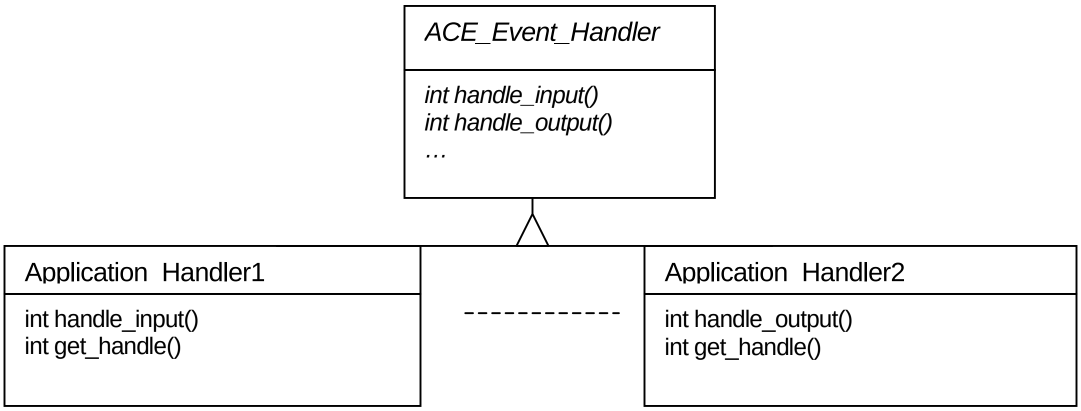
The Reactor pattern is implemented in ACE as the *ACE_Reactor* class, which provides an interface to the reactor framework's functionality.

As was mentioned above, the reactor uses event handler objects as the service providers which handle an event once the reactor has successfully de-multiplexed and dispatched it. The reactor therefore internally remembers **which** event-handler object is to be called back when a certain type of event occurs. This association between events and their event handlers is created when an application registers its handler object with the reactor to handle a certain type of event.

Since the reactor needs to record which Event Handler is to be called back, it needs to know the type of all Event Handler object. This is achieved with the help of the substitution pattern (or in other words through inheritance of the "is a type of" variety). The framework provides an abstract interface class named *ACE_Event_Handler* from which all application specific event handlers MUST derive. (This causes each of the Application Specific Handlers to have the same type, namely *ACE_Event_Handler*, and thus they can be substituted for each other). For more detail on this concept, please see the reference on the Substitution Pattern [V].

If you notice the component diagram above, it shows that the event handler ovals consist of a blue Event_Handler portion, which corresponds to *ACE_Event_Handler*, and a white portion, which corresponds to the application-specific portion.

This is illustrated in the class diagram below:



The *ACE_Event_Handler* class has several different "handle" methods, each of which are used to handle different kinds of events. When an application programmer is interested in a certain event, he subclasses the *ACE_Event_Handler* class and implements the handle methods that he is interested in. As mentioned above, he then proceeds to "register" his event handler class for that particular event with the reactor. The reactor will then make sure that when the event occurs, the appropriate "handle" method in the appropriate event handler object is called back automatically.

Once again, there are basically three steps to using the ACE_Reactor:

- Create a subclass of ACE_Event_Handler and implement the correct "*handle_*" method in your subclass to handle the type of event you wish to service with this event handler. (See table below to determine which "*handle_*" method you need to implement. Note that you may use the same event handler object to handle multiple types of events, and thus may overload more then one of the "*handle_*" methods.)
- Register your Event handler with the reactor by calling *register_handler()* on the reactor object.
- As events occur, the reactor will automatically call back the correct "*handle_*" method of the event handler object that was previously registered with the Reactor to process that event.

A simple example should help to make this a little clearer.

Example 1

```
#include <signal.h>
#include "ace/Reactor.h"
```



```
#include "ace/Event_Handler.h"

//Create our subclass to handle the signal events
//that we wish to handle. Since we know that this particular
//event handler is going to be using signals we only overload the
//handle_signal method.

class
MyEventHandler: public ACE_Event_Handler{
int
handle_signal(int signum, siginfo_t*,ucontext_t*){
   switch(signum){
      case SIGWINCH:
        ACE_DEBUG((LM_DEBUG, "You pressed SIGWINCH \n"));
        break;

      case SIGINT:
        ACE_DEBUG((LM_DEBUG, "You pressed SIGINT \n"));
        break;
        }
   return 0;
   }
};

int main(int argc, char *argv[]){
   //instantiate the handler
   MyEventHandler *eh =new MyEventHandler;
//Register the handler asking to call back when either SIGWINCH
//or SIGINT signals occur. Note that in both the cases we asked the
//Reactor to call back the same Event_Handler i.e., MyEventHandler.
//This is the reason why we had to write a switch statement in the
//handle_signal() method above. Also note that the ACE_Reactor is
//being used as a Singleton object (Singleton pattern)

   ACE_Reactor::instance()->register_handler(SIGWINCH,eh);
   ACE_Reactor::instance()->register_handler(SIGINT,eh);
   while(1)
      //Start the reactors event loop
      ACE_Reactor::instance()->handle_events();
}
```

In the above example, we first create a sub-class of *ACE_Event_Handler* in which we overload the *handle_signal()* method, since we intend to use this handler to handle various types of signals. In the main routine, we instantiate our handler and then call register_handler on the *ACE_Reactor* Singleton, specifying that we wish the event handler 'eh' to be called back when either SIGWINCH (signal on terminal window change) or SIGINT (interrupt signal, usually ^C) occur. After this, we start the reactor's event loop by calling *handle_events()* in an infinite loop. Whenever either of the events happen the reactor will call back the *eh->handle_signal()* method automatically, passing it the signal number which caused the callback, and the `siginfo_t` structure (see siginfo.h for more on `siginfo_t`).

Notice the use of the Singleton pattern to obtain a reference to the global reactor object. Most applications require a single reactor and thus *ACE_Reactor* comes complete with the *instance()* method which insures that whenever this method is called, the same ACE_Reactor instance is returned. (To read more about the Singleton Pattern please see the Design Patterns reference [[VI]].)

The following table shows which methods must be overloaded in the subclass of *ACE_Event_Handler* to process different event types.

Handle Methods in ACE_Event_Handler	Overloaded in subclass to be used to handle event of type:
`handle_signal()`	Signals. When any signal is registered with the reactor, it will call back this method automatically when the signal occurs.
`handle_input()`	Input from I/O device. When input is available on an I/O handle (such as a file descriptor in UNIX), this method will be called back by the reactor automatically.
`handle_exception()`	Exceptional Event. When an exceptional event occurs on an event that has been registered with the reactor (for example, if SIGURG (Urgent Signal) is received), then this method will be automatically called back.
`handle_timeout()`	Timer. When the time for any registered timer expires, this method will be called back automatically.
`handle_output()`	Output possible on I/O device. When the output queue of an I/O device has space available on it, this method will automatically be called back.

Registration of Event Handlers

As we saw in the example above, an event handler is registered to handle a certain event by calling the *register_handler()* method on the reactor. The *register_handler()* method is overloaded, i.e. there are actually several methods for registering different event types, each called *register_handler()*, but having a different signature, i.e. the methods differ in their arguments. The *register_handler()* methods basically take the *handle/event_handler* tuple or the *signal/event_handler* tuple as arguments and add it to the reactor's internal dispatch tables. When an event occurs on *handle*, it find the corresponding *event_handler* in its internal dispatch table and automatically calls back the appropriate method on the *event_handler* it finds. More details of specific calls to register handlers will be illustrated in later sections.

Removal and lifetime management of Event Handlers

Once the required event has been processed, it may not be necessary to keep the event handler registered with the reactor. The reactor thus offers techniques to remove an event handler from its internal dispatch tables. Once the event handler is removed, it will no longer be called back by the reactor.

An example of such a situation could be a server which serves multiple clients. The clients connect to the server, have it perform some work and then disconnect later. When a new client connects to the server, an event handler object is instantiated and registered in the server's reactor to handle all I/O from this client. When the client disconnects then the server must remove the event handler from the reactor's dispatch queue, since it no longer expects any further I/O from the client. In this example, the client/server connection may be closed down, which leaves the I/O handle (file descriptor in UNIX) invalid. It is important that such a defunct handle be removed from the Reactor, since, if this is not done, the Reactor will mark the handle as “ready for reading” and continually call back the *handle_input()* method of the event handler forever.

There are several techniques to remove an event handler from the reactor.

Implicit Removal of Event Handlers from the Reactors Internal dispatch tables

The more common technique to remove a handler from the reactor is implicit removal. Each of the “*handle_*” methods of the event handler returns an integer to the reactor. If this integer is 0, then the event handler remains registered with the reactor after the handle method is completed. However, if the “*handle_*” method returns <0, then the reactor will automatically call back the *handle_close()* method of the Event Handler and remove it from its internal dispatch tables. The *handle_close()* method is used to perform any handler specific cleanup that needs to be done before the event handler is removed, which may include things like deleting dynamic memory that had been allocated by the handler or closing log files.

In the example described above, it is necessary to actually remove the event handler from memory. Such removal can also occur in the *handle_close()* method of the concrete event handler class. Consider the following concrete event handler:

```
class MyEventHandler: public ACE_Event_Handler{
public:
   MyEventHandler(){//construct internal data members}
   virtual int
   handle_close(ACE_HANDLE handle, ACE_Reactor_Mask mask){
      delete this; //commit suicide
      }
   ~MyEventHandler(){//destroy internal data members}
private:
//internal data members
}
```

This class deletes itself when it is de-registers from the reactor and the *handle_close()* hook method is called. It is VERY important however that MyEventHandler is always allocated dynamically otherwise the global memory heap may be corrupted. One way to ensure that the class is always created dynamically is to move the destructor into the private section of the class. For example:

```
class MyEventHandler: public ACE_Event_Handler{
public:
   MyEventHandler(){//construct internal data members}
   virtual int handle_close(ACE_HANDLE handle, ACE_Reactor_Mask mask){
      delete this; //commit suicide}
private:
   //Class must be allocated dynamically
   ~MyEventHandler(){//destroy internal data members}
};
```

Another way to remove an Event Handler from the reactor's internal tables is to explicitly call the *remove_handler()* set of methods of the reactor. This method is also overloaded, as is *register_handler()*. It takes the handle or the signal number whose handler is to be removed and removes it from the reactor's internal dispatch tables. When the *remove_handler()* is called, it also calls the *handle_close()* method of the Event Handler (which is being removed) automatically. This can be controlled by passing in the mask *ACE_Event_Handler::DONT_CALL* to the *remove_handler()* method, which causes the *handle_close()* method NOT to be called. More specific examples of the use of *remove_handler()* will be shown in the next few sections.

Event Handling with the Reactor

In the next few sections, we will illustrate how the Reactor is used to handle various types of events.

I/O Event De-multiplexing

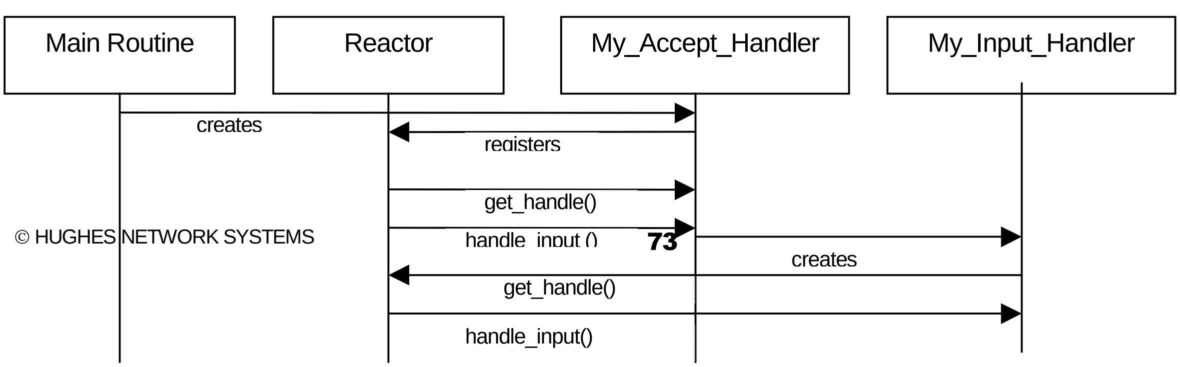
The Reactor can be used to handle I/O device based input events by overloading the *handle_input()* method in the concrete event handler class. Such I/O may be on disk files, pipes, FIFOs or network sockets. For I/O device-based event handling, the Reactor internally uses the handle to the device which is obtained from the operating system. (The handle on UNIX-based system is the file descriptor returned by the OS when a file or socket is opened. In Windows the handle is a handle to the device returned by Windows.). One of the most useful applications of such de-multiplexing is obviously for network applications. The following example will help illustrate how the reactor may be used in conjunction with the concrete acceptor to build a server.

Example 2
```
#include "ace/Reactor.h"
#include "ace/SOCK_Acceptor.h"
#define PORT_NO 19998
typedef ACE_SOCK_Acceptor Acceptor;
//forward declaration
class My_Accept_Handler;

class
My_Input_Handler: public ACE_Event_Handler{
public:
   //Constructor
   My_Input_Handler(){
      ACE_DEBUG((LM_DEBUG,"Constructor\n");
   }
```

```
   //Called back to handle any input received
   int
   handle_input(ACE_HANDLE){
      //receive the data
      peer().recv_n(data,12);
      ACE_DEBUG((LM_DEBUG,"%s\n",data));

      // do something with the input received.
      // ...

      //keep yourself registered with the reactor
      return 0;
      }

   //Used by the reactor to determine the underlying handle
   ACE_HANDLE
   get_handle()const {
      return this->peer_i().get_handle();
      }

   //Returns a reference to the underlying stream.
   ACE_SOCK_Stream &
   peer_i(){
      return this->peer_;
      }

private:
   ACE_SOCK_Stream peer_;
   char data [12];
};


class
My_Accept_Handler: public ACE_Event_Handler{
public:
   //Constructor
   My_Accept_Handler(ACE_Addr &addr){
      this->open(addr);
      }
   //Open the peer_acceptor so it starts to "listen"
   //for incoming clients.
   int
   open(ACE_Addr &addr){
      peer_acceptor.open(addr);
      return 0;
      }
   //Overload the handle input method
   int
   handle_input(ACE_HANDLE handle){
      //Client has requested connection to server.
      //Create a handler to handle the connection
      My_Input_Handler *eh= new My_Input_Handler();

      //Accept the connection "into" the Event Handler
```

```
      if (this->peer_acceptor.accept (eh->peer (), // stream
                        0, // remote address
                        0, // timeout
                        1) ==-1) //restart if interrupted
            ACE_DEBUG((LM_ERROR,"Error in connection\n"));

      ACE_DEBUG((LM_DEBUG,"Connection established\n"));

      //Register the input event handler for reading
      ACE_Reactor::instance()->
        register_handler(eh,ACE_Event_Handler::READ_MASK);

      //Unregister as the acceptor is not expecting new clients
      return -1;
      }

   //Used by the reactor to determine the underlying handle
   ACE_HANDLE
   get_handle(void) const{
      return this->peer_acceptor.get_handle();
      }
private:
   Acceptor peer_acceptor;
};

int main(int argc, char * argv[]){
   //Create an address on which to receive connections
   ACE_INET_Addr addr(PORT_NO);
   //Create the Accept Handler which automatically begins to "listen"
   //for client requests for connections
   My_Accept_Handler *eh=new My_Accept_Handler(addr);
   //Register the reactor to call back when incoming client connects
   ACE_Reactor::instance()->register_handler(eh,
                  ACE_Event_Handler::ACCEPT_MASK);
   //Start the event loop
   while(1)
      ACE_Reactor::instance()->handle_events();
}
```

In the above example, two concrete event handlers are created. The first concrete event handler, `My_Accept_Handler`, is used to accept and establish incoming connections from clients. The other event handler is `My_Input_Handler`, which is used to handle the connection after it has been established. Thus `My_Accept_Handler` accepts the connection and delegates the actual handling off to `My_Input_Handler`.



In the above example, first we create an *ACE_INET_Addr* object passing it the port on which we wish to accept connections. Next, an object of type `My_Accept_Handler` is instantiated. The address object is then passed to `My_Accept_Handler` through its constructor. `My_Accept_Handler` has an underlying "concrete acceptor" (read more about concrete acceptors in the chapter on "IPC") which it uses to establish the connection. The constructor of `My_Accept_Handler` delegates the "listening" for new connections to the *open()* method of this concrete acceptor. After the handler starts listening for connections it is registered with the reactor informing it that it is to be called back when a new connection request is received. To do this, we call *register_handler()* with the mask "*ACE_Event_Handler::ACCEPT_MASK*".

When the reactor is told to register the handler, it performs a "double dispatch" to determine the underlying handle of the event handler. To do this, it calls the *get_handle()* method. Since the reactor uses the *get_handle()* method to determine the handle to the underlying stream, it is necessary that this method be implemented in `My_Accept_Handler`. In this example, we simply call *get_handle()* on the concrete acceptor, which returns the appropriate handle to the reactor.

Once a new connection request is received on this handle, the reactor will automatically call back the *handle_input()* method `of My_Accept_Handler`. The Accept Handler then instantiates a new Input Handler and calls *accept()* on the concrete acceptor to actually establish the connection. Notice that the underlying stream of the Input Handler is passed in as the first argument to the *accept()* call. This causes the stream in the newly instantiated input handler to be set to the new stream which has just been created after establishment of the connection (by the *accept()* call). The Accept Handler then registers the Input Handler with the reactor, informing it to call back if any input is available to read (using *ACE_Event_Handler::READ_MASK*). It then returns -1, which causes it to be removed from the reactor's internal event dispatch tables.

When any input now arrives from the client, *My_Input_Handler::handle_input()* will automatically be called back by the reactor. Note that in the *handle_input()* method of `My_Input_Handler,` 0 is returned to the reactor. This indicates that we wish to keep it registered, whereas in `My_Accept_Handler` we insured that is was de-registered by returning -1 in its *handle_input()* method.

Besides the READ_MASK and ACCEPT_MASK that are used in the example above, there are several other masks that can be used when registering and removing handles from the reactor. These masks are shown in the table below, and can be used in conjunction with the *register_handler()* and *remove_handler()* methods. Each mask insures different behavior of the reactor when it calls back an event handler, usually meaning a different "handle" method is to be called.

MASK	Calls back method	When	Used with
ACE_Event_Handler:: READ_MASK	handle_input().	When there is data available to read on the handle.	register_handler()
ACE_Event_Handler:: WRITE_MASK	handle_output().	When there is room available on the I/O devices output buffer and new data can be sent to it.	register_handler()
ACE_Event_Handler:: TIMER_MASK	handle_close().	Passed to handle_close() to indicate the reason for calling it was a time-out.	Acceptor and Connectors handle_timeout methods and NOT by the Reactor.
ACE_Event_Handler:: ACCEPT_MASK	handle_input().	When a client request for a new connection is heard on the OS's internal listen queue.	register_handler()
ACE_Event_Handler:: CONNECT_MASK	handle_input().	When the connection has been established.	register_handler()
ACE_Event_Handler:: DONT_CALL	None.	Insures that the handle_close() method of the Event_Handler is NOT called when the reactor's remove_handler() method is called.	remove_handler()

Timers

The Reactor also includes methods to schedule timers, which on expiry call back the *handle_timeout()* method of the appropriate event handler. To schedule such timers, the reactor has a *schedule_timer()* method. This method is passed the event handler ,whose *handle_timeout()* method is to be called back, and the delay in the form of an *ACE_Time_Value* object. In addition, an interval may also be specified which causes the timer to be reset automatically after it expires.

Internally, the Reactor maintains an *ACE_Timer_Queue* which maintains all of the timers in the order in which they are to be scheduled. The actual data structure used to hold the timers can be varied by using the *set_timer_queue()* method of the reactor. Several different timer structures are available to use with the reactor, including timer wheels, timer heaps and hashed timer wheels. These are discussed in a later section in detail.

ACE_Time_Value

The *ACE_Time_Value* object is a wrapper class which encapsulates the data and time structure of the underlying OS platform. It is based on the *timeval* structure available on most UNIX operating systems, which stores absolute time in seconds and micro-seconds. Other OS platforms, such as POSIX and Win32, use slightly different representations. This class encapsulates these differences and provides a portable C++ interface.

The *ACE_Time_Value* class uses operator overloading, which provides for simple arithmetic additions, subtractions and comparisons. Methods in this class are implemented to “normalize” time quantities. Normalization adjusts the two fields in a timeval struct to use a canonical encoding scheme that ensures accurate comparisons. (For more see Appendix and Reference Guide).

Setting and Removing Timers

The following example illustrates how timers can be used with the reactor.

Example 3

```
#include ”test_config.h”
#include ”ace/Timer_Queue.h”
#include ”ace/Reactor.h”
#define NUMBER_TIMERS 10

static int done = 0;
static int count = 0;

class Time_Handler : public ACE_Event_Handler
{
public:
   //Method which is called back by the Reactor when timeout occurs.
   virtual int handle_timeout (const ACE_Time_Value &tv,
   const void *arg){
      long current_count = long (arg);
      ACE_ASSERT (current_count == count);
      ACE_DEBUG ((LM_DEBUG, ”%d: Timer #%d timed out at %d!\n”,
                  count, current_count, tv.sec()));
   //Increment count
      count ++;
   //Make sure assertion doesn’t fail for missing 5th timer.
      if (count ==5)
       count++;

      //If all timers done then set done flag
      if (current_count == NUMBER_TIMERS - 1)
       done = 1;
      //Keep yourself registered with the Reactor.
      return 0;
      }
};

int
main (int, char *[])
{
   ACE_Reactor reactor;
   Time_Handler *th=new Time_Handler;
   int timer_id[NUMBER_TIMERS];
   int i;
```

```
   for (i = 0; i < NUMBER_TIMERS; i++)
      timer_id[i] = reactor.schedule_timer (th,
        (const void *) i, // argument sent to handle_timeout()
        ACE_Time_Value (2 * i + 1)); //set timer to go off with delay

   //Cancel the fifth timer before it goes off
   reactor.cancel_timer(timer_id[5]);//Timer ID of timer to be removed
   while (!done)
      reactor.handle_events ();

   return 0;
}
```

In the above example, an event handler, `Time_Handler` is first set up to handle the time-outs by implementing the *handle_timeout()* method. The main routine instantiates an object of type `Time_Handler` and schedules multiple timers (10 timers) using the *schedule_timer()* method of the reactor. This method takes, as arguments, a pointer to the handler which will be called back, the time after which the timer will go off and an argument that will be sent to the *handle_timeout()* method when it is called back. Each time *schedule_timer()* is called, it returns a unique timer identifier which is stored in the array *timer_id[]*. This identifier may be used to cancel that timer at any time. An example of canceling a timer is also shown in the above example, where the fifth timer is canceled by calling the reactor's *cancel_timer()* method after all the timers have been initially scheduled. We cancel this timer by using its timer_id as an argument to the *cancel_timer()* method of the reactor.

Using different Timer Queues

Different environments may require different ways of scheduling and canceling timers. The performance of algorithms to implement timers become an issue when any of the following are true:

- Fine granularity timers are required.
- The number of outstanding timers at any one time can potentially be very large.
- The algorithm is implemented using hardware interrupts which are too expensive.

ACE allows the user to choose from among several timers which pre-exist in ACE, or to develop their own timers to an interface defined for timers. The different timers available in ACE are detailed in the following table:

Timer	Description of data structure	Performance
`ACE_Timer_Heap`	The timers are stored in a heap implementation of a priority queue.	Cost of *schedule_timer()*= O(lg n) Cost of *cancel_timer()*= O(lgn) Cost of finding current timer O(1)
`ACE_Timer_List`	The timers are stored in a doubly linked list .. insertions are..??	Cost of *schedule_timer()*= O(n) Cost of *cancel_timer()*= O(1) Cost of finding current timer O(1)
`ACE_Timer_Hash`	This structure used in this case is a variation on the timer wheel algorithm. The performance is highly dependent on the hashing function used.	Cost of *schedule_timer()*= Worst = O(n) Best = O(1) Cost of *cancel_timer()*= O(1) Cost of finding current timer O(1)

`ACE_Timer_Wheel`	The timers are stored in an array of "pointers to arrays". With each array being pointed to is sorted.	Cost of *schedule_timer()*= Worst =O(n) Cost of *cancel_timer()*= O(1) Cost of finding current timer O(1)

See reference [VII] on Timers for more.

Handling Signals

As we saw in example 1, the Reactor includes methods to allow the handling of signals. The Event Handler which handles the signals should overload the *handle_signal()* method, since this will be called back by the reactor when the signal occurs. To register for a signal, we use one of the *register_handler()* methods, as was illustrated in example 1. When interest in a certain signal ends, the handler can be removed and restored to the previously installed signal handler by calling *remove_handler()*. The Reactor internally uses the sigaction() system call to set and reset signal handlers. Signal handling can also be done without the reactor by using the *ACE_Sig_Handlers* class and its associated methods.

One important difference in using the reactor for handling signals and using the *ACE_Sig_Handlers* class is that the reactor-based mechanism only allows the application to associate one event handler with each signal. The *ACE_Sig_Handlers* class however allows multiple event handlers to be called back when a signal occurs.

Using Notifications

The reactor not only issues callbacks when system events occur, but can also call back handlers when user defined events occur. This is done through the reactor's "Notification" interface, which consists of two methods, *notify()* and *max_notify_iterations()*.

The reactor can be explicitly instructed to issue a callback on a certain event handler object by using the *notify()* method. This is very useful when the reactor is used in conjunction with message queues or with co-operating tasks. Good examples of this kind of usage can be found when the ASX framework components are used with the reactor.

The *max_notify_iterations()* method informs the reactor to perform only the specified number of "iterations" at a time. Here, "iterations" refers to the number of "notifications" that can occur in a single *handle_events()* call. Thus if *max_notify_iterations()* is used to set the max number of iterations to 20, and 25 notifications arrive simultaneously, then the *handle_events()* method will only service 20 of the notifications at a time. The remaining five notifications will be handled when *handle_events()* is called the next time in the event loop.

An example will help illustrate these ideas further:

```
Example 4
#include "ace/Reactor.h"
#include "ace/Event_Handler.h"
#include "ace/Synch_T.h"
```

```
#include "ace/Thread_Manager.h"
#define WAIT_TIME 1
#define SLEEP_TIME 2

class My_Handler: public ACE_Event_Handler{
   public:

   //Start the event handling process.
   My_Handler(){
      ACE_DEBUG((LM_DEBUG,"Event Handler created\n"));
      ACE_Reactor::instance()->max_notify_iterations(5);
      return 0;
      }
//Perform the notifications i.e., notify the reactor 10 times
   void perform_notifications(){
      for(int i=0;i<10;i++)
        ACE_Reactor::instance()->
             notify(this,ACE_Event_Handler::READ_MASK);
      }
//The actual handler which in this case will handle the notifications
   int handle_input(int){
      ACE_DEBUG((LM_DEBUG,"Got notification # %d\n",no));
      no++;
      return 0;
      }
private:
   static int no;
};

//Static members
int My_Handler::no=1;


int main(int argc, char *argv[]){
   ACE_DEBUG((LM_DEBUG,"Starting test \n"));
   //Instantiating the handler
   My_Handler handler;

   //The done flag is set to not done yet.
   int done=0;
   while(1){
      //After WAIT_TIME the handle_events will fall through if no events
      //arrive.
      ACE_Reactor::instance()->handle_events(ACE_Time_Value(WAIT_TIME));
      if(!done){
        handler.perform_notifications();
        done=1;
        }
      sleep(SLEEP_TIME);
   }
}
```

In the above example, a concrete handler is created as usual and the handle_input() method is overload, as it would be if we were expecting our handler to handle input data from an I/O device. The handler also contains an *open()* method, which performs initialization for the handler, and a method which actually performs the notifications.

In the *main()* function, we first instantiate an instance of our concrete handler. The constructor of the handler insures that the number of *max_notify_iterations* is set to 5 by using the *max_notify_iterations()* method of the reactor. After this, the reactor's event handling loop is started.

One major difference in the event-handling loop to be noted here is that *handle_events()* is passed an *ACE_Time_Value*. If no events occur within this time, then the *handle_events()* method will fall through. After *handle_events()* falls through, *perform_notifications()* is called, which uses the reactor's *notify()* method to request it to notify the handler that is passed in as an argument of the occurrence of an event. The reactor will then proceed to use the mask that it is passed to perform an upcall on the appropriate "handle" method of the handler. In this case, we use *notify()* to inform our event handler of input by passing it the *ACE_Event_Handler::READ_MASK*. This causes the reactor to call back the *handle_input()* method of the handler.

Since we have set the *max_notify_iterations* to 5 therefore only 5 of the notifications will actually be issued by the reactor during one call to *handle_events()*. To make this clear, we stop the reactive event loop for SLEEP_TIME before issuing the next call to *handle_events()*.

The above example is overly simplistic and very non-realistic, since the notifications occur in the same thread as the reactor. A more realistic example would be of events which occur in another thread and which then notify the reactor thread of these events. The same example with a different thread to perform the notifications is shown below:

Example 5

```
#include "ace/Reactor.h"
#include "ace/Event_Handler.h"
#include "ace/Synch_T.h"
#include "ace/Thread_Manager.h"

class My_Handler: public ACE_Event_Handler{
public:
   //Start the event handling process.
   My_Handler(){
      ACE_DEBUG((LM_DEBUG,"Got open\n"));
      activate_threads();
      ACE_Reactor::instance()->max_notify_iterations(5);
      return 0;
      }
   //Spawn a separate thread so that it notifies the reactor
   void activate_threads(){
      ACE_Thread_Manager::instance()
                  ->spawn((ACE_THR_FUNC)svc_start,(void*)this);
      }
```

```
   //Notify the Reactor 10 times.
   void svc(){
      for(int i=0;i<10;i++)
       ACE_Reactor::instance()->
            notify(this, ACE_Event_Handler::READ_MASK);
       }
   //The actual handler which in this case will handle the notifications
   int handle_input(int){
      ACE_DEBUG((LM_DEBUG, ”Got notification # %d\n”, no));
      no++;
      return 0;
      }

   //The entry point for the new thread that is to be created.
   static int svc_start(void* arg);
private:
   static int no;
};

//Static members
int My_Handler::no=1;
int My_Handler::svc_start(void* arg){
   My_Handler *eh= (My_Handler*)arg;
   eh->svc();
   return -1; //de-register from the reactor
   }

int main(int argc, char *argv[]){
   ACE_DEBUG((LM_DEBUG,”Starting test \n”));
   My_Handler handler;

   while(1){
      ACE_Reactor::instance()->handle_events();
      sleep(3);
      }
}
```

This example is very similar to the previous example, except for a few additional methods to spawn a thread and then activate it in the event handler. In particular, the constructor of the concrete handler `My_Handler` calls the activate method. This method uses the *ACE_Thread_Manager::spawn()* method to spawn a separate thread with its entry point as *svc_start()*.

The *svc_start()* method calls *perform_notifications()* and the notifications are sent to the reactor, but this time they are sent from this new thread instead of from the same thread that the reactor resides in. Note that the entry point of the thread, *svc_start()*,was defined as a static method in the function, which then called the non-static *svc()* method. This is a requirement when using thread libraries, i.e. the entry point of a thread be a static function with file scope.

Chapter

7

The Acceptor and Connector

Patterns for connection establishment

The Acceptor/Connector pattern has been designed to decouple connection establishment from the service which is performed after the connection has been established. For example, in a WWW browser, the service or “actual work” performed is the parsing and displaying of the HTML page that has been received by the client browser. Connection establishment may be of secondary significance, and is probably done through the BSD sockets interface or some other equivalent IPC mechanism. The usage of these patterns will allow the programmer to focus on this actual “work” with minimal concern as to how the connection between the server and client is actually made. On the flip side, the programmer can also fine tune the connection establishment policies independent of the service routines he may have written or is about to write.

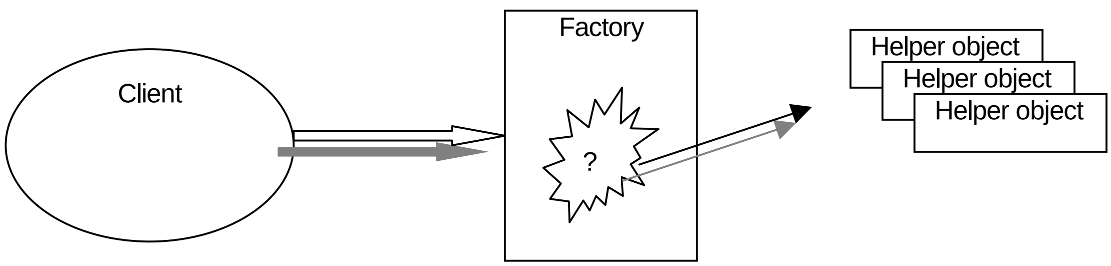
Since this pattern decouples the service from the connection establishment method, it is very easy to change either one without affecting the other. This allows for code re-use both of pre-written connection establishment mechanisms and of pre-written service routines. In the same example, the browser programmer using these patterns could initially build his system and run it using a certain connection establishment mechanism and test it. Later, he may decide that he wishes to change the underlying connection mechanism to be multi-threaded, using a thread pool policy perhaps, if the previous connection mechanism proved undesirable for the browser he has built. Using this pattern, this could be achieved with a minimal amount of effort, due to the strict de-coupling it provides.

You need to read:

You will have to read through the chapter on the Reactor and on IPC_SAP (in particular the section on acceptors and connectors) before you will be able to clearly understand many of the examples shown in this chapter, and especially to understand the more advanced section. In addition, you may have to refer to the section on threads and thread management.

THE ACCEPTOR PATTERN

An acceptor is usually used where you would imagine you would use the BSD *accept()* system call, as was discussed in the chapter on stand-alone acceptors. The Acceptor Pattern is also applicable in the same situation, but as we will see, provides a lot more functionality. In ACE, the acceptor pattern is implemented with the help of a "Factory" which is named *ACE_Acceptor*. A factory is a class which is used to abstract the instantiation process of helper objects (usually). It is common in OO designs for a complex class to delegate certain functions to a helper class. The choice of which class the complex class creates as a helper and then delegates to may have to be flexible. This flexibility is afforded with the help of a factory. Thus a factory allows an object to change its underlying strategies by changing the object that it delegates the work to. The factory, however, provides the same interface to the applications which are using the factory, and thus the client code may not need to be changed at all. (Read more about factories in the reference on "Design Patterns").



The *ACE_Acceptor* factory allows an application developer to change the "helper" objects used for:

- Passive Connection Establishment
- Handling of the connection after establishment

Similarly, the *ACE_Connector* factory allows an application developer to change the "helper" objects used for:

- Active Connection Establishment
- Handling of the connection after establishment

The following discussion applies equally to acceptors and connectors, so I will just talk about acceptors here and the argument will hold equally well for connectors.

The *ACE_Acceptor* has been implemented as a template container, which is instantiated with two classes as its actual parameters. The first implements the particular service (and is of type *ACE_Event_Handler*, since it is to be used to handle I/O events and must be from the event handling hierarchy of classes) that the application is to perform after it establishes its connection, and the second is a "concrete" acceptor (of the variety that was discussed in the chapter on IPC_SAP).

An important point to note is that the *ACE_Acceptor* factory and the underlying concrete acceptor that is used are both very different things. The concrete acceptor can be used independently of the *ACE_Acceptor* factory without having anything to do with the acceptor pattern that we are discussing here. (The independent usage of acceptors was discussed and illustrated in the chapter on IPC_SAP). The *ACE_Acceptor* factory is intrinsic to this pattern and cannot be used without an underlying concrete acceptor. Thus *ACE_Acceptor* **USES** the underlying concrete acceptors to establish connections. As we have seen, there are several classes which come bundled with ACE which can be used as the second parameter (i.e., the concrete acceptor class) to the *ACE_Acceptor* factory template. However the service handling class must be implemented by the application developer and must be of type *ACE_Event_Handler*. The *ACE_Acceptor factory* could be instantiated as:

```
typedef ACE_Acceptor<My_Service_Handler,ACE_SOCK_ACCEPTOR> MyAcceptor;
```

Here `MyAcceptor` has been passed the event handler called `My_Service_Handler` and the concrete acceptor *ACE_SOCK_ACCEPTOR*. *ACE_SOCK_ACCEPTOR* is a TCP acceptor based on the BSD sockets stream family. (See table at the end of this section and in IPC chapter for different types of acceptors that can be passed to the acceptor factory). Note once again that, when using the acceptor pattern, we always deal with two acceptors. The factory acceptor called *ACE_Acceptor* and one of the concrete acceptors which are available in ACE, such as *ACE_SOCK_ACCEPTOR*. (You can create custom concrete acceptors which would replace *ACE_SOCK_ACCEPTOR* but you probably won't change anything in the *ACE_Acceptor* factory class).

> **Important Note:** *ACE_SOCK_ACCEPTOR* is actually a macro defined as:
>
> ```
> #define ACE_SOCK_ACCEPTOR ACE_SOCK_Acceptor, ACE_INET_Addr
> ```
>
> The use of this macro was deemed necessary since *typedef*s inside a class don't work for compilers on certain platforms. If this hadn't been the case, then *ACE_Acceptor* would have been instantiated as:
>
> ```
> typedef ACE_Acceptor<My_Service_Handler,ACE_SOCK_Acceptor>
> MyAcceptor;
> ```
>
> The macros are illustrated in the table at the end of this section.

COMPONENTS

As is clear from the above discussion, there are three major participant classes in the Acceptor pattern:

- The **concrete acceptor,** which contains a specific strategy for establishing a connection which is tied to an underlying transport protocol mechanism. Examples of different concrete acceptors that can be used in ACE are *ACE_SOCK_ACCEPTOR* (uses TCP to establish the connection), *ACE_LSOCK_ACCEPTOR* (uses UNIX domain sockets to establish the connection), etc.
- The **concrete service handler,** which is written by the application developer and whose *open()* method is called back automatically when the connection has been

established. The acceptor framework assumes that the service handling class is of type *ACE_Event_Handler*, which is an interface class defined in ACE (This class was discussed in detail in the chapter on the Reactor). Another class which has been exclusively created for service handling for the acceptor and connector patterns is *ACE_Svc_Handler*. This class is based not only on the *ACE_Event_Handler* interface (which is necessary to use the Reactor), but also on the *ACE_Task* classes, which are used in the ASX Streams Framework. The *ACE_Task* classes provide the ability to create separate threads, use message queues to store incoming data messages, to process them concurrently and several other useful functions. This extra functionality can be obtained if the concrete service handler that is used with the acceptor pattern is created by deriving from *ACE_Svc_Handler* instead of *ACE_Event_Handler*. The usage of the extra functionality available in *ACE_Svc_Handler* is discussed in detail in the advanced sections of this chapter. In the following discussion we will use the *ACE_Svc_Handler* as our event handler. One important difference between a simple *ACE_Event_Handler* and the *ACE_Svc_Handler* class is that the service handler contains an underlying communication stream component (of the variety discussed in the IPC SAP chapter). This stream is set when the *ACE_Svc_Handler* template is instantiated. In the case of *ACE_Event_Handler*, we had to add the I/O communication endpoint (i.e. the stream object) as a private data member of the event handler ourselves. Thus, in this case the application developer creates his service handler as a subclass of the *ACE_Svc_Handler* class and implements the *open()* method as the first method which will be called back automatically by the framework. In addition, since *ACE_Svc_Handler* is a template, the communication stream component and the locking mechanism are passed in as template parameters.

- The **reactor,** which is used in conjunction with the *ACE_Acceptor*. As we will see, after instantiating the acceptor we start the reactor's event handling loop. The reactor, as explained earlier, is an event-dispatching class and in this case is used by the acceptor to handle the dispatching of the connection establishment event to the appropriate service handling routine.

USAGE

Further understanding of the Acceptor can be gained by looking at a simple example. This example is of a simple application which uses the acceptor to accept connections and then call back the service routine. When the service routine is called back, it just lets the user know that the connection has been established successfully.

Example 1

```
#include ”ace/Reactor.h”
#include ”ace/Svc_Handler.h”
#include ”ace/Acceptor.h”
#include ”ace/Synch.h”
#include ”ace/SOCK_Acceptor.h”
```

```
//Create a Service Handler whose open() method will be called back
//automatically. This class MUST derive from ACE_Svc_Handler which is an
//interface and as can be seen is a template container class itself. The
//first parameter to ACE_Svc_Handler is the underlying stream that it
//may use for communication. Since we are using TCP sockets the stream
//is ACE_SOCK_STREAM. The second is the internal synchronization
//mechanism it could use. Since we have a single threaded application we
//pass it a "null" lock which will do nothing.

class My_Svc_Handler:
 public ACE_Svc_Handler <ACE_SOCK_STREAM,ACE_NULL_SYNCH>{
//the open method which will be called back automatically after the
//connection has been established.

public:
int open(void*){
   cout<<"Connection established"<<endl;
   }
};
// Create the acceptor as described above.
typedef ACE_Acceptor<My_Svc_Handler,ACE_SOCK_ACCEPTOR> MyAcceptor;

int main(int argc, char* argv[]){
//create the  address on which we wish to connect. The constructor takes
//the port number on which to listen and will automatically take the
//host's IP address as the IP Address for the addr object

ACE_INET_Addr addr(PORT_NUM);

//instantiate the appropriate acceptor object with the address on which
//we wish to accept and the Reactor instance we want to use. In this
//case we just use the global ACE_Reactor singleton. (Read more about
//the reactor in the previous chapter)
MyAcceptor acceptor(addr, ACE_Reactor::instance());

while(1)
   // Start the reactor's event loop
   ACE_Reactor::instance()->handle_events();
}
```

In the above example, we first create an endpoint address on which we wish to accept. Since we have decided to use TCP/IP as the underlying connection protocol, we create an *ACE_INET_Addr* as our endpoint and pass it the port number we want it to listen on. We pass this address and an instance of the reactor singleton to the acceptor that we instantiate after this. This acceptor, after being instantiated, will automatically accept any connection requests on `PORT_NUM` and call back `My_Svc_Handler`'s *open()* method after establishing connections for such requests. Notice that when we instantiated the *ACE_Acceptor* factory, we passed it the concrete acceptor we wanted to use, i.e. *ACE_SOCK_ACCEPTOR*, and the concrete service handler we wanted to use, i.e. `My_Svc_Handler`.

Now let’s try something a bit more interesting. In the next example, we will register our service handler back with the reactor after it is called back on connection establishment. Now, if any data comes on the newly created connection, then our service handling routines *handle_input()* method would be called back automatically. Thus in this example, we are using the features of both the reactor and acceptor together:

Example 2

```
#include ”ace/Reactor.h”
#include ”ace/Svc_Handler.h”
#include ”ace/Acceptor.h”
#include ”ace/Synch.h”
#include ”ace/SOCK_Acceptor.h”
#define PORT_NUM 10101
#define DATA_SIZE 12

//forward declaration
class My_Svc_Handler;

//Create the Acceptor class
typedef ACE_Acceptor<My_Svc_Handler,ACE_SOCK_ACCEPTOR>
MyAcceptor;

//Create a service handler similar to as seen in example 1. Except this
//time include the handle_input() method which will be called back
//automatically by the reactor when new data arrives on the newly
//established connection
class My_Svc_Handler:
 public ACE_Svc_Handler <ACE_SOCK_STREAM,ACE_NULL_SYNCH>{
public:
My_Svc_Handler(){
   data= new char[DATA_SIZE];
   }
int open(void*){
   cout<<”Connection established”<<endl;

   //Register the service handler with the reactor
   ACE_Reactor::instance()->register_handler(this,
   ACE_Event_Handler::READ_MASK);
   return 0;
   }

int handle_input(ACE_HANDLE){
   //After using the peer() method of ACE_Svc_Handler to obtain a
   //reference to the underlying stream of the service handler class
   //we call recv_n() on it to read the data which has been received.
   //This data is stored in the data array and then printed out
   peer().recv_n(data,DATA_SIZE);
   ACE_OS::printf(”<< %s\n”,data);

   //keep yourself registered with the reactor
   return 0;
   }
```

```
private:
   char* data;
};

int main(int argc, char* argv[]){
   ACE_INET_Addr addr(PORT_NUM);
   //create the acceptor
   MyAcceptor acceptor(addr, //address to accept on
      ACE_Reactor::instance()); //the reactor to use

while(1)
   //Start the reactor’s event loop
   ACE_Reactor::instance()->handle_events();
}
```

The only difference between this example and the previous one is that we register the service handler with the reactor in the *open()* method of our service handler. We consequently have to write a *handle_input()* method which will be called back by the reactor when data comes in on the connection. In this case we just print out the data we receive on the screen. The *peer()* method of the *ACE_Svc_Handler* class is a useful method which returns a reference to the underlying peer stream. We use the *recv_n()* method of the underlying stream wrapper class to obtain the data received on the connection.

The real power of this pattern lies in the fact that the underlying connection establishment mechanism is fully encapsulated in the concrete acceptor. This can very easily be changed. In the next example, we change the underlying connection establishment mechanism so that it uses UNIX domain sockets instead of TCP sockets, as we were using before. The example, again with minimal changes (underlined), is as follows:

Example3

```
class My_Svc_Handler:
public ACE_Svc_Handler <ACE_LSOCK_STREAM,ACE_NULL_SYNCH>{
public:
   int open(void*){
      cout<<”Connection established”<<endl;
      ACE_Reactor::instance()
            ->register_handler(this,ACE_Event_Handler::READ_MASK);
      }

   int handle_input(ACE_HANDLE){
      char* data= new char[DATA_SIZE];
      peer().recv_n(data,DATA_SIZE);
      ACE_OS::printf(”<< %s\n”,data);
      return 0;
      }
};

typedef ACE_Acceptor<My_Svc_Handler,ACE_LSOCK_ACCEPTOR> MyAcceptor;
```

```
int main(int argc, char* argv[]){
   ACE_UNIX_Addr addr("/tmp/addr.ace");
   MyAcceptor acceptor(address, ACE_Reactor::instance());
   while(1) /* Start the reactor's event loop */
      ACE_Reactor::instance()->handle_events();
}
```

The differences between example 2 and example 3 are underlined. As noted, the differences between the two programs are very minimal. However they both use very different connection establishment paradigms. Some of the connection establishment mechanisms that are available in ACE are listed in the table below.

Type of Acceptor	Address used	Stream used	Concrete Acceptor
TCP stream Acceptor	ACE_INET_Addr	ACE_SOCK_STREAM	ACE_SOCK_ACCEPTOR
UNIX domain local stream socket acceptor	ACE_UNIX_Addr	ACE_LSOCK_STREAM	ACE_LSOCK_ACCEPTOR
PIPES as the underlying communication mechanism	ACE_SPIPE_Addr	ACE_SPIPE_STREAM	ACE_SPIPE_ACCEPTOR

THE CONNECTOR

The Connector is very similar to the Acceptor. It is also a factory, but in this case it is used to *actively* connect to a remote host. After the connection has been established, it will automatically call back the *open()* method of the appropriate service handling object. The connector is usually used where you would use the BSD *connect()* call. In ACE, the connector, just like the acceptor, is implemented as a template container class called *ACE_Connector*. As mentioned earlier, it takes two parameters, the first being the event handler class which is to be called when the connection is established and the second being a "concrete" connector class .

You **MUST** note that the underlying concrete connector and the factory *ACE_Connector* are both very different things. The *ACE_Connector* factory USES the underlying concrete connector to establish the connection. The *ACE_Connector* factory then USES the appropriate event or service handling routine (the one passed in through its template argument) to handle the new connection after the connection has been established by the concrete connector. The concrete connectors can be used without the *ACE_Connector* factory as we saw in the IPC chapter. The *ACE_Connector* factory, however, cannot be used without a concrete connector class (since it is this class which actually handles connection establishment).

An example of instantiating the *ACE_Connector* class is:

```
typedef ACE_Connector<My_Svc_Handler,ACE_SOCK_CONNECTOR> MyConnector;
```

The second parameter in this case is the concrete connector class *ACE_SOCK_CONNECTOR*. The connector, like the acceptor pattern, uses the reactor internally to call back the *open()* method of the service handler when the connection has been established. We can reuse the service handling routines we had written for the previous examples with the connector.

An example using the connector should make this clearer.

```
Example 4
typedef ACE_Connector<My_Svc_Handler,ACE_SOCK_CONNECTOR> MyConnector;

int main(int argc, char * argv[]){
   ACE_INET_Addr addr(PORT_NO,HOSTNAME);
   My_Svc_Handler * handler= new My_Svc_Handler;
   //Create the connector
   MyConnector connector;

   //Connects to remote machine
   if(connector.connect(handler,addr)==-1)
      ACE_ERROR(LM_ERROR,"%P|%t, %p","Connection failed");

   //Registers with the Reactor
   while(1)
      ACE_Reactor::instance()->handle_events();
}
```

In the above example, `PORT_NO` and `HOSTNAME` are the machine and port we wish to actively connect to. After instantiating the connector, we call its connect method, passing it the service routine that is to be called back when the connection is fully established, and the address that we wish to connect to.

USING THE ACCEPTOR AND CONNECTOR TOGETHER

The Acceptor and Connector patterns will, in general, be used together. In a client-server application, the server will typically contain the acceptor, whereas a client will contain the connector. However, in certain applications, both the acceptor and connector may be used together. An example of such an application is given below. In this example, a single message is repeatedly sent to the peer machine, and at the same time another message is received from the remote. Since two functions must be performed at the same time, an easy solution is to send and receive messages in separate threads.

This example contains both an acceptor and a connector. The user can take arguments at the command prompt and tell the application whether it is going to play a server or client role. The application will then call `main_accept()` or `main_connect()` as appropriate.

```
Example 5
#include "ace/Reactor.h"
#include "ace/Svc_Handler.h"
#include "ace/Acceptor.h"
```

```
#include ”ace/Synch.h”
#include ”ace/SOCK_Acceptor.h”
#include ”ace/Thread.h”

//Add our own Reactor singleton
typedef ACE_Singleton<ACE_Reactor,ACE_Null_Mutex> Reactor;
//Create an Acceptor
typedef ACE_Acceptor<MyServiceHandler,ACE_SOCK_ACCEPTOR> Acceptor;
//Create a Connector
typedef ACE_Connector<MyServiceHandler,ACE_SOCK_CONNECTOR> Connector;

class MyServiceHandler:
public ACE_Svc_Handler<ACE_SOCK_STREAM,ACE_NULL_SYNCH>{
public:
   //Used by the two threads “globally” to determine their peer stream
   static ACE_SOCK_Stream* Peer;
   //Thread ID used to identify the threads
   ACE_thread_t t_id;
   int open(void*){
      cout<<”Acceptor: received new connection”<<endl;
      //Register with the reactor to remember this handle
      Reactor::instance()
        ->register_handler(this,ACE_Event_Handler::READ_MASK);

      //Determine the peer stream and record it globally
      MyServiceHandler::Peer=&peer();

      //Spawn new thread to send string every second
      ACE_Thread::spawn((ACE_THR_FUNC)send_data,0,THR_NEW_LWP,&t_id);

      //keep the service handler registered by returning 0 to the
      //reactor
      return 0;
      }

   static void* send_data(void*){
      while(1){
        cout<<”>>Hello World”<<endl;
        Peer->send_n(”Hello World”,sizeof(”Hello World”));

        //Go to sleep for a second before sending again
        ACE_OS::sleep(1);
             }
      return 0;
      }

   int handle_input(ACE_HANDLE){
      char* data= new char[12];

      //Check if peer aborted the connection
      if(Peer.recv_n(data,12)==0){
        cout<<”Peer probably aborted connection”);
        ACE_Thread::cancel(t_id); //kill sending thread ..
        return -1; //de-register from the Reactor.
```

```
        }

      //Show what you got..
      cout<<"<< %s\n",data"<<endl;

      //keep yourself registered
      return 0;
        }
};

//Global stream identifier used by both threads
ACE_SOCK_Stream * MyServiceHandler::Peer=0;


void main_accept(){
   ACE_INET_Addr addr(PORT_NO);
   Acceptor myacceptor(addr,Reactor::instance());
   while(1)
      Reactor::instance()->handle_events();

   return 0;
   }

void main_connect(){
   ACE_INET_Addr addr(PORT_NO,HOSTNAME);
   Connector myconnector;
   myconnector.connect(my_svc_handler,addr);
   while(1)
      Reactor::instance()->handle_events();

   }


int main(int argc, char* argv[]){
   // Use ACE_Get_Opt to parse and obtain arguments and then call the
   // appropriate function for accept or connect.
   ...
}
```

This is a simple example which illustrates how the acceptor and connector patterns can be used in combination to produce a service handling routine which is completely decoupled from the underlying network establishment method. The above example can be easily changed to use any other underlying network establishment protocol, by changing the appropriate template parameters for the concrete connector and acceptor.

Advanced Sections

The following sections give a more detailed explanation of how the Acceptor and Connector patterns actually work. This is required if you wish to tune the service handling and connection establishment policies. This includes tuning the creation and concurrency strategy of your service handling routine and the connection establishment strategy that the underlying concrete connector will use. In addition, there is a section which explains how you can use the advanced features you automatically get by using the *ACE_Svc_Handler* classes. Lastly, we show how you can use a simple lightweight*ACE_Event_Handler* with the acceptor and connector patterns.

THE ACE_SVC_HANDLER CLASS

The *ACE_Svc_Handler* class, as mentioned above, is based both on *ACE_Task*, which is a part of the ASX Streams framework, and on the *ACE_Event_Handler* interface class. Thus an *ACE_Svc_Handler* is both a Task and an Event Handler. Here we will give you a brief introduction to *ACE_Task* and what you can do with *ACE_Svc_Handler*.

ACE_Task

ACE_Task has been designed to be used with the ASX Streams framework, which is based on the streams facility in UNIX System V. ASX is also very similar in design to the X-kernel protocol tools built by Larry Peterson [[VIII]].

The basic idea in ASX is that an incoming message is assigned to a *stream*. This stream is constructed out of several *modules.* Each module performs some fixed function on the incoming message, which is then passed on to the next module for further processing until it reaches the end of the stream. The actual processing in the module is done by *tasks*. There are usually two tasks to each module, one for processing incoming messages and one for processing outgoing messages. This kind of a structure is very useful when constructing protocol stacks. Since each module used has a fixed simple interface, modules can be created and easily re-used across different applications. For example, consider an application which is to process incoming messages from the data link layer. The programmer would construct several modules, each dealing with a different level of the protocol processing. Thus, he would construct a separate module to do network layer processing, another for transport layer processing and still another for presentation layer processing. After constructing these modules they can be *chained* together into a stream (with the help of ASX) and used. At a later time, if a new (and perhaps better) transport module is created, then the earlier transport module can be replaced in the stream without affecting anything else in the program. Note that the module is like a container which contains tasks. The tasks are the actual processing elements. A module may need to have two tasks, as in the example above, or may just need one. *ACE_Task*, as you may have guessed, is the implementation of the processing elements in the modules which are called tasks.

An Architecture: Communicating Tasks

Each *ACE_Task* has an internal message queue which is its means of communicating with other tasks, modules or the outside world. If one *ACE_Task* wishes to send a message to another task, it will enqueue the message on the destination tasks message queue. Once the task receives the message, it will immediately begin processing it.

Every *ACE_Task* can run as zero or more threads. Messages can be enqueued and dequeued by multiple threads on an *ACE_Task*'s message queue without the programmer worrying about corrupting any of the data structures. Thus tasks may be used as the fundamental architectural component of a system of co-operating threads. Each thread of control can be encapsulated in an *ACE_Task,* which interacts with other tasks by sending messages to their message queues, which they process and then respond to.

The only problem with this kind of architecture is that tasks can only communicate with each other through their message queues within the same process. The *ACE_Svc_Handler* solves this problem. *ACE_Svc_Handler* inherits from both *ACE_Task* and *ACE_Event_Handler*, and adds a private data stream. This combination makes it possible for an *ACE_Svc_Handler* object to act as a task that has the ability to react to events and to send and receive data between remote tasks on remote hosts.

ACE_Task has been implemented as a template container, which is instantiated with a locking mechanism, the lock being used to insure the integrity of the internal message queue in a multi-threaded environment. As mentioned earlier, *ACE_Svc_Handler* is also a template container which is passed not only the locking mechanism, but also the underlying data stream that it will use for communication to remote tasks.

Creating an ACE_ Svc_Handler

The *ACE_Svc_Handler* template is instantiated to create the desired service handler by passing in the locking mechanism and the underlying stream. If no lock is to be used, as would be done if the application was only single threaded, it can be instantiated with *ACE_NULL_SYNCH*, as we did above. However, if we intend to use it in a multi-threaded application (which is the common case), it would be done as:

```
class MySvcHandler:
public ACE_Svc_Handler<ACE_SOCK_STREAM,ACE_MT_SYNCH>{
   ...
   }
```

Creating multiple threads in the Service Handler

In Example 5 above, we created a separate thread to send data to the remote peer using the *ACE_Thread* wrapper class and its static *spawn()* method. When we did this, however, we had to define the `send_data()` method, which was written at file scope using the C++ static specifier. The consequence of this, of course, was that we couldn't access any data members of the actual object we had instantiated. In other words, we were forced to make the `send_data()` member function class-wide when this was **NOT** what we wanted to do. The only reason this was done was because *ACE_Thread::spawn()* can only use a static member function as the entry point for the thread it creates. Another adverse side affect was that a reference to the peer stream had to be made static also. In short, this wasn't the best way this code could have been written.

ACE_Task provides a nice mechanism to avoid this problem. Each *ACE_Task* has an *activate()* method which can be called to create threads for the *ACE_Task*. The entry point of the created thread will be in the non-static member function *svc()*. Since *svc()* is a non-static member function, it can call any object instance-specific data or member functions. ACE hides all the nitty-gritty of how this is done from the programmer. The *activate()* method is very versatile. It allows the programmer to create multiple threads, all of which use the *svc()* method as their entry point. Thread priorities, handles, names, etc. can also be set. The method prototype for activate is:

```
// = Active object activation method.
  virtual int activate (long flags = THR_NEW_LWP,
                        int n_threads = 1,
                        int force_active = 0,
                        long priority = ACE_DEFAULT_THREAD_PRIORITY,
                        int grp_id = -1,
                        ACE_Task_Base *task = 0,
                        ACE_hthread_t thread_handles[] = 0,
                        void *stack[] = 0,
                        size_t stack_size[] = 0,
                        ACE_thread_t  thread_names[] = 0);
```

The first parameter, flags, describes desired properties of the threads which are to be created. These are described in detail on the chapter on threads. The possible flags here are:

```
  THR_CANCEL_DISABLE, THR_CANCEL_ENABLE, THR_CANCEL_DEFERRED,
  THR_CANCEL_ASYNCHRONOUS, THR_BOUND, THR_NEW_LWP, THR_DETACHED,
  THR_SUSPENDED, THR_DAEMON, THR_JOINABLE, THR_SCHED_FIFO,
  THR_SCHED_RR, THR_SCHED_DEFAULT
```

The second parameter, *n_threads*, specifies the number of threads to be created. The third parameter, *force_active*, is used to specify whether new threads should be created, even if the *activate()* method has already been called previously, and thus the task or service handler is already running multiple threads. If this is set to false (0), then if *activate()* is called again, it will result in the failure code being set and no further threads will be spawned.

The fourth parameter is used to set the priority of the running threads. By default, or if the priority is set to *ACE_DEFAULT_THREAD_PRIORITY*, an "appropriate" priority value for the given scheduling policy (specified in flags e.g., THR_SCHED_DEFAULT) is used. This value is calculated dynamically, and is the median value between the minimum and maximum priority values for the given policy. If an explicit value is given, it is used. Note that actual priority values are EXTREMELY implementation-dependent, and are probably best avoided. More can be read on priorities of threads on the chapter on threads.

Thread Handles, Thread names and stack spaces for the threads to be created can be passed in, to be used by the thread creation calls. If these are null they are not used.

However if multiple threads are created using activate, it will be necessary to pass either names or handles for the threads before they can be used effectively.

An example will help in further understanding how the activate method may be used:

Example 6

```
#include ”ace/Reactor.h”
#include ”ace/Svc_Handler.h”
#include ”ace/Acceptor.h”
#include ”ace/Synch.h”
#include ”ace/SOCK_Acceptor.h”

class MyServiceHandler; //forward declaration
typedef ACE_Singleton<ACE_Reactor,ACE_Null_Mutex> Reactor;
typedef ACE_Acceptor<MyServiceHandler,ACE_SOCK_ACCEPTOR> Acceptor;

class MyServiceHandler:
public ACE_Svc_Handler<ACE_SOCK_STREAM,ACE_MT_SYNCH>{
// The two thread names are kept here
ACE_thread_t thread_names[2];

public:
int open(void*){
   ACE_DEBUG((LM_DEBUG, ”Acceptor: received new connection \n”));

   //Register with the reactor to remember this handler..
   Reactor::instance()
        ->register_handler(this,ACE_Event_Handler::READ_MASK);
   ACE_DEBUG((LM_DEBUG,”Acceptor: ThreadID:(%t) open\n”));

   //Create two new threads to create and send messages to the
   //remote machine.
   activate(THR_NEW_LWP,
       2, //2 new threads
       0, //force active false, if already created don’t try again.
       ACE_DEFAULT_THREAD_PRIORITY,//Use default thread priority
       -1,
       this,//Which ACE_Task object to create? In this case this one.
       0,// don’t care about thread handles used
       0,// don’t care about where stacks are created
       0,//don’t care about stack sizes
       thread_names); // keep identifiers in thread_names

   //keep the service handler registered with the acceptor.
   return 0;
   }

void send_message1(void){
   //Send message type 1
   ACE_DEBUG((LM_DEBUG,”(%t)Sending message::>>”));

   //Send the data to the remote peer
```

```
   ACE_DEBUG((LM_DEBUG,"Sent message1"));
   peer().send_n("Message1",LENGTH_MSG_1);
   } //end send_message1

int send_message2(void){
   //Send message type 1
   ACE_DEBUG((LM_DEBUG,"(%t)Sending message::>>"));

   //Send the data to the remote peer
   ACE_DEBUG((LM_DEBUG,"Sent Message2"));
   peer().send_n("Message2",LENGTH_MSG_2);
   }//end send_message_2

int svc(void){
   ACE_DEBUG( (LM_DEBUG,"(%t) Svc thread \n"));

   if(ACE_Thread::self()== thread_names[0])
      while(1) send_message1(); //send message 1s forever
   else
      while(1) send_message2(); //send message 2s forever
   return 0; // keep the compiler happy.
}

int handle_input(ACE_HANDLE){
   ACE_DEBUG((LM_DEBUG,"(%t) handle_input ::"));
   char* data= new char[13];

   //Check if peer aborted the connection
   if(peer().recv_n(data,12)==0){
      printf("Peer probably aborted connection");
      return -1; //de-register from the Reactor.
      }

   //Show what you got..
   ACE_OS::printf("<< %s\n",data);

   //keep yourself registered
   return 0;

   }
};

int main(int argc, char* argv[]){
   ACE_INET_Addr addr(10101);
   ACE_DEBUG((LM_DEBUG,"Thread: (%t) main"));

   //Prepare to accept connections
   Acceptor myacceptor(addr,Reactor::instance());

   // wait for something to happen.
   while(1)
      Reactor::instance()->handle_events();
```

```
    return 0;
}
```

In this example, *activate()* is called after the service handler is registered with the reactor in its *open()* method. It is used to create 2 threads. The names of the threads are remembered so that when they call the *svc()* routine, we can distinguish between them. Each thread sends a different type of message to the remote peer. Notice that in this case the thread creation is totally transparent. In addition, since the entry point is a normal non-static member function, it is used without any ugly changes to remember data members, such as the peer stream. We can simply call the member function *peer()* to obtain the underlying stream whenever we need it.

Using the message queue facilities in the Service Handler

As mentioned before, the *ACE_Svc_Handler* class has a built in message queue. This message queue is used as the primary communication interface between an *ACE_Svc_Handler* and the outside world. Messages that other tasks wish to send to the service handler are enqueued into its message queue. These messages may then be processed in a separate thread (created by calling the *activate()* method). Yet another thread may then take the processed message and send it across the network to a different remote destination (quite possibly to another *ACE_Svc_Handler*).

As mentioned earlier, in this multi-threaded scenario the *ACE_Svc_Handler* will automatically ensure that the integrity of the message queue is maintained with the use of locks. The lock used will be the same lock which was passed when the concrete service handler was created by instantiating the *ACE_Svc_Handler* template class. The reason that the locks are passed in this way is so that the programmer may "tune" his application. Different locking mechanisms on different platforms have different amounts of overhead. If required, the programmer may create his own optimized lock, which obeys the ACE interface for a lock and use this lock with the service handler. This is just another example of the kind of flexibility that the programmer can achieve by using ACE. The important thing that the programmer MUST be aware of is that additional threads in the service handling routines WILL cause significant locking overhead. To keep this overhead down to a minimum, the programmer must design his program carefully, ensuring that such overhead is minimized. In particular, the example described above probably will have excessive overhead and may be infeasible in most situations.

ACE_Task and thus *ACE_Svc_Handler* (as the service handler is a type of task) has several methods which can be used to set, manipulate, enqueue and dequeue from the underlying queue. We will discuss only a few of these methods here. Since a pointer to the message queue itself can be obtained in the service handler (by using the *msg_queue()* method), all public methods on the underlying queue (i.e. *ACE_Message_Queue*) may also be invoked directly. (For further details on all the methods the message queue provides, please see the next chapter on message queues.)

The underlying message queue for the service handler, as mentioned above, is an instance of *ACE_Message_Queue*, which is created automatically by the service handler. In most cases it is not necessary to call the underlying methods of *ACE_Message_Queue*, as most of them have wrappers in the *ACE_Svc_Handler* class. An *ACE_Message_Queue* is a queue which enqueues and dequeues *ACE_Message_Blocks*. Each of these *ACE_Message_Blocks* contains a pointer to a reference-counted *ACE_Data_Block*, which in turn points to the actual data stored in the block. (See next chapter on Message Queues). This allows for easy sharing of the data between *ACE_Message_Blocks*.

The main purpose of *ACE_Message_Blocks* is to allow efficient manipulation of data without much copying overhead. A read pointer and write pointer are provided with each message block. The read pointer will be incremented forward in the data block whenever we read from the block. Similarly, the write pointer moves forward when we write into the block, much like it would in a stream-type system. An *ACE_Message_Blocks* can be passed an allocator through its constructor that it then uses for allocating memory (See chapter on Memory Management for more on Allocators). For example, it may use the *ACE_Cached_Allocation_Strategy*, which pre-allocates memory and then will return pointers in the memory pool instead of actually allocating memory on the heap when it is required. Such functionality is useful when predictable performance is required, as in real-time systems.

The following example will show how to use some of the message queues functionality.

Example 7

```
#include ”ace/Reactor.h”
#include ”ace/Svc_Handler.h”
#include ”ace/Acceptor.h”
#include ”ace/Synch.h”
#include ”ace/SOCK_Acceptor.h”
#include ”ace/Thread.h”
#define NETWORK_SPEED 3
class MyServiceHandler; //forward declaration
typedef ACE_Singleton<ACE_Reactor,ACE_Null_Mutex> Reactor;
typedef ACE_Acceptor<MyServiceHandler,ACE_SOCK_ACCEPTOR> Acceptor;

class MyServiceHandler:
public ACE_Svc_Handler<ACE_SOCK_STREAM,ACE_MT_SYNCH>{
// The message sender and creator threads are handled here.
ACE_thread_t thread_names[2];

public:
int open(void*){
   ACE_DEBUG((LM_DEBUG, ”Acceptor: received new connection \n”));

   //Register with the reactor to remember this handler..
   Reactor::instance()
            ->register_handler(this,ACE_Event_Handler::READ_MASK);
   ACE_DEBUG((LM_DEBUG,”Acceptor: ThreadID:(%t) open\n”));

   //Create two new threads to create and send messages to the
   //remote machine.
   activate(THR_NEW_LWP,
       2, //2 new threads
       0,
       ACE_DEFAULT_THREAD_PRIORITY,
```

```
        -1,
        this,
        0,
        0,
        0,
        thread_names); // identifiers in thread_handles

    //keep the service handler registered with the acceptor.
    return 0;
    }

void send_message(void){
    //Dequeue the message and send it off
    ACE_DEBUG((LM_DEBUG,"(%t)Sending message::>>"));

    //dequeue the message from the message queue
    ACE_Message_Block *mb;
    ACE_ASSERT(this->getq(mb)!=-1);
    int length=mb->length();
    char *data =mb->rd_ptr();


    //Send the data to the remote peer
    ACE_DEBUG((LM_DEBUG,"%s \n",data,length));
    peer().send_n(data,length);

    //Simulate very SLOW network.
    ACE_OS::sleep(NETWORK_SPEED);

    //release the message block
    mb->release();
    } //end send_message

int construct_message(void){
    // A very fast message creation algorithm
    // would lead to the need for queuing messages..
    // here. These messages are created and then sent
    // using the SLOW send_message() routine which is
    // running in a different thread so that the message
    //construction thread isn't blocked.
    ACE_DEBUG((LM_DEBUG,"(%t)Constructing message::>> "));



     // Create a new message to send
    ACE_Message_Block *mb;
    char *data="Hello Connector";
    ACE_NEW_RETURN (mb,ACE_Message_Block (16,//Message 16 bytes long
         ACE_Message_Block::MB_DATA,//Set header to data
         0,//No continuations.
         data//The data we want to send
         ), 0);
    mb->wr_ptr(16); //Set the write pointer.
```

```
   // Enqueue the message into the message queue
   // we COULD have done a timed wait for enqueuing in case
   // someone else holds the lock to the queue so it doesn’t block
   //forever..
   ACE_ASSERT(this->putq(mb)!=-1);
   ACE_DEBUG((LM_DEBUG,”Enqueued msg successfully\n”));
}

int svc(void){
   ACE_DEBUG( (LM_DEBUG,”(%t) Svc thread \n”));

   //call the message creator thread
   if(ACE_Thread::self()== thread_names[0])
      while(1) construct_message(); //create messages forever
   else
      while(1) send_message(); //send messages forever
   return 0; // keep the compiler happy.
}

int handle_input(ACE_HANDLE){
   ACE_DEBUG((LM_DEBUG,”(%t) handle_input ::”));
   char* data= new char[13];

   //Check if peer aborted the connection
   if(peer().recv_n(data,12)==0){
      printf(”Peer probably aborted connection”);
      return -1; //de-register from the Reactor.
      }

   //Show what you got..
   ACE_OS::printf(”<< %s\n”,data);

   //keep yourself registered
   return 0;

   }
};

int main(int argc, char* argv[]){
   ACE_INET_Addr addr(10101);
   ACE_DEBUG((LM_DEBUG,”Thread: (%t) main”));

   //Prepare to accept connections
   Acceptor myacceptor(addr,Reactor::instance());

   // wait for something to happen.
   while(1)
      Reactor::instance()->handle_events();

   return 0;
}
```

This example illustrates the use of the *putq()* and *getq()* methods to enqueue and dequeue message blocks onto the queue. It also illustrates how to create a message block and then how to set its write pointer and read from its read pointer. Note that the actual data inside the message block starts at the read pointer of the message block. The *length()* member function of the message block returns the length of the underlying data stored in the message block and does not include the parts of *ACE_Message_Block* which are used for book keeping purposes. In addition, we also show how to release the message block `(mb)` using the *release()* method.

To read more about how to use message blocks, data blocks or the message queue, please read the sections in this manual on message queues and the ASX framework. Also, see the relevant sections in the reference manual.

HOW THE ACCEPTOR AND CONNECTOR PATTERNS WORK

Both the acceptor and connector factories, i.e. *ACE_Connector* and *ACE_Acceptor*, have a very similar operational structure. Their workings can be roughly divided into three stages.

- Endpoint or connection initialization phase
- Service Initialization phase
- Service Processing phase

Endpoint or connection initialization phase

In the case of the acceptor, the application-level programmer may either call the *open()* method of the factory, *ACE_Acceptor*, or its default constructor (which in fact WILL call the *open()* method) to start to passively listen to connections. When the *open()* method is called on the acceptor factory, it first instantiates the Reactor singleton if it has not already been instantiated. It then proceeds to call the underlying concrete acceptors *open()* method. The concrete acceptor will then go through the necessary initialization it needs to listen for incoming connections. For example, in the case of *ACE_SOCK_Acceptor*, it will open a socket and bind the socket to the port and address on which the user wishes to listen for new connections. After binding the port, it will proceed to issue the listen call. The open method then registers the acceptor factory with the Reactor. Thus when any incoming connection requests are received, the reactor will automatically call back the Acceptor factories *handle_input()* method. Notice that the Acceptor factory itself derives from the *ACE_Event_Handler* hierarchy for this very reason, so that it can respond to *ACCEPT* events and can be called back from the Reactor.

In the case of the connector, the application programmer will call the *connect()* method or the *connect_n()* method on the connector factory to initiate a connection to the peer. Both these methods take, among other options, the remote address to which we wish to connect and whether we want to synchronously or asynchronously complete the connection. We would initiate *NUMBER_CONN* connections either synchronously or asynchronously as:

```
//Synchronous
OurConnector.connect_n(NUMBER_CONN,ArrayofMySvcHandlers,Remote_Addr,0,
                                          ACE_Synch_Options::synch);
//Asynchronous
OurConnector.connect_n(NUMBER_CONN,ArrayofMySvcHandlers,Remote_Addr,0,
                                    ACE_Synch_Options::asynch);
```

If the connect call is issued to be asynchronous, then the *ACE_Connector* will register itself with the reactor awaiting the connection to be established (again *ACE_Connector* also is from the *ACE_Event_Handler* hierarchy). Once the connection is established, the reactor will then automatically call back the connector. In the synchronous case, however, the *connect()* call will block until either the connection is established or a timeout value expires. The time out value can be specified by changing certain *ACE_Synch_Options*. For details please see the reference manual.

Service Initialization Phase for the Acceptor

When an incoming request comes in on the specified address and port, the reactor automatically calls back the *ACE_Acceptor* factory's *handle_input()* method.

This method is a "**Template Method**". A template method is used to define the order of the steps of an algorithm, but allow variation in how certain steps are performed. This variation is achieved by allowing subclasses to define the implementation of these methods. (For more on the Template Method see the reference on Design Patterns).

In this case the Template method defines the algorithm as

- *make_svc_handler()*: Creates the Service Handler.
- *accept_svc_handler()*: Accept the connection into the created Service Handler from the previous step.
- *activate_svc_handler()*: Start the new service handler up.

Each of these methods can be rewritten to provide flexibility in how these operations are actually performed.

Thus the *handle_input()* will first call the *make_svc_handler()* method, which creates the service handler of the appropriate type (the type of the service handler is passed in by the application programmer when the *ACE_Acceptor* template is instantiated, as we saw in the examples above). In the default case, the *make_svc_handler()* method just instantiates the correct service handler. However, the *make_svc_handler()* is a "bridge" method that can be overloaded to provide more complex functionality. (A bridge is a design pattern which decouples the interface of a hierarchy of classes from the implementation hierarchy - read more about this in the reference on "Design Patterns"). For example, the service handle can be created so that it is a process-wide or thread-specific singleton, or it can be dynamically linked in from a library, loaded from disk or even created by doing something as complicated as finding and obtaining the service handler in a database and then bringing it into memory.

After the service handler has been created, the *handle_input()* method proceeds to call *accept_svc_handler()*. This method "accepts" the connection "into" the service handler.

The default case is to call the underlying concrete acceptor's *accept()* method. In the case that *ACE_SOCK_Acceptor* is being used as the concrete acceptor, it proceeds to call the BSD *accept()* routine which establishes the connection ("accepts" the connection). After the connection is established, the handle to the connection is automatically set inside the service handler which was previously created by calling *make_svc_handler()* (accepts "into" the service handler). This method can also be overloaded to provide more complicated functionality. For example, instead of actually creating a new connection, an older connection could be "recycled". This will be discussed in more detail when we show various different accepting and connecting strategies.

Service Initialization Phase for the Connector

The *connect()* method, which is issued by the application, is similar to the *handle_input()* method in the Acceptor factory, i.e. it is a "Template Method".

In this case, the Template method *connect()* defines the following steps, which can be redefined.

- *make_svc_handler()*: Creates the Service Handler.
- *connect_svc_handler()*: Accept the connection into the created Service Handler from the previous step.
- *activate_svc_handler()*: Start the new service handler up.

Each of these methods can be rewritten to provide flexibility in how these operations are actually performed.

Thus after the *connect()* call is issued by the application, the connector factory proceeds to instantiate the correct service handler by calling the *make_svc_handler()* call, exactly as it does in the case of the acceptor. The default behavior is to just instantiate the appropriate class. This can be overloaded exactly in the same manner as was discussed for the Acceptor. The reasons for doing such an overload would probably very similar to the ones mentioned above.

After the service handler has been created, the *connect()* call determines if the connect is to be asynchronous or synchronous. If it is asynchronous, it registers itself with the reactor before continuing on to the next step. It then proceeds to call the *connect_svc_handler()* method. This method, by default, calls the underlying concrete connector's *connect()* method. In the case of *ACE_SOCK_Connector* this would mean issuing the BSD *connect()* call with the correct options for blocking or non-blocking I/O.

If the connection was specified to be synchronous, this call will block until the connection has been established. In this case, when the connection has been established, it will proceed to set the handle in the service handler so that it can communicate with the peer it is now connected to (this is the handle stored in the stream which is obtained by calling the *peer()* method inside the service handler, see examples above). After setting the handle in the service handler, the connector pattern would then continue to the final stage of service processing.

However, if the connection is specified to be asynchronous, the call to *connect_svc_handler()* will return immediately after issuing a non-blocking *connect()* call to the underlying concrete connector. In the case of *ACE_SOCK_Connector*, this would mean a non-blocking BSD *connect()* call. When the connection is in fact established at a later time, the reactor will call back the *ACE_*Connector factory's *handle_output()* method, which would set the new handle in the service handler that was created with the *make_svc_handler()* method. The factory would then continue to the next stage of service processing.

As was in the case of the *accept_svc_handler()*, *connect_svc_handler()* is a "bridge" method that can be overloaded to provide varying functionality.

Service Processing

Once the service handler has been created, the connection has been established, and the handle has been set in the service handler, the *handle_input()* method of *ACE_Acceptor* (or *handle_output()* or *connect_svc_handler()* in the case of *ACE_Connector*) will call the *activate_svc_handler()* method. This method will then proceed to activate the service handler, i.e. to will start it running. The default method is just to call the *open()* method as the entry point into the service handler. As we saw in the examples above, the *open()* method was indeed the first method which was called when the service handler started running. It was here that we called the *activate()* method to create multiple threads of control and also registered the service handler with the reactor so that it was automatically called back when new data arrived on the connection. This method is also a "bridge" method and can be overloaded to provide more complicated functionality. In particular, this overloaded method may provide for a more complicated concurrency strategy, such as running the service handler in a different process.

TUNING THE ACCEPTOR AND CONNECTOR POLICIES

As mentioned above, the acceptor and connector can be easily tuned because of the bridge methods which can be overloaded. The bridge methods allow tuning of:

- **Creation Strategy for the Service Handler:** By overloading the *make_svc_handler()* method in either the acceptor or connector. For example, this could mean re-using an existing service handler or using some complicated method to obtain the service handler, as was discussed above.
- **Connection Strategy:** The connection creation strategy can be changed by overloading the *connect_svc_handler()* or *accept_svc_handler()* methods.
- **Concurrency Strategy for the Service Handler:** The concurrency strategy for the service handler can be changed by overloading the *activate_svc_handler()* method. For example, the service handler can be created in a different process.

As noted above, the tuning is done by overloading bridge methods in the *ACE_Acceptor* or *ACE_Connector* classes. ACE has been designed so that such overloading and tuning can be done very easily.

The ACE_Strategy_Connector and ACE_Strategy_Acceptor classes

To facilitate the tuning of the acceptor and connector patterns along the lines mentioned above, ACE provides two special "tunable" acceptor and connector factories that are very similar to *ACE_Acceptor* and *ACE_Connector*. These are *ACE_Strategy_Acceptor* and *ACE_Strategy_Connector*. These classes make use of the "Strategy" Pattern.

The strategy pattern is used to decouple algorithmic behavior from the interface of a class. The basic idea is to allow the underlying algorithms of a class (call it the Context Class) to vary independently from the clients that use the class. This is done with the help of concrete strategy classes. Concrete strategy classes encapsulate an algorithm or method to perform an operation. These concrete strategy classes are then used by the context class to perform operations (The context class *delegates* the "work" to the concrete strategy class). Since the context class doesn't perform any of the operations directly, it does not have to be modified when functionality is to be changed. The only modification to the context class is that a different concrete strategy class will be used to perform the now changed operation. (To read more about the Strategy Pattern read the appendix on Design Patterns).

In the case of ACE, the *ACE_Strategy_Connector* and the *ACE_Strategy_Acceptor* are Strategy Pattern classes which use several concrete strategy classes to vary the algorithms for creating service handlers, establishing connections and for setting the concurrency method for service handlers. As you may have guessed, the *ACE_Strategy_Connector* and *ACE_Strategy_Acceptor* exploit the tunability provided by the bridge methods which were mentioned above.

Using the Strategy Acceptor and Connector

Several concrete strategy classes are already available in ACE that can be used to "tune" the Strategy Acceptor and Connector. They are passed in as parameters to either Strategy Acceptor or Connector when the class is instantiated. The following table shows some of the classes that can be used to tune the Strategy Acceptor and Connector classes.

To modify the	Concrete Strategy Class	Description
Creation Strategy (overrides make_svc_handler())	`ACE_NOOP_Creation_Strategy`	This concrete strategy does NOT instantiate the service handler and is just a no-op.
	`ACE_Singleton_Strategy`	Ensures that the service handler that is created is a singleton. That is, all connections will effectively use the same service handling routine.
	`ACE_DLL_Strategy`	Creates a service handler by dynamically linking it from a dynamic link library.
Connection Strategy (overrides connect_svc_handler())	`ACE_Cached_Connect_Strategy`	Checks to see if there is already a service handler connected to a particular remote address which isn't being used. If there is such a service handler, then it will re-use that old service handler.

Concurrency Strategy (overrides activate_svc_handler())	`ACE_NOOP_Concurrency_Strategy`	A do-nothing concurrency strategy. Will NOT even call the open() method of the service handler.
	`ACE_Process_Strategy`	Create the service handler in a different process and call its open() hook.
	`ACE_Reactive_Strategy`	First register the service handler with the reactor and then call its open() hook.
	`ACE_Thread_Strategy`	First call the service handlers open method and then call its activate() method so that another thread starts the svc() method of the service handler.

Some examples will help illustrate the use of the strategy acceptor and connector classes.

Example 8

```
#include ”ace/Reactor.h”
#include ”ace/Svc_Handler.h”
#include ”ace/Acceptor.h”
#include ”ace/Synch.h”
#include ”ace/SOCK_Acceptor.h”
#define PORT_NUM 10101
#define DATA_SIZE 12
//forward declaration
class My_Svc_Handler;
//instantiate a strategy acceptor
typedef ACE_Strategy_Acceptor<My_Svc_Handler,ACE_SOCK_ACCEPTOR>
MyAcceptor;
//instantiate a concurrency strategy
typedef ACE_Process_Strategy<My_Svc_Handler> Concurrency_Strategy;

// Define the Service Handler
class My_Svc_Handler:
 public ACE_Svc_Handler <ACE_SOCK_STREAM,ACE_NULL_SYNCH>{
private:
char* data;
public:
My_Svc_Handler(){
data= new char[DATA_SIZE];
}
My_Svc_Handler(ACE_Thread_Manager* tm){
data= new char[DATA_SIZE];
}
int open(void*){
   cout<<”Connection established”<<endl;
   //Register with the reactor
   ACE_Reactor::instance()->register_handler(this,
   ACE_Event_Handler::READ_MASK);
   return 0;
   }

int handle_input(ACE_HANDLE){
   peer().recv_n(data,DATA_SIZE);
   ACE_OS::printf(”<< %s\n”,data);

   // keep yourself registered with the reactor
```

```
    return 0;
}
};

int main(int argc, char* argv[]){
   ACE_INET_Addr addr(PORT_NUM);

   //Concurrency Strategy
   Concurrency_Strategy my_con_strat;

//Instantiate the acceptor
   MyAcceptor acceptor(addr, //address to accept on
      ACE_Reactor::instance(), //the reactor to use
      0, // don't care about creation strategy
      0, // don't care about connection estb. strategy
      &my_con_strat); // use our new process concurrency strategy

while(1) /* Start the reactor's event loop */
   ACE_Reactor::instance()->handle_events();
}
```

This example is based on example 2 above. The only difference is that it uses the *ACE_Strategy_Acceptor* instead of using the *ACE_Acceptor*, and uses the *ACE_Process_Strategy* as the concurrency strategy for the service handler. This concurrency strategy ensures that the service handler is instantiated in a separate process once the connection has been established. If the load on a certain service is going to be extremely high, it may be a good idea to use the *ACE_Process_Strategy*. In most cases, however, using the *ACE_Process_Strategy* would be too expensive, and *ACE_Thread_Strategy* would probably be the better concurrency strategy to use.

Using the ACE_Cached_Connect_Strategy for Connection caching

In many applications, clients connect and then reconnect to the same server several times, each time establishing the connection, performing some work and then tearing down the connection (such as is done in Web clients). Needless to say, this is very inefficient and expensive as connection establishment and teardown are expensive operations. A better strategy in such a case would be for the connector to "remember" old connections and not tear them down untill it is sufficiently sure that the client will not try to re-establish a connection again. The *ACE_Cached_Connect_Strategy* provides just such a caching strategy. This strategy object is used by the *ACE_Strategy_Connector* to provide for cache-based connection establishment. If a connection already exists, the *ACE_Strategy_Connector* will reuse it instead of creating a new connection.

When the client tries to re-establish a connection to a server that it had previously formed a connection with, the *ACE_Cached_Connect_Strategy* ensures that the old connection and the old service handler are reused instead of creating a new connection and new service handler. Thus, in truth, the *ACE_Cached_Connect_Strategy* not only manages the connection establishment strategy, it also manages the service handler creation strategy. Since in this case, the user does NOT want to create new service handlers, we pass the

ACE_Strategy_Connector an *ACE_Null_Creation_Strategy*. If the connection has never been established before, then *ACE_Cached_Connect_Strategy* will automatically instantiate the correct service handler that was passed to it when this template class is instantiated using an internal creation strategy. This strategy can be set to any strategy the user wishes to use. Besides this, the *ACE_Cached_Connect_Strategy* itself can be passed the creation, concurrency and recycling strategies it uses in its constructor. The following example illustrates these ideas.

Example 9

```
#include "ace/Reactor.h"
#include "ace/Svc_Handler.h"
#include "ace/Connector.h"
#include "ace/Synch.h"
#include "ace/SOCK_Connector.h"
#include "ace/INET_Addr.h"

#define PORT_NUM 10101
#define DATA_SIZE 16

//forward declaration
class My_Svc_Handler;
//Function prototype
static void make_connections(void *arg);

// Template specializations for the hashing function for the
// hash_map which is used by the cache. The cache is used internally by
the
// Cached Connection Strategy . Here we use ACE_Hash_Addr
// as our external identifier. This utility class has already
// overloaded the == operator and the hash() method. (The
// hashing function). The hash() method delegates the work to
// hash_i() and we use the IP address and port to get a
// a unique integer hash value.
size_t
ACE_Hash_Addr<ACE_INET_Addr>::hash_i (const ACE_INET_Addr &addr) const
{
  return addr.get_ip_address () + addr.get_port_number ();
}

//instantiate a strategy acceptor
typedef ACE_Strategy_Connector<My_Svc_Handler,ACE_SOCK_CONNECTOR>
STRATEGY_CONNECTOR;

//Instantiate the Creation Strategy
typedef ACE_NOOP_Creation_Strategy<My_Svc_Handler>
        NULL_CREATION_STRATEGY;
//Instantiate the Concurrency Strategy
typedef ACE_NOOP_Concurrency_Strategy<My_Svc_Handler>
        NULL_CONCURRENCY_STRATEGY;
//Instantiate the Connection Strategy
typedef ACE_Cached_Connect_Strategy<My_Svc_Handler,
                                    ACE_SOCK_CONNECTOR,
                                    ACE_SYNCH_RW_MUTEX>
```

```
        CACHED_CONNECT_STRATEGY;

class My_Svc_Handler:
public ACE_Svc_Handler <ACE_SOCK_STREAM,ACE_MT_SYNCH>{
private:
char* data;

public:
My_Svc_Handler(){
data= new char[DATA_SIZE];
   }
My_Svc_Handler(ACE_Thread_Manager* tm){
data= new char[DATA_SIZE];
   }
//Called before the service handler is recycled..
int
recycle (void *a=0){
   ACE_DEBUG ((LM_DEBUG,
            "(%P|%t) recycling Svc_Handler %d with handle %d\n",
              this, this->peer ().get_handle ()));
   return 0;
   }

int open(void*){
   ACE_DEBUG((LM_DEBUG,"(%t)Connection established \n"));




   //Register the service handler with the reactor
   ACE_Reactor::instance()
        ->register_handler(this,ACE_Event_Handler::READ_MASK);
   activate(THR_NEW_LWP|THR_DETACHED);
   return 0;
   }

int handle_input(ACE_HANDLE){
   ACE_DEBUG((LM_DEBUG,"Got input in thread: (%t) \n"));
   peer().recv_n(data,DATA_SIZE);
   ACE_DEBUG((LM_DEBUG,"<< %s\n",data));

   //keep yourself registered with the reactor
   return 0;
   }

int svc(void){
   //send a few messages and then mark connection as idle so that it can
   // be recycled later.
   ACE_DEBUG((LM_DEBUG,"Started the service routine \n"));

   for(int i=0;i<3;i++){
      ACE_DEBUG((LM_DEBUG,"(%t)>>Hello World\n"));
      ACE_OS::fflush(stdout);
      peer().send_n("Hello World",sizeof("Hello World"));
```

```
      }

   //Mark the service handler as being idle now and let the
   //other threads reuse this connection
   this->idle(1);

   //Wait for the thread to die
   this->thr_mgr()->wait();
   return 0;
   }
};
ACE_INET_Addr *addr;

int main(int argc, char* argv[]){
   addr= new ACE_INET_Addr(PORT_NUM,argv[1]);
   //Creation Strategy
   NULL_CREATION_STRATEGY creation_strategy;

   //Concurrency Strategy
   NULL_CONCURRENCY_STRATEGY concurrency_strategy;

   //Connection Strategy
   CACHED_CONNECT_STRATEGY caching_connect_strategy;


   //instantiate the connector
   STRATEGY_CONNECTOR connector(
        ACE_Reactor::instance(), //the reactor to use
        &creation_strategy,
        &caching_connect_strategy,
        &concurrency_strategy);
   //Use the thread manager to  spawn a single thread
   //to connect multiple times passing it the address
   //of the strategy connector
   if(ACE_Thread_Manager::instance()->spawn(
            (ACE_THR_FUNC) make_connections,
            (void *) &connector,
            THR_NEW_LWP) == -1)
   ACE_ERROR ((LM_ERROR, "(%P|%t) %p\n%a", "client thread spawn
failed"));

while(1) /* Start the reactor's event loop */
   ACE_Reactor::instance()->handle_events();
}

//Connection establishment function, tries to establish connections
//to the same server again and re-uses the connections from the cache
void make_connections(void *arg){
   ACE_DEBUG((LM_DEBUG,"(%t)Prepared to connect \n"));
   STRATEGY_CONNECTOR *connector= (STRATEGY_CONNECTOR*) arg;
   for (int i = 0; i < 10; i++){
      My_Svc_Handler *svc_handler = 0;
```

```
   // Perform a blocking connect to the server using the Strategy
   // Connector with a connection caching strategy.  Since we are
   // connecting to the same <server_addr> these calls will return the
   // same dynamically allocated <Svc_Handler> for each <connect> call.
   if (connector->connect (svc_handler, *addr) == -1){
      ACE_ERROR ((LM_ERROR, "(%P|%t) %p\n", "connection failed\n"));
      return;
      }

   // Rest for a few seconds so that the connection has been freed up
   ACE_OS::sleep (5);
   }
}
```

In the above example, the Cached Connection Strategy is used to cache connections. To use this strategy, a little extra effort is required to define the *hash()* method on the hash map manager that is used internally by *ACE_Cached_Connect_Strategy*. The *hash()* method is the hashing function, which is used to hash into the cache map of service handlers and connections that is used internally by the *ACE_Cached_Connect_Strategy*. It simply uses the sum of the IP address and port number as the hashing function, which is probably not a very good hash function.

The example is also a little more complicated then the ones that have been shown so far and thus warrants a little extra discussion.

We use a no-op concurrency and creation strategy with the *ACE_Strategy_Acceptor*. Using a no-op creation strategy IS necessary, as was explained above, if this is not set to a *ACE_NOOP_Creation_Strategy*, the *ACE_Cached_Connection_Strategy* will cause an assertion failure. When using the *ACE_Cached_Connect_Strategy*, however, any concurrency strategy can be used with the strategy acceptor. As was mentioned above, the underlying creation strategy used by the *ACE_Cached_Connect_Strategy* can be set by the user. The recycling strategy can also be set. This is done when instantiating the caching_connect_strategy by passing its constructor the strategy objects for the required creation and recycling strategies. Here, we have not done so, and are using both the default creation and recycling strategy.

After the connector has been set up appropriately, we use the *Thread_Manager* to spawn a new thread with the `make_connections()` method as its entry point. This method uses our new strategy connector to connect to a remote site. After the connection is established, this thread goes to sleep for five seconds and then tries to re-create the same connection using our cached connector. This thread should then, in its next attempt, find the connection in the connectors cache and reuse it.

Our service handler (`My_Svc_Handler`) is called back by the reactor, as usual, once the connection has been established. The *open()* method of `My_Svc_Handler` then makes itself into an active object by calling its *activate()* method. The *svc()* method then proceeds to send three messages to the remote host and then marks the connection idle by calling the *idle()* method of the service handler. Note the call to the thread manager, asking it to wait `(this->thr_mgr-wait())` for all threads in the thread manager to terminate. If you do not ask the thread manager to wait for the other threads, then the semantics have been set up in ACE such that once the thread in an *ACE_Task* (or in this case the *ACE_Svc_Handler* which is a type of *ACE_Task*) is terminated, the *ACE_Task* object (or in this case the *ACE_My_Svc_Handler*) will automatically be deleted. If this happens, then, when the *Cache_Connect_Strategy* goes looking for previously cached connections, it will NOT find `My_Svc_Handler` as we expect it too, as, of course, this has been deleted.

The *recycle()* method in *ACE_Svc_Handler* has also been overloaded in `My_Svc_Handler`. This method is automatically called back when an old connection is found by the *ACE_Cache_Connect_Strategy*, so that the service handler may do recycle specific operations in this method. In our case, we just print out the address of the `this` pointer of the handler which was found in the cache. When the program is run, we notice that the address of the handle being used after each connection is established is the same, indicating that the caching is working correctly.

Using Simple Event Handlers with the Acceptor and Connector patterns

At times, using the heavy weight *ACE_Svc_Handler* as the handler for acceptors and connectors may be unwarranted and cause code bloat. In these cases, the user may use the lighter *ACE_Event_Handler* method as the class which is called back by the reactor once the connection has been established. To do so, one needs to overload the *get_handle()* method and also include a concrete underlying stream which will be used by the event handler. An example should help illustrate these changes. Here we have also written a new *peer()* method which returns a reference to the underlying stream, as it did in the *ACE_Svc_Handler* class.

Example 10

```
#include ”ace/Reactor.h”
#include ”ace/Svc_Handler.h”
#include ”ace/Acceptor.h”
#include ”ace/Synch.h”
#include ”ace/SOCK_Acceptor.h”
#define PORT_NUM 10101
#define DATA_SIZE 12

//forward declaration
class My_Event_Handler;

//Create the Acceptor class
typedef ACE_Acceptor<My_Event_Handler,ACE_SOCK_ACCEPTOR>
MyAcceptor;

//Create an event handler similar to as seen in example 2. We have to
//overload the get_handle() method and write the peer() method. We also
//provide the data member peer_ as the underlying stream which is
//used.
class My_Event_Handler:
 public ACE_Event_Handler{
```

```
private:
char* data;
//Add a new attribute for the underlying stream which will be used by
//the Event Handler
ACE_SOCK_Stream peer_;
public:
My_Event_Handler(){
   data= new char[DATA_SIZE];
      }

int
open(void*){
   cout<<"Connection established"<<endl;
   //Register the event handler with the reactor
   ACE_Reactor::instance()->register_handler(this,
   ACE_Event_Handler::READ_MASK);
   return 0;
   }

int
handle_input(ACE_HANDLE){
   // After using the peer() method of our ACE_Event_Handler to obtain a
   //reference to the underlying stream of the service handler class we
   //call recv_n() on it to read the data which has been received. This
   //data is stored in the data array and then printed out
   peer().recv_n(data,DATA_SIZE);
   ACE_OS::printf("<< %s\n",data);

   // keep yourself registered with the reactor
   return 0;
   }

// new method which returns the handle to the reactor when it
//asks for it.
ACE_HANDLE
get_handle(void) const{
   return this->peer_.get_handle();
   }

//new method which returns a reference to the peer stream
ACE_SOCK_Stream &
peer(void) const{
   return (ACE_SOCK_Stream &) this->peer_;
   }
};

int main(int argc, char* argv[]){
   ACE_INET_Addr addr(PORT_NUM);
   //create the acceptor
   MyAcceptor acceptor(addr, //address to accept on
      ACE_Reactor::instance()); //the reactor to use
while(1) /* Start the reactor's event loop */
   ACE_Reactor::instance()->handle_events();
```

Chapter

8

The Service Configurator

A pattern for Dynamic Configuration of Services

Many distributed systems contain a collection of *global* services. These services can be called upon by an application developer to assist in him in his distributed development needs. Global services, such as name services, remote terminal access services, logging and time services are needed when constructing distrbuted applications. One way to construct these services is to write each service as a separate program. These service programs are then executed and run in their own private processes. However, this leads to configuration nightmares. An administrator would have to go to each node and execute the service programs in accordance with current user needs and policies. If new services have to be added or older ones removed, then the administrator has to spend the time to go to each machine again and re-configure. Further, such configuration is *static*. To re-configure, the adminstrator manually stops a service (by *kill()*ing the service process) and then restarts a replacement service. Also, services may be running on machines where they are never used by *any* application. Obviously, such an approach is inefficient and undesirable.

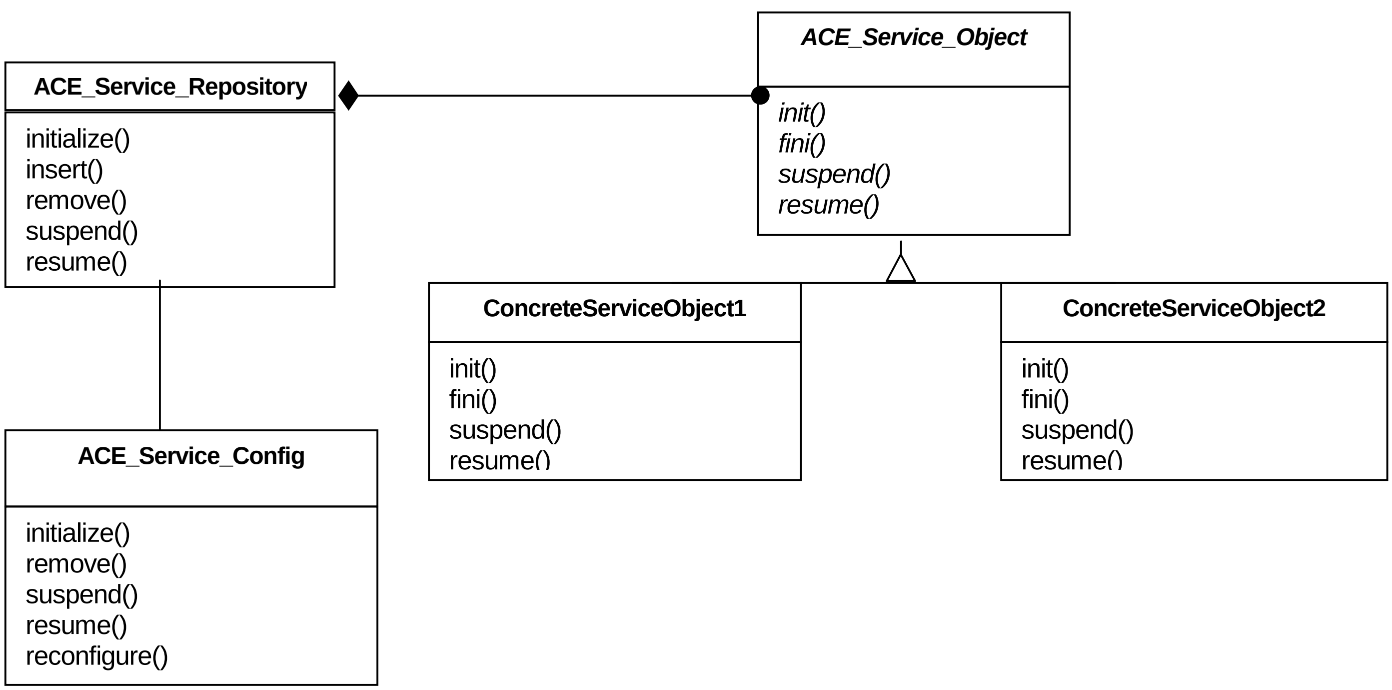
It would be much more convenient if the services could be *dynamically* started, removed, suspended and resumed. Thus, the service developer doesn't have to worry about how the service will be configured. All he cares about is how the service gets the job done. The adminstrator should be able to add and replace new services within the application *without* recompling or shutting down the services server process.

The Service Configurator pattern is a pattern which allows all this. It decouples how a service is *implented* from how it is *configured*. New services can be added in an application and old services can be removed without shutting down the server. Most of the time the server which provides services is implemented as a daemon process.

Framework Components

The Service Configurator in ACE consists of the following components:

- An abstract class named `ACE_Service_Object` from which the application developer must subclass to create his own application-specific concrete Service Object.
- The application-specific concrete service objects.
- A Service Repository, `ACE_Service_Repository,` which records the services that the server has running in it or knows about.
- `ACE_Service_Config` which serves as an application developers interface to the entire service configuration framework.
- A service configuration file. This file contains configuration information for all of the service objects. By default it is named `svc.conf`. When your application issues an open() call on the ACE_Service_Config, the service configurator framework will read and process all configuration information that you write within the file. The application will be configured accordingly.



The `ACE_Service_Object` includes methods which are called by the framework when the service is to start (`init()`), stop (`fini()`), suspend (`suspend()`) or is to be resumed (`resume()`). The `ACE_Service_Object` derieves from `ACE_Shared_Object` and `ACE_Event_Handler`. The `ACE_Shared_Object` serves as the abstract base class when an application wishes it to be loaded using the dynamic linking facilities of the operating system.

`ACE_Event_Handler` was introduced in the discussion of the Reactor. A developer subclasses his class from here when he wants it to respond to events from the Reactor.

Why does the Service Object inherit from `ACE_Event_Handler`? One way to initiate a reconfiguration is for the user to generate a signal. When such a signal event occurs, the reactor is used to handle the signal and issue a reconfiguration request to the `ACE_Service_Config`. Besides this, reconfiguration of the software will probably happen after an event has occurred. Thus all Service Objects are built so that they can handle events.

The service configuration file has its own simple script for describing how you want a service to be started and then run. You can define whether you want to add a new service or to suspend, resume or remove an existing service in the application. Parameters can also be sent to these services. The Service Configurator also allows the reconfiguration of ACE based *streams.* We will talk about this more when we have discussed the ACE streams framework.

Specifying the configuration file

The service configuration file specifies which services to load and start in an application. In addition, you can specify which services are to be stopped, suspended or resumed. Parameters can also be sent to your service objects *init()* method.

Starting a service

Services can be started up statically or dynamically. If the service is to be started up dynamically, the service configurator will actually load up the service object from a shared library object (i.e. a dynamic link library). In order to do this, the service configurator needs to know which library contains the object and also it needs to know the name of the object in that library. Thus, in your code file you must also instantiate the service object with a name that you will remember. Thus a dynamic service is configured as:

*dynamic service_name type_of_service * location_of_shared_lib:name_of_object "parameters"*

A static service is initialized as:

static service_name "parameters_sent_to_service_object"

Suspending or resuming a service

When you start a service you assign it a name, as I just mentioned. This name is then used to suspend or resume that service. So all you have to do to suspend a service is to specify:

suspend service_name

in the *svc.conf* file. This causes the *suspend()* method in the service object to be called. Your service object should then suspend itself (based on whatever "suspend" means for that particular service).

If you want to then resume this service, all you have to do is specify:

resume service_name

in the *svc.conf* file. This causes the *resume()* method in the service object to be called. Your service object should then suspend itself (based on whatever "resume" means for that particular service).

Stopping a service

Stopping and then removing a service (if it had been dynamically loaded) is also a simple operation that can be achieved by specifying the following in your configuration file:

remove service_name

This causes the service configurator to call the *fini()*method of your application. This method should stop the service. The service configurator itself will take care of unlinking a dynamic object from the servers address space.

Writing Services

Writing your own service for the Service Configurator is relatively simple. You are free to have this service do whatever you want. The only constraint is that it should be subclassed from *ACE_Service_Object*. It must therefore implement the *init()* and *fini()* methods. When *ACE_Service_Config* is *opened*, it reads the configuration file (i.e. *svc.conf*) and then initializes the services according to the file. Once it loads up a service (by dynamically linking it in if specified), it will call that Service Objects *init()* method. Similarily, if the configuration asks for a service to be removed, the *fini()* method will be called. These methods are responsible for starting and destroying any resources that the service may need, such as memory, connections, threads, etc. The parameters that are specified (field that is set) in the svc.conf file are passed in through the *init()* method of the service object.

The following example illustrates a service which derieves from *ACE_Task_Base*. The *ACE_Task_Base* class contains the *activate()* method that is used to create threads within an object. (*ACE_Task*, which was discussed in the chapter on tasks and active objects, derieves from *ACE_Task_Base* and also includes a message queue for communication purposes. Since we don’t need our service to communicate with another task we just use *ACE_Task_Base* to help us get the job done.) For more on this, read the chapter on Tasks and Active Objects. The service is a “do-nothing” service which periodically broadcasts the time of the day once it is started up.

Example 1a

```
//The Services Header File.
#if !defined(MY_SERVICE_H)
#define MY_SERVICE_H
```

```
#include "ace/OS.h"
#include "ace/Task.h"
#include "ace/Synch_T.h"

// A Time Service class. ACE_Task_Base already derives from
//ACE_Service_Object and thus we don't have to subclass from
//ACE_Service_Object in this case.
class TimeService: public  ACE_Task_Base{
public:

   virtual int init(int argc, ASYS_TCHAR *argv[]);
   virtual int fini(void);
   virtual int suspend(void);
   virtual int resume(void);
   virtual int svc(void);
private:
   int canceled_;
   ACE_Condition<ACE_Thread_Mutex> *cancel_cond_;
   ACE_Thread_Mutex *mutex_;
   };

#endif
```

The corresponding implementation is as follows. When the time service recieves the *init()* call it *activates* a single thread in the task. This will cause a new thread to be created which treats the *svc()* method as its entry point. In the *svc()* method, this thread will loop until it finds that the `canceled_` flag has been set. This flag is set when *fini()* is called by the service configuration framework. The *fini()* method, however, has to be **sure** that the underlying thread is dead BEFORE it returns to the underlying service configuration framework. Why? Because the service configuration will actually unload the shared library which contains the `TimeService`. This will effectively delete the TimeService object from the application process. If the thread isn't dead **before** that happens, it will be issuing a call on code that has been vaporized by the service configurator! Not what we want. To ensure that the thread dies **before** the service configurator vaporizes the `TimeService` object, a condition variable is used. (For more on how condition variables are used, please read the chapter on threads).

Example 1b
```
#include "Services.h"

int TimeService::init(int argc, char *argv[]){
   ACE_DEBUG((LM_DEBUG,"(%t)Starting up the time Service\n"));
   mutex_ = new ACE_Thread_Mutex;
   cancel_cond_ = new ACE_Condition<ACE_Thread_Mutex>(*mutex_);
   activate(THR_NEW_LWP|THR_DETACHED);
   return 0;
   }
```

```
int TimeService::fini(void){
      ACE_DEBUG((LM_DEBUG,
            "(%t)FINISH!Closing down the Time Service\n"));
      //All of the following code is here to make sure that the
      //thread in the task is destroyed before the service configurator
      //deletes this object.
      canceled_=1;
      mutex_->acquire();
      while(canceled_)
        cancel_cond_->wait();
      mutex_->release();
      ACE_DEBUG((LM_DEBUG,"(%t)Time Service is exiting \n"));

      return 0;
      }



//Suspend the Time Service.
int TimeService::suspend(void){
   ACE_DEBUG((LM_DEBUG,"(%t)Time Service has been suspended\n"));
   int result=ACE_Task_Base::suspend();
   return result;
   }

//Resume the Time Service.
int TimeService::resume(void){
   ACE_DEBUG((LM_DEBUG,"(%t)Resuming Time Service\n"));
   int result=ACE_Task_Base::resume();
   return result;
   }

//The entry function for the thread. The tasks underlying thread
//starts here and keeps sending out messages. It stops when:
// a) it is suspeneded
// b) it is removed by fini(). This happens when the fini() method
//    sets the cancelled_ flag to true. Thus causes the TimeService
//    thread to fall through the while loop and die. Before dying it
//    informs the main thread of its imminent death. The main task
//    that was previously blocked in fini() can then continue into the
//    framework and destroy the TimeService object.
int TimeService::svc(void){
   char *time = new char[36];
   while(!canceled_){
      ACE::timestamp(time,36);
      ACE_DEBUG((LM_DEBUG,"(%t)Current time is %s\n",time));
      ACE_OS::fflush(stdout);
      ACE_OS::sleep(1);
      }
   //Signal the Service Configurator informing it that the task is now
   //exiting so it can delete it.
   canceled_=0;
   cancel_cond_->signal();
   ACE_DEBUG((LM_DEBUG,
```

```
            "Signalled main task that Time Service is exiting \n"));
   return 0;
   }

//Define the object here
TimeService time_service;
```

And here is a simple configuration file that is currently set just to activate the time service. The comment # marks can be removed to suspend, resume or remove the service.

```
Example 1c
# To configure different services, simply uncomment the appropriate
#lines in this file!
#resume TimeService
#suspend TimeService
#remove TimeService
#set to dynamically configure the TimeService object and do so without
#sending any parameters to its init method
dynamic TimeService Service_Object * ./Server:time_service  ""
```

And, last but not least, here is the piece of code that starts the service configurator. This code also sets up a signal handler object that is used to initiate the re-configuration. The signal handler has been set up so that it responds to a `SIGWINCH` (signal that is generated when a window is changed). After starting the service configurator, the application enters into a reactive loop waiting for a `SIGWINCH` signal event to occur. This would then call back the signal handler which would call *reconfigure()* on *ACE_Service_Config*. As explained earlier, when this happens, the service configurator re-reads the file and processes whatever new directives the user has put in there. For example, after issuing the dynamic start for the TimeService, in this example you could change the *svc.conf* file so that it has the single suspend statement in it. When the configurator reads this, it will call suspend on the TimeService which will cause it to suspend its underlying thread. Similarily, if later you change *svc.conf* again so that it asks for the service to be resumed then this will call the `TimeService::resume()` method. This in turn resumes the thread that had been suspended earlier.

```
Example 1d
#include "ace/OS.h"
#include "ace/Service_Config.h"
#include "ace/Event_Handler.h"
#include <signal.h>

//The Signal Handler which is used to issue the reconfigure()
//call on the service configurator.

class Signal_Handler: public ACE_Event_Handler{
public:
int open(){
   //register the Signal Handler with the Reactor to handle
   //re-configuration signals
   ACE_Reactor::instance()->register_handler(SIGWINCH,this);
   return 0;
```

```
   }
int handle_signal(int signum, siginfo*,ucontext_t *){
   if(signum==SIGWINCH)
      ACE_Service_Config::reconfigure();
   return 0;
   }
};

int main(int argc, char *argv[]){
   //Instantiate and start up the Signal Handler. This is uses to
   //handle re-configuration events.
   Signal_Handler sh;
   sh.open();
   if (ACE_Service_Config::open (argc, argv) == -1)
       ACE_ERROR_RETURN ((LM_ERROR,
                  "%p\n","ACE_Service_Config::open"),-1);
   while(1)
      ACE_Reactor::instance()->handle_events();
   }
```

Using the Service Manager

The *ACE_Service_Manager* is a service that is can be used to remotely manage the service configurator. It can currently receive two types of requests. First, you can send it a "help" message which will list all the services that are currently loaded into the application. Second, you can send the service manager a "reconfigure" message. This causes the service configurator to reconfigure itself.

Following is an example which illustrates a client that sends these two types of commands to the service manager.

Example 2

```
#include "ace/OS.h"
#include "ace/SOCK_Stream.h"
#include "ace/SOCK_Connector.h"
#include "ace/Event_Handler.h"
#include "ace/Get_Opt.h"
#include "ace/Reactor.h"
#include "ace/Thread_Manager.h"

#define BUFSIZE 128

class Client: public ACE_Event_Handler{
public:
~Client(){
   ACE_DEBUG((LM_DEBUG,"Destructor \n"));
   }


//Constructor
Client(int argc, char *argv[]): connector_(), stream_(){
```

```
   //The user must specify address and port number
   ACE_Get_Opt get_opt(argc,argv,"a:p:");
   for(int c;(c=get_opt())!=-1;){
      switch(c){
        case 'a':
             addr_=get_opt.optarg;
             break;
        case 'p':
             port_= ((u_short)ACE_OS::atoi(get_opt.optarg));
             break;
        default:
             break;
        }
      }
   address_.set(port_,addr_);
   }


//Connect to the remote machine
int connect(){
   connector_.connect(stream_,address_);
   ACE_Reactor::instance()->
      register_handler(this,ACE_Event_Handler::READ_MASK);
   return 0;
   }

//Send a list_services command
int list_services(){
   stream_.send_n("help",5);
   return 0;
   }

//Send the reconfiguration command
int reconfigure(){
   stream_.send_n("reconfigure",12);
   return 0;
   }


//Handle both standard input and remote data from the
//ACE_Service_Manager
int handle_input(ACE_HANDLE h){
   char buf[BUFSIZE];

   //Got command from the user
   if(h== ACE_STDIN){
      int result = ACE_OS::read (h, buf, BUFSIZ);
            if (result == -1)
              ACE_ERROR((LM_ERROR,"can't read from STDIN"));
      else if (result > 0){
        //Connect to the Service Manager
        this->connect();

        if(ACE_OS::strncmp(buf,"list",4)==0)
```

```
            this->list_services();
        else if(ACE_OS::strncmp(buf,"reconfigure",11)==0)
            this->reconfigure();
            }
        return 0;
        }

    //We got input from remote
    else{
            switch(stream_.recv(buf,BUFSIZE)){
            case -1:
                  //ACE_ERROR((LM_ERROR,
                        "Error in recieving from remote\n"));
                  ACE_Reactor::instance()->remove_handler(this,
                                    ACE_Event_Handler::READ_MASK);
                  return 0;
            case 0:
                  return 0;
            default:
                   ACE_OS::printf("%s",buf);
                   return 0;
            }
       }
    }


//Used by the Reactor Framework
ACE_HANDLE get_handle() const{
   return stream_.get_handle();
   }

//Close down the underlying stream
int handle_close(ACE_HANDLE,ACE_Reactor_Mask){
   return stream_.close();
   }

private:
   ACE_SOCK_Connector connector_;
   ACE_SOCK_Stream stream_;
   ACE_INET_Addr address_;
   char *addr_;
   u_short port_;
};







int main(int argc, char *argv[]){
   Client client(argc,argv);
```

```
    //Register the the client event handler as the standard
    //input handler
    ACE::register_stdin_handler(&client,
             ACE_Reactor::instance(),
             ACE_Thread_Manager::instance());
    ACE_Reactor::run_event_loop();
}
```

In this example, the Client class is an event handler which handles two types of events. Standard input events from the user and replies from the *ACE_Service_Manager*. If the user types in a "list" or "reconfigure" command, then the corresponding messages are sent to the remote *ACE_Service_Manager*. The Service Manager in turn will reply with a list of the currently configured services or with "done" (indicating that the service re-configuration is done). Since the *ACE_Service_Manager* is a service, it is added into an application using the service configurator framework, i.e. you specify whether you wish it to be loaded statically or dynamically in the *svc.conf* file.

For example, this will statically start up the service manager at port 9876.

static ACE_Service_Manager "-p 9876"

Chapter

9

Message Queues

The use of Message Queues in ACE

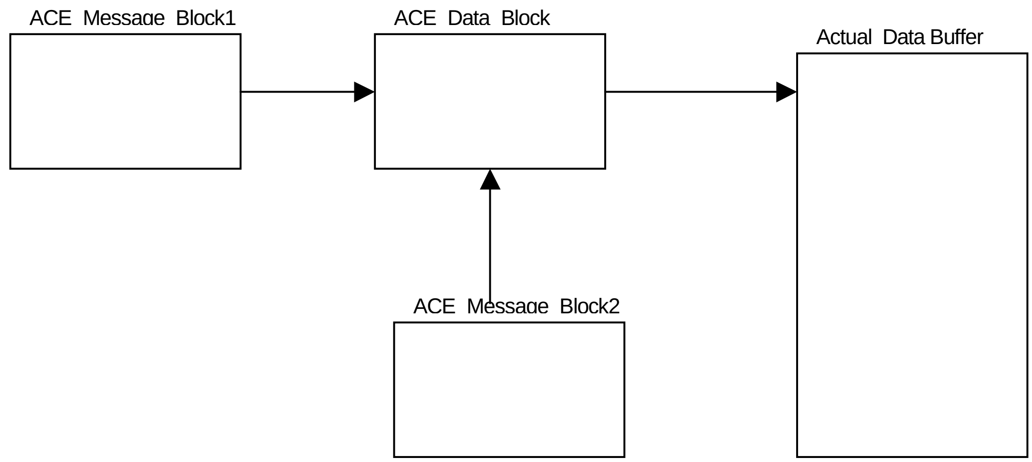
Modern real-time applications are usually constructed as a set of communicating but independent tasks. These tasks can communicate with each other through several mechanisms, one which is commonly used is a message queue. The basic mode of communication in this case is for a sender (or producer) task to enqueue a message onto a message queue and the receiver (or consumer) task to dequeue the message from that queue. This of course is just one of the ways message queues can be used. We will see several different examples of message queue usage in the ensuing discussion.

The message queue in ACE has been modeled after the UNIX System V message queue mechanisms, and thus for those familiar with System V it should be easy to understand their usage and structure. ACE has provided for several different types of message queues. Each of these provide different features and performance characteristics, many of which are specifically required in real-time systems.

Message Blocks

Messages that are enqueued onto message queues are termed message blocks in ACE. Each message block "contains" a header and a data block. Note that the "contains" does NOT mean that the data block is inside the message block but rather that the message block holds a pointer to a data block, which it only logically contains. The data block in turn holds a pointer to an actual data buffer. This allows flexible sharing of data between multiple message blocks. This is illustrated in the figure below. Note that now the two message blocks may be enqueued onto different queues but both use the same data block and thus data. No overhead due to data copying will thus be incurred.

In ACE the message block is implemented as *ACE_Message_Block* and the data block as *ACE_Data_Block. ACE_Message_Block* contains several different constructors which furnish different means to create message blocks efficiently.



Constructing Message Blocks

The easiest way to create a message block is to pass it the data buffer which it is to store (remember that the *ACE_Message_Block* does this indirectly through an *ACE_Data_Block*) and let the constructor of the message block be responsible for creating the underlying data block and setting up the corresponding pointers. This can be done as:

```
char *data[size]=”This is my data”;
ACE_Message_Block *mb = new ACE_Message_Block(data,size);
```

Note that when the message block `mb` is destroyed, the associated data buffer `data` will NOT be destroyed (i.e. this memory will not be deallocated). This only makes sense since the memory was not allocated by the message block in the first place, so it should not be responsible for its deallocation.

A more powerful and thus useful constructor is also available. Among other things, this method allows to use any of the ACE memory allocators to efficiently manage the memory associated with the data buffer. The constructor is:

```
ACE_Message_Block (size_t size,
     ACE_Message_Type type = MB_DATA,
     ACE_Message_Block *cont = 0,
     const char *data = 0,
     ACE_Allocator *allocator_strategy = 0,
     ACE_Lock *locking_strategy = 0,
     u_long priority = 0,
     const ACE_Time_Value & execution_time = ACE_Time_Value::zero,
     const ACE_Time_Value & deadline_time = ACE_Time_Value::max_time);
```

The above constructor is called with the parameters:

1. The size of the data buffer that is to be associated with the message block. Note that the *size* of the message block will be `size`, but the *length* will be 0 until the *wr_ptr* is set. This will be explained further later.
2. The type of the message. (There are several types available in the *ACE_Message_Type* enumeration including data messages, which is the default).

3. A pointer to the next message block in the “fragment chain”. Message blocks can actually be linked together to form chains. Each of these chains can then be enqueued onto a message queue as if it were only a single message block. This defaults to 0, meaning that chaining is not used for this block.
4. A pointer to the data buffer which is to be stored in this message block. If the value of this parameter is zero, then a buffer of the size specified in the first parameter is created and managed by the message block. When the message block is deleted, the corresponding data buffer is also deleted. However, if the data buffer is specified in this parameter, i.e. the argument is not null, then the message block will NOT delete the data buffer once it is destroyed. This is an important distinction and should be noted carefully.
5. The `allocator_strategy` to be used to allocate the data buffer (if needed), i.e. if the 4th parameter was null (as explained above). Any of the *ACE_Allocator* sub-classes can be used as this argument. (See the chapter on Memory Management for more on *ACE_Allocators).*
6. If `locking_strategy` is non-zero, then this is used to protect regions of code that access shared state (e.g. reference counting) from race conditions.
7. This and the next two parameters are used for scheduling for the real-time message queues which are available with ACE, and should be left at their default for now.

Inserting and manipulating data in a message block

In addition to the constructors, *ACE_Message_Block* offers several methods to insert data into a message block directly. Furthermore, methods are also available to manipulate the data present in a message block.

Each *ACE_Message_Block* has two underlying pointers which are used to manipulate the reading and writing of data to and from a message block, aptly named the *rd_ptr* and *wr_ptr*. They are accessible directly by calling the *rd_ptr()* and *wr_ptr()* methods. The *rd_ptr* points to the location where data is to be read from next in the data block, and the *wr_ptr* points to the location where data is to be written to next. Careful management by the programmer is required to insure that both of these pointers always point to the correct location, i.e. when data is read or written using them that they are advanced to the right positions again and not left as is. Most of the internal methods also make use of these two underlying pointers and thus can change their state. Furthermoe, if they are in an incorrect state before these methods are called, then the methods will produce incorrect behavior. It is the programmer's responsibility to make sure that he/she knows what is happening in regard to these pointers.

For inserting new data into an existing message block, a simple *copy()* method is available.

```
int copy(const char *buf, size_t n);
```

The copy method takes a pointer to the buffer that is to be copied into the message block and the size of the data to be copied as arguments. This method uses the *wr_ptr* and begins writing from this point onwards till it reaches the end of the data buffer as specified by the size argument. *copy()* will also ensure that the *wr_ptr* is updated so that is points to the new end of the buffer. Note that this method will actually perform a physical copy, and thus should be used with caution.

The *base()* and *length()* methods can be used in conjunction to obtain the data buffer from a message block. *base()* returns a pointer to the beginning of the data block and *length()* returns its size.

The *duplicate()* and *clone()* methods are used to make a "copy" of a message block. The *clone()* method as the name suggests actually creates a fresh copy of the entire message block including its data blocks and continuations, i.e. a deep copy. The *duplicate()* method, on the other hand, uses the *ACE_Message_Block*'s reference counting mechanism. It returns a pointer to the message block which is to be duplicated and internally increments an internal reference count. Once done with a message block, the programmer can call the *release()* method on it. If the message block was reference counted, then the *release()* will cause the count to decrement until the count reaches zero, after which the message block and its associated data blocks are removed from memory.

Message Queues in ACE

As mentioned earlier, ACE has several different types of message queues, which in general can be divided into two categories, static and dynamic. The static queue is a general purpose message queue named *ACE_Message_Queue* (as if you couldn't guess) whereas the dynamic message queues (*ACE_Dynamic_Message_Queue)* are real-time message queues. The major difference between these two types of queues is that messages on static queues have static priority, i.e. once the priority is set it does not change. On the other hand, in the dynamic message queues, the priority of messages may change dynamically, based on parameters such as execution time and deadline.

The following example illustrates how to create a simple static message queue and then how to enqueue and dequeue message blocks onto it.

Example 1

```
#include "ace/Message_Queue.h"
#include "ace/Get_Opt.h"
#define SIZE_BLOCK 1
#define NO_MSGS 10

class QTest{
public:
QTest():no_msgs_(NO_MSGS){
   //First create a message queue of default size.
   if(!(this->mq_=new ACE_Message_Queue<ACE_NULL_SYNCH> ()))
      ACE_DEBUG((LM_ERROR,"Error in message queue initialization \n"));
   }

int start_test(){
  for(int i=0; i<no_msgs_;i++){
```

```
   //create a new message block of size 1
   ACE_Message_Block *mb= new ACE_Message_Block(SIZE_BLOCK);


   //Insert data into the message block using the rd_ptr
   *mb->wr_ptr()=i;

   //Be careful to advance the wr_ptr
   mb->wr_ptr(1);

   //Enqueue the message block onto the message queue
   if(this->mq_->enqueue_prio(mb)==-1){
      ACE_DEBUG((LM_ERROR,"\nCould not enqueue on to mq!!\n"));
      return -1;
      }

   ACE_DEBUG((LM_INFO,"EQ'd data: %d\n",*mb->rd_ptr()));
   } //end for

//Now dequeue all the messages
this->dequeue_all();
return 0;
   }

void dequeue_all(){
   ACE_DEBUG((LM_INFO,"\n\nBeginning DQ \n"));
   ACE_DEBUG((LM_INFO,"No. of Messages on Q:%d Bytes on Q:%d \n"
        ,mq_->message_count(),mq_->message_bytes()));
   ACE_Message_Block *mb;

   //dequeue the head of the message queue until no more messages are
   //left
   for(int i=0;i<no_msgs_;i++){
      mq_->dequeue_head(mb);
      ACE_DEBUG((LM_INFO,"DQ'd data %d\n",*mb->rd_ptr()));
      }
   }
private:
   ACE_Message_Queue<ACE_NULL_SYNCH> *mq_;
   int no_msgs_;
};


int main(int argc, char* argv[]){
   QTest test;
   if(test.start_test()<0)
      ACE_DEBUG((LM_ERROR,"Program failure \n"));
}
```

The above example illustrates several methods of the message queue class. The example consists of a single QTest class which instantiates a message queue of default size with ACE_NULL_SYNCH locking. The locks (a mutex and a condition variable) are used by the message queue to

- Ensure the safety of the reference count maintained by message blocks against race conditions when accessed by multiple threads.
- To "wake up" all threads that are sleeping because the message queue was empty or full.

In this example, since there is just a single thread, the template synchronization parameter for the message queue is set to null (*ACE_NULL_SYNCH* which means use *ACE_Null_Mutex* and *ACE_Null_Condition*). The `start_test()` method of `QTest` is then called, which enters a loop that creates and enqueues messages onto the message queue. A simple constructor of *ACE_Message_Block* is used to create message blocks of `SIZE_BLOCK` size. This constructor creates a data buffer of size `SIZE_BLOCK` and manages the memory for it automatically (i.e. the memory will be released when the message block is deleted). The *wr_ptr* is then obtained (using the *wr_ptr()* accessor method) and its value is set to the message number. After this the *wr_ptr* is advanced forward. The *enqueue_prio()* method of the message queue is then used to actually enqueue the message block onto the underlying message queue.

After `no_msgs_` message blocks have been created, initialized and inserted onto the message queue, `start_test()` calls the `dequeue_all()` method. This method in turn dequeues each of the messages from the message queue using the *dequeue_head()* method of *ACE_Message_Queue.*

Water Marks

Water marks are used in message queues to indicate when the message queue has too much data on it (the message queue has reached the high water mark) or when the message queue has an insufficient amount of data on it (the message queue has reached its low water mark). Both these marks are used for flow control - for example the low_water_mark may be used to avoid situations like the "silly window syndrome" in TCP and the high_water_mark may be used to "stem" or slow down a sender or producer of data.

The message queues in ACE achieve this functionality by maintaining a count of the total amount of data in bytes that has been enqueued. Thus, whenever a new message block is enqueued on to the message queue, it will first determine its length, then check if it can enqueue the message block (i.e. make sure that the message queue doesn't exceed its high water mark if this new message block is enqueued). If the message queue cannot enqueue the data and it possesses a lock (i.e. ACE_SYNC is used and not ACE_NULL_SYNCH as the template parameter to the message queue), it will block the caller until sufficient room is available, or until the timeout in the enqueue method expires. If the timeout expires or if the queue possessed a null lock, then the enqueue method will return with a value of -1, indicating that it was unable to enqueue the message.

Similarly, when the *dequeue_head* method of *ACE_Message_Queue* is called, it checks to make sure that after dequeuing the amount of data left is more then the low water mark. If this is not the case, it blocks if the queue has a lock otherwise it returns -1, indicating failure (the same way the enqueue methods work).

There are two methods which can be used to set and get the water marks that are

```
   //get the high water mark
    size_t high_water_mark(void)
   //set the high water mark
   void high_water_mark(size_t hwm);
   //get the low water_mark
   size_t low_water_mark(void)
   //set the low water_mark
   void low_water_mark(size_t lwm)
```

Using Message Queue Iterators

As is common with other container classes, forward and reverse iterators are available for message queues in ACE. These iterators are named *ACE_Message_Queue_Iterator* and *ACE_Message_Queue_Reverse_Iterator*. Each of these require a template parameter which is used for synchronization while traversing the message queue. If multiple threads are using the message queue, then this should be set to *ACE_SYNCH* - otherwise it may be set to *ACE_NULL_SYNCH*. When an iterator object is created, its constructor must be passed a reference to the message queue we wish it to iterate over.

The following example illustrates the water marks and the iterators:

Example 2

```
#include ”ace/Message_Queue.h”
#include ”ace/Get_Opt.h”
#include ”ace/Malloc_T.h”
#define SIZE_BLOCK 1

class Args{
public:
Args(int argc, char*argv[],int& no_msgs,
ACE_Message_Queue<ACE_NULL_SYNCH>* &mq){
   ACE_Get_Opt get_opts(argc,argv,”h:l:t:n:xsd”);
   while((opt=get_opts())!=-1)
      switch(opt){
      case ’n’:
        //set the number of messages we wish to enqueue and dequeue
        no_msgs=ACE_OS::atoi(get_opts.optarg);
        ACE_DEBUG((LM_INFO,”Number of Messages %d \n”,no_msgs));
        break;

      case ’h’:
        //set the high water mark
        hwm=ACE_OS::atoi(get_opts.optarg);
        mq->high_water_mark(hwm);
        ACE_DEBUG((LM_INFO,”High Water Mark %d msgs \n”,hwm));
        break;
      case ’l’:
        //set the low water mark
        lwm=ACE_OS::atoi(get_opts.optarg);
        mq->low_water_mark(lwm);
        ACE_DEBUG((LM_INFO,”Low Water Mark %d msgs \n”,lwm));
```

```
        break;
      default:
        ACE_DEBUG((LM_ERROR,
                   "Usage -n<no. messages> -h<hwm> -l<lwm>\n"));
        break;
      }
   }

private:
   int opt;
   int hwm;
   int lwm;
};



class QTest{
public:
QTest(int argc, char*argv[]){
   //First create a message queue of default size.
   if(!(this->mq_=new ACE_Message_Queue<ACE_NULL_SYNCH> ()))
      ACE_DEBUG((LM_ERROR,"Error in message queue initialization \n"));

   //Use the arguments to set the water marks and the no of messages
   args_ = new Args(argc,argv,no_msgs_,mq_);
   }
int start_test(){
  for(int i=0; i<no_msgs_;i++){
   //Create a new message block of data buffer size 1
   ACE_Message_Block * mb= new ACE_Message_Block(SIZE_BLOCK);

   //Insert data into the message block using the rd_ptr
   *mb->wr_ptr()=i;

   //Be careful to advance the wr_ptr
   mb->wr_ptr(1);

   //Enqueue the message block onto the message queue
   if(this->mq_->enqueue_prio(mb)==-1){
      ACE_DEBUG((LM_ERROR,"\nCould not enqueue on to mq!!\n"));
      return -1;
      }

   ACE_DEBUG((LM_INFO,"EQ'd data: %d\n",*mb->rd_ptr()));
      }
   //Use the iterators to read
        this->read_all();

   //Dequeue all the messages
   this->dequeue_all();
        return 0;
   }

void read_all(){
```

```
   ACE_DEBUG((LM_INFO,"No. of Messages on Q:%d Bytes on Q:%d \n"
            ,mq_->message_count(),mq_->message_bytes()));
   ACE_Message_Block *mb;

   //Use the forward iterator
   ACE_DEBUG((LM_INFO,"\n\nBeginning Forward Read \n"));
   ACE_Message_Queue_Iterator<ACE_NULL_SYNCH> mq_iter_(*mq_);
   while(mq_iter_.next(mb)){
      mq_iter_.advance();
      ACE_DEBUG((LM_INFO,"Read data %d\n",*mb->rd_ptr()));
      }

   //Use the reverse iterator
   ACE_DEBUG((LM_INFO,"\n\nBeginning Reverse Read \n"));
   ACE_Message_Queue_Reverse_Iterator<ACE_NULL_SYNCH>
                                  mq_rev_iter_(*mq_);
   while(mq_rev_iter_.next(mb)){
      mq_rev_iter_.advance();
      ACE_DEBUG((LM_INFO,"Read data %d\n",*mb->rd_ptr()));
      }

   }

void dequeue_all(){
   ACE_DEBUG((LM_INFO,"\n\nBeginning DQ \n"));
   ACE_DEBUG((LM_INFO,"No. of Messages on Q:%d Bytes on Q:%d \n",
            mq_->message_count(),mq_->message_bytes()));
   ACE_Message_Block *mb;

   //dequeue the head of the message queue until no more messages
   //are left
   for(int i=0;i<no_msgs_;i++){
      mq_->dequeue_head(mb);
      ACE_DEBUG((LM_INFO,"DQ'd data %d\n",*mb->rd_ptr()));
      }
   }

private:
   Args *args_;
   ACE_Message_Queue<ACE_NULL_SYNCH> *mq_;
   int no_msgs_;
};



int main(int argc, char* argv[]){
   QTest test(argc,argv);
   if(test.start_test()<0)
      ACE_DEBUG((LM_ERROR,"Program failure \n"));

}
```

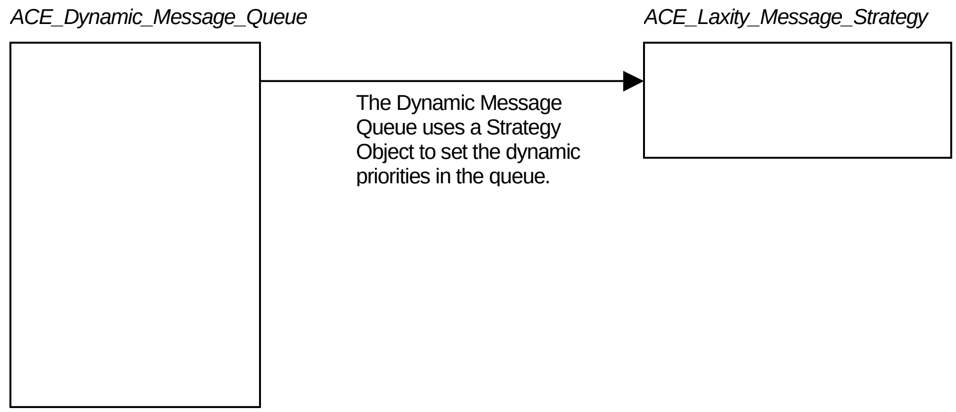
This example uses the *ACE_Get_Opt* class (See Appendix for more on this utility class) to obtain the low and high water marks (in the `Args` class). The low and high water marks are set using the *low_water_mark()* and *high_water_mark()* accessor functions. Besides this, there is a `read_all()` method which uses the *ACE_Message_Queue_Iterator* and *ACE_Message_Queue_Reverse_Iterator* to first read in the forward and then in the reverse direction.

Dynamic or Real-Time Message Queues

As was mentioned above, dynamic message queues are queues in which the priority of the messages enqueued change with time. Such message queues are thus inherently more useful in real-time applications, where such kind of behavior is desirable.

ACE currently provides for two types of dynamic message queues, deadline-based and laxity-based (see [IX]). The deadline-based message queues use the deadlines of each of the messages to set their priority. The message block on the queue which has the earliest deadline will be dequeued first when the *dequeue_head()* method is called using the earliest deadline first algorithm. The laxity-based message queues, however, use both execution time and deadline together to calculate the laxity, which is then used to prioritize each message block. The laxity is useful, since when scheduling by deadline, it may be possible that a task is scheduled which has the earliest deadline, but has such a long execution time that it will not complete even if it is scheduled immediately. This negatively affects other tasks, since it may block out tasks which are schedulable. The laxity will take into account this long execution time and make sure that if the task will not complete, that it is not scheduled. The scheduling in laxity queues is based on the minimum laxity first algorithm.

Both laxity-based message queues and deadline-based message queues are implemented as *ACE_Dynamic_Message_Queue*'s. ACE uses the **STRATEGY** pattern to provide for dynamic queues with different scheduling characteristics. Each message queue uses a different "strategy" object to dynamically set priorities of the messages on the message queue. These "strategy" objects each encapsulate a different algorithm to calculate priorities based on execution time, deadlines, etc., and are called to do so whenever messages are enqueued or removed from the message queue. (For more on the **STRATEGY** pattern please see the reference "*Design Patterns*"). The message strategy patterns derive from *ACE_Dynamic_Message_Strategy* and currently there are two strategies available: *ACE_Laxity_Message_Strategy* and *ACE_Deadline_Message_Strategy*. Therefore, to create a "laxity-based" dynamic message queue, an *ACE_Laxity_Message_Strategy* object must be created first. Subsequently, an *ACE_Dynamic_Message_Queue* object should be instantiated, which is passed the new strategy object as one of the parameters to its constructor.



Creating Message Queues

To simplify the creation of these different types of message queues, ACE provides for a concrete message queue factory named *ACE_Message_Queue_Factory*, which creates message queues of the appropriate type using a variant of the **FACTORY METHOD** pattern. (For more on the **FACTORY METHOD** pattern please see reference "*Design Patterns"*). The message queue factory has three static factory methods to create three different types of message queues:

*static ACE_Message_Queue<ACE_SYNCH_USE> **
create_static_message_queue ()

*static ACE_Dynamic_Message_Queue<ACE_SYNCH_USE> **
create_deadline_message_queue ();

*static ACE_Dynamic_Message_Queue<ACE_SYNCH_USE> **
create_laxity_message_queue ();

Each of these methods returns a pointer to the message queue it has just created. Notice that all methods are static and that the *create_static_message_queue()* method returns an *ACE_Message_Queue* whereas the other two methods return an *ACE_Dynamic_Message_Queue*.

This simple example illustrates the creation and use of dynamic and static message queues.

Example 3
```
#include ”ace/Message_Queue.h”
#include ”ace/Get_Opt.h”
#include ”ace/OS.h”

class Args{
public:
Args(int argc, char*argv[],int& no_msgs, int&
time,ACE_Message_Queue<ACE_NULL_SYNCH>* &mq){
   ACE_Get_Opt get_opts(argc,argv,”h:l:t:n:xsd”);
   while((opt=get_opts())!=-1)
```

```
      switch(opt){
      case 't':
        time=ACE_OS::atoi(get_opts.optarg);
        ACE_DEBUG((LM_INFO,"Time: %d \n",time));
        break;
      case 'n':
        no_msgs=ACE_OS::atoi(get_opts.optarg);
        ACE_DEBUG((LM_INFO,"Number of Messages %d \n",no_msgs));
        break;
      case 'x':
        mq=ACE_Message_Queue_Factory<ACE_NULL_SYNCH>::
                   create_laxity_message_queue();
        ACE_DEBUG((LM_DEBUG,"Creating laxity q\n"));
        break;
      case 'd':
        mq=ACE_Message_Queue_Factory<ACE_NULL_SYNCH>::
                   create_deadline_message_queue();
        ACE_DEBUG((LM_DEBUG,"Creating deadline q\n"));
        break;
      case 's':
        mq=ACE_Message_Queue_Factory<ACE_NULL_SYNCH>::
                   create_static_message_queue();
        ACE_DEBUG((LM_DEBUG,"Creating static q\n"));
        break;
      case 'h':
        hwm=ACE_OS::atoi(get_opts.optarg);
        mq->high_water_mark(hwm);
        ACE_DEBUG((LM_INFO,"High Water Mark %d msgs \n",hwm));
        break;
      case 'l':
        lwm=ACE_OS::atoi(get_opts.optarg);
        mq->low_water_mark(lwm);
        ACE_DEBUG((LM_INFO,"Low Water Mark %d msgs \n",lwm));
        break;
      default:
        ACE_DEBUG((LM_ERROR,"Usage specify queue type\n"));
        break;
      }
   }

private:
   int opt;
   int hwm;
   int lwm;
};



class QTest{
public:
QTest(int argc, char*argv[]){
   args_ = new Args(argc,argv,no_msgs_,time_,mq_);
   array_ =new ACE_Message_Block*[no_msgs_];
   }
```

```
int start_test(){
   for(int i=0; i<no_msgs_;i++){
      ACE_NEW_RETURN (array_[i], ACE_Message_Block (1), -1);
      set_deadline(i);
      set_execution_time(i);
      enqueue(i);
      }

   this->dequeue_all();
   return 0;
   }

//Call the underlying ACE_Message_Block method msg_deadline_time() to
//set the deadline of the message.
void set_deadline(int msg_no){
   float temp=(float) time_/(msg_no+1);
   ACE_Time_Value tv;
   tv.set(temp);
   ACE_Time_Value deadline(ACE_OS::gettimeofday()+tv);
   array_[msg_no]->msg_deadline_time(deadline);
   ACE_DEBUG((LM_INFO,"EQ'd with DLine %d:%d,",
deadline.sec(),deadline.usec()));
   }

//Call the underlying ACE_Message_Block method to set the execution time
void set_execution_time(int msg_no){
   float temp=(float) time_/10*msg_no;
   ACE_Time_Value tv;
   tv.set(temp);
   ACE_Time_Value xtime(ACE_OS::gettimeofday()+tv);
   array_[msg_no]->msg_execution_time (xtime);
   ACE_DEBUG((LM_INFO,"Xtime %d:%d,",xtime.sec(),xtime.usec()));
   }

void enqueue(int msg_no){
   //Set the value of data at the read position
   *array_[msg_no]->rd_ptr()=msg_no;
   //Advance write pointer
   array_[msg_no]->wr_ptr(1);
   //Enqueue on the message queue
   if(mq_->enqueue_prio(array_[msg_no])==-1){
      ACE_DEBUG((LM_ERROR,"\nCould not enqueue on to mq!!\n"));
      return;
      }
   ACE_DEBUG((LM_INFO,"Data %d\n",*array_[msg_no]->rd_ptr()));
   }

void dequeue_all(){
   ACE_DEBUG((LM_INFO,"Beginning DQ \n"));
   ACE_DEBUG((LM_INFO,"No. of Messages on Q:%d Bytes on Q:%d \n",
        mq_->message_count(),mq_->message_bytes()));
   for(int i=0;i<no_msgs_ ;i++){
      ACE_Message_Block *mb;
```

```
      if(mq_->dequeue_head(mb)==-1){
        ACE_DEBUG((LM_ERROR,"\nCould not dequeue from mq!!\n"));
        return;
        }
      ACE_DEBUG((LM_INFO,"DQ'd data %d\n",*mb->rd_ptr()));
      }
   }
private:
   Args *args_;
   ACE_Message_Block **array_;
   ACE_Message_Queue<ACE_NULL_SYNCH> *mq_;
   int no_msgs_;
   int time_;
};

int main(int argc, char* argv[]){
   QTest test(argc,argv);
   if(test.start_test()<0)
      ACE_DEBUG((LM_ERROR,"Program failure \n"));

}
```

The above example is very similar to the previous examples, but adds the dynamic message queues into the picture. In the `Args` class, we have added options to create all the different types of message queues using the *ACE_Message_Queue_Factory*. Furthermore, two new methods have been added to the `QTest` class to set the deadlines and execution times of each of the message blocks as they are created (`set_deadline()`and `set_execution_time()`). These methods use the *ACE_Message_Block* methods *msg_execution_time()* and *msg_deadline_time()*. Note that these methods take the absolute and NOT the relative time, which is why they are used in conjunction with the *ACE_OS::gettimeofday()* method.

The deadlines and execution times are set with the help of a `time` parameter. The deadline is set such that the first message will have the latest deadline and should be scheduled last in the case of deadline message queues. Both the execution time and deadline are taken into account when using the laxity queues, however.

Chapter

A

Appendix: Utility Classes

Utility classes used in ACE

Address Wrapper Classes

ACE_INET_Addr

The *ACE_INET_Addr* is a wrapper class around the Internet domain address family (AF_INET). This class derives from *ACE_Addr* in ACE. The various constructors of this class can be used to initialize the object to have a certain IP address and port. Besides this, the class has several set and get methods and has overloaded the comparison operations i.e., = = operator and the != operator. For further details on how to use this class see the reference manual.

ACE_UNIX_Addr

The *ACE_UNIX_Addr* class is a wrapper class around the UNIX domain address family (AF_UNIX) and also derives from *ACE_Addr*. This class has functionality similar to the *ACE_INET_Addr* class. For further details see the reference manual.

Time wrapper classes

ACE_Time_Value

Logging with ACE_DEBUG and ACE_ERROR

The *ACE_DEBUG* and *ACE_ERROR* macros are useful macros for printing and logging debug and error information. Their usage has been illustrated throughout this tutorial.

The following format specifiers are available for these macros. Each of these options is prefixed by ’%’, as in printf format strings:

Format Specifier	Description
’a’	exit the program at this point (var-argument is the exit status!)
’c’	print a character
’i’, ’d’	print a decimal number
’I’,	indent according to nesting depth
’e’, ’E’, ’f’, ’F’, ’g’, ’G’	print a double
’l’	print line number where an error occurred
’N’	print file name where the error occurred.
’n’	print the name of the program (or "<unknown>" if not set)
’o’	print as an octal number
’P’	print out the current process id
’p’	print out the appropriate errno value from sys_errlist
’r’	call the function pointed to by the corresponding argument
’R’	print return status
’S’	print out the appropriate _sys_siglist entry corresponding to var-argument.
’s’	print out a character string
’T’	print timestamp in hour:minute:sec: usec format.
’D’	print timestamp in month/day/year hour:minute:sec:usec format.
’t’	print thread id (1 if single-threaded)
’u’	print as unsigned int
’X’, ’x’	print as a hex number
’%’	print out a single percent sign, ’%’

Obtaining command line arguments

ACE_Get_Opt

This utility class is used to obtain arguments from the user and is based on the `getopt()` `<3c>` function in `stdlib`. The constructor of this class is passed in a string called the `optstring`, which specifies the switches that the application wishes to respond to. If the switch letter is followed by a colon, it means that the switch also expects an argument. For example, if the optstring is "`ab:c`", then the application expect "`-a`" and "`-c`" without an argument and "`-b`" with an argument. For example, the application would be run as:

```
MyApplication –a –b 10 –c
```

The `()` operator has been overloaded and is used to scan the elements of `argv` for the options specified in the option string.

The following example will help make it clear how to use this class to obtain arguments from the user.

```
Example
#include "ace/Get_Opt.h"
int main (int argc, char *argv[])
{
   //Specify option string so that switches b, d, f and h all expect
   //arguments. Switches a, c, e and g expect no arguments.
   ACE_Get_Opt get_opt (argc, argv, "ab:cd:ef:gh:");
   int c;

   //Process the scanned options with the help of the overloaded ()
   //operator.
   while ((c = get_opt ()) != EOF)
      switch (c)
      {
      case 'a':
        ACE_DEBUG ((LM_DEBUG, "got a\n"));
        break;
      case 'b':
        ACE_DEBUG ((LM_DEBUG, "got b with arg %s\n",
                                    get_opt.optarg));
        break;
      case 'c':
        ACE_DEBUG ((LM_DEBUG, "got c\n"));
        break;
      case 'd':
        ACE_DEBUG ((LM_DEBUG, "got d with arg %s\n",
                                     get_opt.optarg));
        break;
      case 'e':
        ACE_DEBUG ((LM_DEBUG, "got e\n"));
        break;
      case 'f':
```

```
        ACE_DEBUG ((LM_DEBUG, "got f with arg %s\n",
                  get_opt.optarg));
        break;
      case 'g':
        ACE_DEBUG ((LM_DEBUG, "got g\n"));
        break;
      case 'h':
        ACE_DEBUG ((LM_DEBUG, "got h with arg %s\n",
                                    get_opt.optarg));
        break;
      default:
        ACE_DEBUG ((LM_DEBUG, "got %c, which is unrecognized!\n",
                                                        c));
        break;
      }

//optind indicates how much of argv has been scanned so far, while
//get_opt hasn't returned EOF. In this case it indicates the index in
//argv from where the option switches have been fully recognized and the
//remaining elements must be scanned by the called himself.
  for (int i = get_opt.optind; i < argc; i++)
    ACE_DEBUG ((LM_DEBUG, "optind = %d, argv[optind] = %s\n",
                i, argv[i]));

  return 0;
}
```

For further details on using this utility wrapper class, please see the reference manual.

ACE_Arg_Shifter

This ADT shifts known arguments, or options, to the back of the `argv` vector, so deeper levels of argument parsing can locate the yet-unprocessed arguments at the beginning of the vector.

The *ACE_Arg_Shifter* copies the pointers of the `argv` vector into a temporary array. As the *ACE_Arg_Shifter* iterates over the temp, it places known arguments in the rear of the `argv` and unknown ones in the beginning. So, after having visited all the arguments in the temp vector, *ACE_Arg_Shifter* has placed all the unknown arguments in their original order at the front of `argv`.

This class is also very useful in parsing options from the command line. The following example will help illustrate this:

Example

```
#include "ace/Arg_Shifter.h"
int main(int argc, char *argv[]){
   ACE_Arg_Shifter arg(argc,argv);

   while(arg.is_anything_left ()){
      char *current_arg=arg.get_current();

      if(ACE_OS::strcmp(current_arg,"-region")==0){
            arg.consume_arg();
            ACE_OS::printf("<region>= %s \n",arg.get_current());
```

```
        }
      else if(ACE_OS::strcmp(current_arg,"-tag")==0){
            arg.consume_arg();
            ACE_OS::printf("<tag>= %s \n",arg.get_current());
        }
      else if(ACE_OS::strcmp(current_arg,"-view_uuid")==0){
            arg.consume_arg();
            ACE_OS::printf("<view_uuid>=%s\n",arg.get_current());
        }
      arg.consume_arg();
   }
   for(int i=0;argv[i]!=NULL;i++)
      ACE_DEBUG((LM_DEBUG,"Resultant vector": %s \n",argv[i]));
}
```

If the above example is run as follows:

```
.../tests<330> arg -region missouri -tag 10 -view_uuid syyid  -teacher
schmidt -student tim
```

The results obtained are:

```

<region> missouri
<tag>= 10
<view_uuid>=syyid
Resultant Vector: tim
Resultant Vector: -student
Resultant Vector: schmidt
Resultant Vector: -teacher
Resultant Vector: syyid
Resultant Vector: -view_uuid
Resultant Vector: 10
Resultant Vector: -tag
Resultant Vector: missouri
Resultant Vector: -region
Resultant Vector: ./arg
```

Chapter

R

References

References and Bibliography

This is a list of the references that have been mentioned in the text

[I] *"An introduction to programming with threads"*, Andrew Birell, Digital Systems Research Center, 1989.
[II] *"Active Object, An object behavioral Pattern for Concurrent Programming"*, Douglas C. Schmidt, Greg Lavender. Pattern Languages of Programming Design 2, Addison Wesley 1996.
[III] *"A model of Concurrent Computation in Distributed Systems"*. MIT Press, 1986.
(Also see: *http://www-osl.cs.uiuc.edu/)*
[IV] *"A Polymorphic Future and First-Class Function Type for Concurrent Object-Oriented Programming in C++"*. R.G. Lavender and D.G. Kafura. Forthcoming 1995.
(Also see: *http://www.cs.utexas.edu/users/lavender/papers/futures.ps*)
[V] *"Foundation Patterns"*, Dwight Deugo. Proceedings PLoP' 98.
[VI] *"Design Patterns: Elements of reusable Object-Oriented Software"*, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Addison Wesley Longman Publishing 1995.
[VII] *"Hashed and Hierarchical Timing Wheels: data structures to efficiently implement a timer facility"*, George Varghese. IEEE Trans Networking, Dec 97. Revised from a 1984 SOSP paper; used in X kernel, Siemens and DEC OS etc.
[VIII] *"The x-Kernel: An architecture for implementing network protocols"*. N. C. Hutchinson and L. L. Peterson. IEEE Transactions on Software Engineering, 17(1):64-76, Jan. 1991.
(Also see *http://www.cs.arizona.edu/xkernel)*
[IX] "*Evaluating Strategies for Real-Time CORBA Dynamic Scheduling"*, Douglas Schmidt, Chris Gill and David Levine (updated June 20th), Submitted to the International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware, guest editor Wei Zhao.