The Design and Use of the ACE Reactor

An Object-Oriented Framework for Event Demultiplexing

Douglas C. Schmidt and Irfan Pyarali
{schmidt,irfan}@cs.wustl.edu
Department of Computer Science
Washington University, St. Louis 63130[1]

1 Introduction

This article describes the design and implementation of the Reactor pattern [1] contained in the ACE framework [2]. The Reactor pattern handles service requests that are delivered concurrently to an application by one or more clients. Each service of the application is implemented by a separate *event handler* that contains one or more methods responsible for processing service-specific requests.

In the implementation of the Reactor pattern described in this paper, event handler dispatching is performed by an `ACE_Reactor`. The `ACE_Reactor` combines the demultiplexing of input and output (I/O) events with other types of events, such as timers and signals. At the core of the `ACE_Reactor` implementation is a synchronous event demultiplexer, such as `select` [3] or `WaitForMultipleObjects` [4]. When the demultiplexer indicates the occurrence of designated events, the `ACE_Reactor` automatically dispatches the method(s) of pre-registered event handlers, which perform application-specified services in response to the events.

This paper is organized as follows: Section 2 describes the primary features in the `ACE_Reactor` framework; Section 3 outlines the OO design of the `ACE_Reactor` implementation [2]; Section 4 examines several examples that demonstrate how the `ACE_Reactor` simplifies the development of concurrent, event-driven network applications; Section 5 describes design rules to follow when using the `ACE_Reactor` to develop event-driven applications; and Section 6 presents concluding remarks.

2 Features of the ACE Reactor

The `ACE_Reactor` provides an OO demultiplexing and dispatching framework that simplifies the development of event-driven applications by providing the following features:

Uniform OO demultiplexing and dispatching interface: Applications that use the `ACE_Reactor` do not directly call low-level OS event demultiplexing APIs, such as `select` or `WaitForMultipleObjects`. Instead, they create concrete event handlers by inheriting from the `ACE_Event_Handler` base class. This class specifies virtual methods that handle various types of events, such as I/O events, timer events, signals, and synchronization events. Applications that use the Reactor framework create concrete event handlers and register them with the `ACE_Reactor`. Figure 1 shows the key components in the `ACE_Reactor`. This figure depicts concrete event handlers that implement

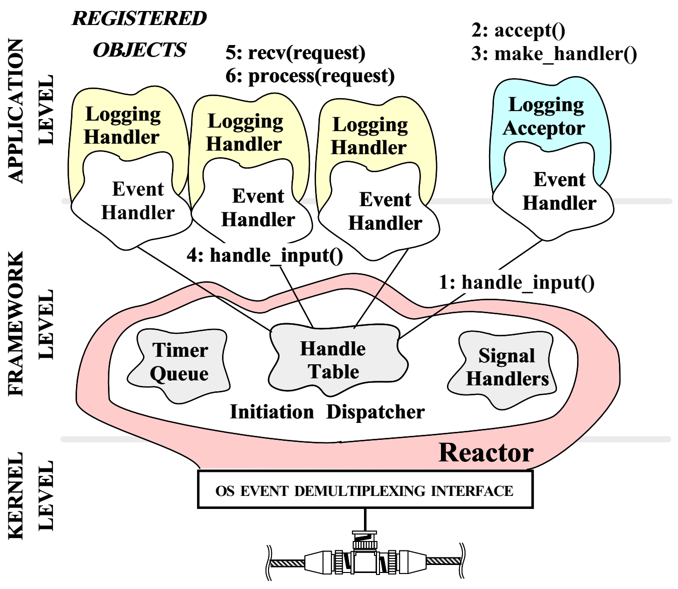


Figure 1: The Reactor Components

the logging server described in Section 4.

Automate event handler dispatching: When activity occurs on handles managed by an `ACE_Reactor`, it automatically invokes the appropriate virtual methods on the pre-registered concrete event handlers. C++ event handler objects are registered with the `ACE_Reactor`, rather than stand-alone functions. Registering objects as event handlers allows state to be retained between hook method invocations on concrete event handlers. This style of OO programming is useful for developing event handlers that hold state across multiple callbacks by the `ACE_Reactor` dispatcher.

[1] This work was supported in part by Siemens SCR.

Support transparent extensibility: The functionality of the `ACE_Reactor` and its registered event handlers can be extended transparently *without* modifying or recompiling existing code. To support this degree of extensibility, the Reactor framework employs inheritance and dynamic binding to decouple the following two responsibilities:

1. *Lower-level event demultiplexing and dispatching mechanisms* – Low-level mechanisms managed by the `ACE_Reactor` include detecting events on multiple I/O handles, expiring timers, and dispatching the appropriate event handler methods to process these events.
2. *Higher-level policies defined by applications to process events* – Higher-level policies performed by application-specified concrete event handlers include connection establishment strategies, data encoding and decoding, and processing of service requests from clients. For instance, the TAO [5] CORBA ORB uses the Reactor framework to separate its low-level event demultiplexing mechanisms from its higher-level policies for GIOP connection management and protocol processing [5].

Increase reuse: The `ACE_Reactor`'s demultiplexing and dispatching mechanisms can be reused by many network applications. By *reusing*, rather than *reinventing*, these mechanisms, developers can concentrate on higher-level application-specific event handler policies, rather than wrestling repeatedly with low-level event demultiplexing and dispatching mechanisms.

Developers who write programs using low-level event demultiplexing operations like `select` and `WaitForMultipleObjects` directly must reimplement, debug, and tune the same demultiplexing and dispatching code for every application. In contrast, all applications that utilize the `ACE_Reactor` automatically reuse its features, as well as future enhancements and optimizations.

Eliminate common error-prone programming details: The `ACE_Reactor` shields application developers from error-prone details associated with programming low-level OS event demultiplexing APIs like `select`. These error-prone details involve setting and clearing bitmasks, detecting and responding to interrupts, managing internal locks, and dispatching hook methods for I/O and timeout processing. For instance, the `ACE_Reactor` eliminates several subtle causes of errors with `select` involving the misuse of `fd_set` bitmasks.

Improve portability: The `ACE_Reactor` runs atop several event demultiplexing mechanisms, including `WaitForMultipleObjects`, which is available on Win32, and `select`, which is available on Win32 and UNIX. The `ACE_Reactor` shields applications from portability differences between the underlying event demultiplexing mechanisms. As illustrated in Figure 6, the `ACE_Reactor` exports the same interface to applications, regardless of the native OS APIs. Moreover, the `ACE_Reactor` uses design patterns like Bridge [6] to enhance its internal portability. Thus, porting the `ACE_Reactor` from `select` to `WaitForMultipleObjects` required only localized changes to the framework [7].

Thread-safety: The Reactor framework is fully thread-safe. Therefore multiple threads can safely share a single `ACE_Reactor`. Likewise, multiple `ACE_Reactor`s can run in separate threads within a process. The Reactor framework provides the necessary synchronization mechanisms to prevent race conditions and intra-class method deadlock [8].

Efficient demultiplexing: The `ACE_Reactor` performs its event demultiplexing and dispatching logic efficiently. For instance, the `select`-based `ACE_Reactor` uses the `ACE_Handle_Set` class described in Section 3.2 to avoid examining `fd_set` bitmasks one bit at a time. This optimization is based on a sophisticated algorithm that uses the C++ exclusive-or operator to reduce run-time complexity from *O(number of total bits)* to *O(number of enabled bits)*, which can substantially reduce run-time overhead.

3 The OO Design of the Reactor Framework

This section describes the OO design of the `ACE_Reactor` framework. We focus on the structure of its components and key design decisions. Where appropriate, implementation details are also discussed. Section 3.1 outlines the OS platform-independent components and Section 3.2 covers the platform-dependent components.

3.1 Platform-Independent Class Components

This subsection summarizes the platform-independent classes in the Reactor framework, which include the `ACE_Reactor`, `ACE_Time_Value`, `ACE_Timer_Queue`, and `ACE_Event_Handler`.

3.1.1 The ACE_Reactor Class

The `ACE_Reactor` defines the public interface for the Reactor framework. Figure 2 illustrates the key public methods in the `ACE_Reactor` class. The methods in the `ACE_Reactor` can be grouped into the following general categories:

Manager methods: The constructor and `open` methods create and initialize objects of the `ACE_Reactor` by dynamically allocating various implementation objects described in Section 3.2.1 and 3.2.2. The `ACE_Reactor`'s destructor and `close` methods deallocate these objects. In addition, to support the common use-case of one event loop per application process, there is a static `instance` method that returns a pointer to a singleton `ACE_Reactor`, which is created and managed by the Singleton pattern [6].

```
class ACE_Reactor
{
public:
  enum { DEFAULT_SIZE = FD_SETSIZE };

  // = Singleton access point.
  static ACE_Reactor *instance (void);

  // = Initialization and termination methods.

  // Initialize a Reactor instance that may
  // contain <size> entries (<restart> indicates
  // to restart system calls after interrupts).
  ACE_Reactor (int size, int restart = 0);
  virtual int open (int size = DEFAULT_SIZE,
                    int restart = 0);

  // Perform cleanup activities to close down
  // an instance of an <ACE_Reactor>.
  void close (void);
  ˜ACE_Reactor (void);

  // = I/O-based event handler methods.

  // Register an <ACE_Event_Handler> object according
  // to the <ACE_Reactor_Mask>(s), e.g., READ_MASK,
  // WRITE_MASK, etc.
  virtual int register_handler (ACE_Event_Handler *,
                                ACE_Reactor_Mask);

  // Remove the handler associated with the
  // appropriate <ACE_Reactor_Mask>(s).
  virtual int remove_handler (ACE_Event_Handler *,
                              ACE_Reactor_Mask);

  // = Timer-based event handler methods.

  // Schedule an <event_handler> that will expire
  // after <delay> amount of time.  If it expires
  // <arg> is passed in as the value to the
  // <event_handler>'s <handle_timeout> callback
  // method.  If <interval> is != to
  // <ACE_Time_Value::zero> it is used to
  // reschedule the <event_handler> automatically.
  // Returns a timer id that can be used to cancel
  // the timer.
  virtual long schedule_timer
       (ACE_Event_Handler *,
        const void *act,
        const ACE_Time_Value &delta,
        const ACE_Time_Value &interval);

  // Cancel all timers associated with <eh>.
  virtual void cancel_timer (ACE_Event_Handler *eh);

  // Cancel the timer that matches the <timer_id>.
  virtual void cancel_timer (long timer_id,
                             const void **arg = 0);
  // = Event-loop methods

  // Block process until I/O events occur or timer
  // expires, then dispatch activated handler(s).
  virtual int handle_events (void);

  // Perform a timed event-loop that waits up to <tv>
  // time units for events to occur; if no events
  // occur then 0 is returned, otherwise return
  // <TV> - actual-time-waited.
  virtual int handle_events (ACE_Time_Value &tv);

private:
  // Pointer to the implementation class,
  // e.g., <ACE_Select_Reactor> or
  // <ACE_WFMO_Reactor>.
  Reactor_Impl *reactor_impl_;
};
```

Figure 2: Interface for the `ACE_Reactor`

I/O-related methods: Applications can register concrete event handlers that derive from `ACE_Event_Handler` with an `ACE_Reactor` via its `register_handler` method. Likewise, concrete event handlers can be removed via its `remove_handler` method.

Timer-related methods: The `ACE_Reactor`'s timer strategy orders the event handlers that are scheduled with it according to their timeout deadlines. The methods provided by the `ACE_Reactor`'s timer mechanisms include (1) registering concrete event handlers that will be executed at a user-specified time in the future and (2) canceling one or more previously registered event handlers.

Event-loop methods: After registering its initial concrete event handlers, an application enters an event-loop that typically calls an `ACE_Reactor`'s `handle_events` method repeatedly. This method gathers the handles of all registered concrete event handlers, passes them to the underlying OS event demultiplexing call, *e.g.*, `select` or `WaitForMultipleObjects`, and then can block for an application-specified time interval awaiting the occurrence of various events, such as data events to arrive on socket handles or timer deadlines to expire. Subsequently, when I/O events occur and the handles become "ready," the `ACE_Reactor` notifies the appropriate pre-registered concrete event handlers by invoking their `handle_*` hook method(s) defined by the application to process the event(s).

3.1.2 The ACE_Event_Handler Class

This base class specifies the interface used by the `ACE_Reactor` to control and coordinate the demultiplexing and dispatching of concrete event handlers. The virtual methods in the `ACE_Event_Handler` interface are illustrated in Figure 3.

The `ACE_Reactor` implements its event-driven callback mechanism via *concrete event handlers*, which inherit from `ACE_Event_Handler`. Concrete event handlers typically provide an I/O handle that the `ACE_Reactor` can retrieve via the `get_handle` method. When an application registers a concrete event handler with the `ACE_Reactor`'s `register_handler` method, the `ACE_Reactor` calls back to the `get_handle` method on the concrete event handler to retrieve the underlying I/O handle. If a concrete event handler is used entirely to handle signals or timers, the `get_handle` method can be a no-op.

Concrete event handlers typically override one or more virtual methods in `ACE_Event_Handler` to perform application-defined processing in response to various types of events, including (1) I/O events, *e.g.*, reading, writing, and exceptions, (2) timer events and (3) signals.[2]

The `ACE_Timer_Queue` class described in Section 3.1.4 uses concrete event handlers to process time-based events. When a timer managed by this mechanism expires, the

[2]On Win32 the `ACE_Reactor` can also handle synchronization events, such as transitioning from the non-signaled to signaled state with Win32 mutexes [4].

```
// Handle portability issues.
#elif defined (ACE_HAS_WIN32)
typedef HANDLE ACE_HANDLE;
#if defined (UNIX)
typedef int ACE_HANDLE;
#endif /* UNIX */

typedef u_long ACE_Reactor_Mask;

class ACE_Event_Handler
{
public:
  // These values can be bitwise "or'd" together to
  // instruct the <ACE_Reactor> to check for
  // multiple events on a single ACE_HANDLE.
  enum {
    READ_MASK = (1 << 0),
    WRITE_MASK  = (1 << 2),
    EXCEPT_MASK = (1 << 3),
    ACCEPT_MASK = (1 << 3),
    CONNECT_MASK = (1 << 4),
    TIMER_MASK = (1 << 5),
    SIGNAL_MASK = (1 << 8),
    DONT_CALL = (1 << 9)
  };

  // Returns the I/O handle associated with the
  // derived object.
  virtual ACE_HANDLE get_handle (void) const;

  // Called when event handler is removed from
  // an <ACE_Reactor>.
  virtual int handle_close (ACE_HANDLE,
                            ACE_Reactor_Mask);

  // Called when input becomes available.
  virtual int handle_input (ACE_HANDLE);

  // Called when output is possible.
  virtual int handle_output (ACE_HANDLE);

  // Called when urgent data is available.
  virtual int handle_except (ACE_HANDLE);

  // Called when timer expires, <tv> stores the
  // current time and <act> is the argument given
  // when the handler was scheduled originally.
  virtual int handle_timeout
    (const ACE_Time_Value &tv,
     const void *act = 0);

  // Called when signal is triggered.
  virtual int handle_signal (int signum);
};
```

Figure 3: Interface for `ACE_Event_Handler`

`handle_timeout` method of the associated event handler is invoked by the `ACE_Reactor`. This method is passed both (1) the current time and (2) the `void` * *asynchronous completion token* (ACT) [9], which was passed as a parameter to `schedule_timer` when the event handler was scheduled originally.

When any `handle_*` hook method in a concrete event handler returns < 0, the `ACE_Reactor` automatically invokes that handler's `handle_close` cleanup method. When the `ACE_Reactor` invokes this cleanup method, it passes in the `ACE_Reactor_Mask` value, *e.g.*, READ_MASK, SIGNAL_MASK, etc., corresponding to the `handle_*` hook method that returned -1. Application developers can override the `handle_close` method to perform cleanup activities, such as closing log files or deleting dynamic memory allocated by the object. When the `handle_close` method returns and the concrete event handler is no longer registered to handle any events, the `ACE_Reactor` removes the event handler from from its internal tables.

3.1.3 The ACE_Time_Value Class

This C++ wrapper encapsulates the underlying OS platform's date and time structure, such as the `struct timeval` type defined on most UNIX platforms as follows:

```
struct timeval {
  long secs;
  long usecs;
};
```

Other OS platforms, such as POSIX and Win32, use slightly different time representations. Therefore, the `ACE_Time_Value` class encapsulates these details to provide a portable C++ interface.

The primary methods in the `ACE_Time_Value` class are illustrated in Figure 4. The `ACE_Time_Value` wrapper uses operator overloading to simplify time-based comparisons. Overloading permits the use of standard arithmetic syntax for relational expressions involving time comparisons.

Methods on the `ACE_Time_Value` class are implemented to "normalize" time quantities. Normalization adjusts the two fields in a `timeval` structure to use a canonical encoding scheme that ensures accurate comparisons. For example, after normalization, the quantity `ACE_Time_Value (1, 1000000)` will compare equal to `ACE_Time_Value (2)`. In contrast, a direct bitwise comparison of these non-normalized class values would not detect this equality.

The following code creates two `ACE_Time_Value` objects, which are constructed by adding user-supplied command-line arguments to the current time. The appropriate ordering relationship between the two objects is then displayed:

```
int main (int argc, char *argv[])
{
  if (argc != 3)
```

```
class ACE_Time_Value
{
public:
  // = Initialization methods.
  ACE_Time_Value (long sec = 0, long usec = 0);
  ACE_Time_Value (const timeval &t);

  // = Friend methods.
  // Returns sum of two <ACE_Time_Value>s.
  friend ACE_Time_Value operator +
    (const ACE_Time_Value &lhs,
     const ACE_Time_Value &rhs);
  // Returns difference between two
  // <ACE_Time_Value>s.
  friend ACE_Time_Value operator -
    (const ACE_Time_Value &lhs,
     const ACE_Time_Value &rhs);

  // = operators for normalized <ACE_Time_Value>s.
  friend int operator <
    (const ACE_Time_Value &lhs,
     const ACE_Time_Value &rhs);
  // Other relation operators...

private:
  // ...
};
```

Figure 4: Interface for ACE_Time_Value

```
    ACE_ERROR_RETURN ((LM_ERROR,
                       "usage: %d"
                       "time1 time2\n"),
                       1);

  ACE_Time_Value curtime = ACE_OS::gettimeofday ();
  ACE_Time_Value timer1 = curtime +
    ACE_Time_Value (ACE_OS::atoi (argv[1]));
  ACE_Time_Value timer2 = curtime +
    ACE_Time_Value (ACE_OS::atoi (argv[2]));

  if (timer1 > timer2)
    ACE_DEBUG ((LM_DEBUG,
               "timer 1 is greater\n"));
  else if (timer2 > timer1)
    ACE_DEBUG ((LM_DEBUG,
               "timer 2 is greater\n"));
  else
    ACE_DEBUG ((LM_DEBUG,
               "timers are equal\n"));
  return 0;
}
```

The code shown above is portable to all OS platforms. Note how the use of C++ features like wrapper classes and operator overloading simplifies the use of time-related operations.

3.1.4 The ACE_Timer_Queue Class

The ACE_Reactor's timer-based mechanisms are useful for applications that require timer support. For example, Web servers require watch-dog timers that release resources if clients do not send an HTTP request within a specific time interval after they connect. Likewise, daemon configuration frameworks like the Windows NT Service Control Manager [4] require services under their control to periodically report their current status via "heartbeat" messages, which can be used to restart services that have terminated abnormally.

The ACE_Timer_Queue class provides mechanisms that allow applications to register time-based concrete event handlers that derive from ACE_Event_Handler. The ACE_Timer_Queue ensures that the handle_timeout method in these event handlers is invoked at an application-specified time in the future. The methods of the ACE_Timer_Queue class illustrated in Figure 5 enable applications to schedule, cancel, and invoke the timer objects.

```
class ACE_Timer_Queue
{
public:
  // True if queue is empty, else false.
  int is_empty (void) const;

  // Returns earliest time in queue.
  const ACE_Time_Value &earliest_time (void) const;

  // Schedule a <handler> to be dispatched at
  // the <future_time> and at subsequent
  // <interval>s.  Returns a timer id that can
  // be used to cancel the timer.
  virtual long schedule
    (ACE_Event_Handler *handler,
     const void *act,
     const ACE_Time_Value &future_time,
     const ACE_Time_Value &interval);

  // Cancel all registered <ACE_Event_Handlers>
  // that match the address of <handler>, which
  // can be registered multiple times.
  virtual int cancel (ACE_Event_Handler *handler);

  // Cancel the single <ACE_Event_Handler>
  // matching the <timer_id> value returned
  // from <schedule>.
  virtual int cancel (int timer_id,
                      const void **act = 0);

  // Expire all timers <= <expire_time>.  This
  // method must be called manually since  it
  // is not invoked asynchronously.
  virtual void expire
    (const ACE_Time_Value &expire_time);
private:
  // ...
};
```

Figure 5: Interface ACE_Timer_Queue

An application schedules a concrete event handler to expire after delay amount of time. If it expires, the act is passed as the value to the event handler's handle_timeout hook method. The interval value is used to reschedule the event handler automatically if it does not equal ACE_Time_Value::zero.

The schedule method returns a timer id that uniquely identifies each event handler's registration in a timer queue implementation. The timer id is used by the cancel method to remove an event handler before it expires. If a non-NULL act is passed to cancel it is set to the *asynchronous completion token* (ACT) [9] passed in by the application when the timer was originally scheduled. This makes it possible to delete dynamically allocated ACTs to avoid memory leaks.

By default, the ACE_Timer_Queue used by the ACE_Reactor is implemented as a heap. A heap is a

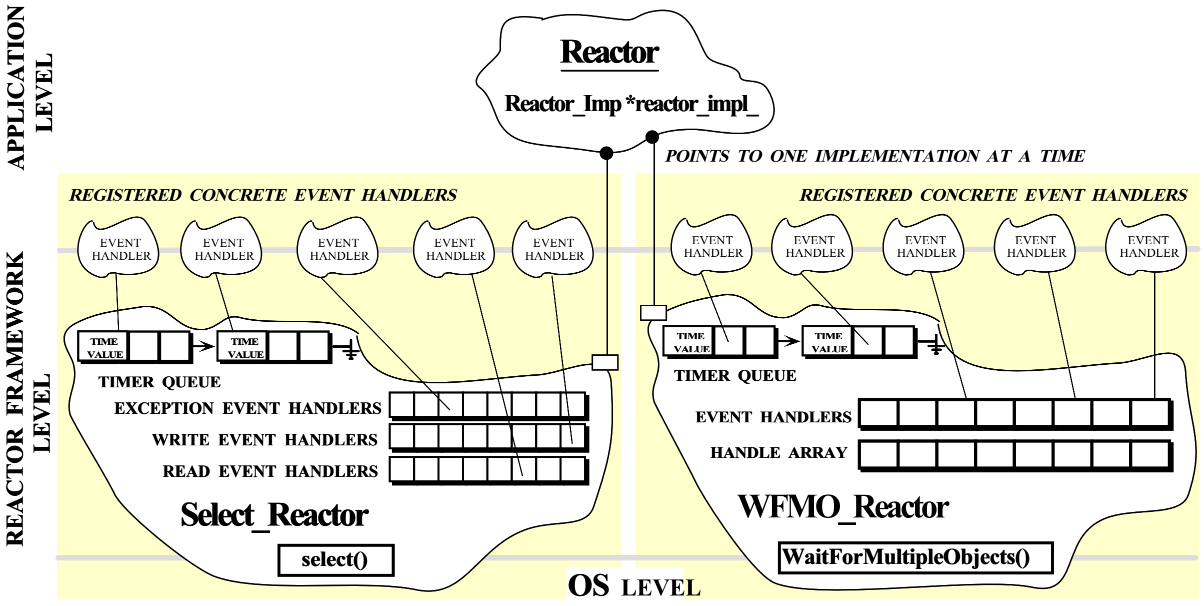


Figure 6: Using the Bridge Pattern for the Reactor Implementations

"partially-ordered, almost-complete binary tree" that ensures the average- and worst-case time complexity for inserting or deleting a concrete event handler is $O(\lg n)$. Implementing timers with heaps is particularly useful for real-time applications [10] and middleware [11] that require predictable and low-latency timer operations.

The ACE_Timer_Queue heap consists of tuples that contain an ACE_Time_Value, ACE_Event_Handler *, and void *. The ACE_Event_Handler * field points to the concrete event handler scheduled to be run at the time specified by the ACE_Time_Value field. The void * field is the ACT argument supplied when a concrete event handler is originally scheduled. When a timer expires, this ACT is automatically passed to the handle_timeout method described in Section 3.1.2. Each ACE_Time_Value in the heap is stored in *absolute* time units, *e.g.*, as generated by the ACE_OS gettimeofday API.

Virtual methods are used in the ACE_Timer_Queue interface. Thus, applications can extend the default ACE implementation to support alternative data structures, such as delta-lists [12] or timing wheels [13]. Delta-lists store time in "relative" units represented as offsets or "deltas" from the earliest ACE_Time_Value at the front of the list. Timing wheels use a circular buffer that makes it possible to start, stop, and maintain timers within the range of the wheel in $O(1)$ time. The ACE framework provides a several alternative timer queue implementations.

3.2 Platform-Dependent Class Components

The ACE_Reactor class is the public interface that applications use to access the ACE Reactor framework. The ACE_Reactor interface consists of virtual methods. Therefore, it can be extended via inheritance. The most common way to extend ACE_Reactor is not to subclass it, however. Instead, the Bridge pattern [6] is used to decouple the ACE_Reactor interface from its ACE_Reactor_Impl subclass implementations, as shown in Figure 6.

The implementation of ACE_Reactor_Impl subclasses differ across OS platforms. However, the method names and overall functionality provided by the ACE_Reactor interface remains the same. This uniformity stems from the modularity of the ACE_Reactor's design, which enhances its reuse, portability, and maintainability.

Two of the ACE_Reactor_Impl subclasses provided by the ACE framework include ACE_Select_Reactor and ACE_WFMO_Reactor, which encapsulate the select and WaitForMultipleObjects OS event demultiplexing calls, respectively. The WaitForMultipleObjects and select versions of the ACE_Reactor implementation are outlined below.

3.2.1 The ACE_Select_Reactor Class

The select-based ACE_Reactor contains three arrays of ACE_Event_Handler *, as shown in Figure 6 (1). These arrays store pointers to registered concrete event handlers that process various types of events specified by applications.

The ACE_Handle_Set class provides an efficient C++ wrapper for the underlying fd_set bitmask data type. An fd_set maps the I/O handle name-space onto a compact bit-vector representation and provides operations for enabling, disabling, and testing bits corresponding to I/O handles. The fd_sets are passed to the select call when an application calls the ACE_Reactor handle_events method.

The ACE_Handle_Set class optimizes several common fd_set operations by (1) using "full-word" comparisons to minimize unnecessary bit manipulations, (2) caching certain values to avoid recalculating bit-offsets on each call, and (3) using an exclusive-or algorithm that is linear in the number of active handles in an `fd_set`, rather than the number of *potentially* active handles, which can substantially reduce run-time overhead.

3.2.2 The ACE_WFMO_Reactor Class

The `WaitForMultipleObjects` interface is more general than `select`, allowing applications to wait for a wider-range of events, such as synchronization events. Therefore, the `WaitForMultipleObjects`-based `ACE_Reactor` requires neither the three `ACE_Event_Handler *` arrays nor the `ACE_Handle_Set` class. Instead, a single array of `ACE_Event_Handler` pointers and an array of handles are allocated and used internally to store the registered concrete event handlers.

4 Distributed Logging Service Example

The Reactor framework is intended to simplify the development of event-driven applications, such as Web servers [14, 15] and CORBA Object Request Brokers [11]. This section describes the design and implementation of a distributed logging service that illustrates how to use the `ACE_Reactor`.

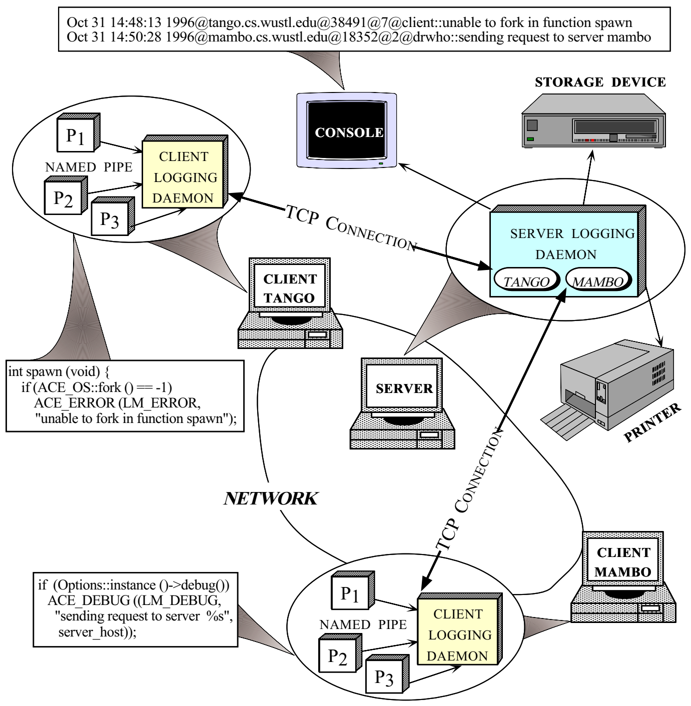
4.1 Overview

A distributed logging service keeps records diagnostic information sent from one or more applications in a central location. The logging service described below allows clients to send log records to a central logging server, as shown in Figure 7. The logging service combines the event demultiplexing and dispatching features of the `ACE_Reactor` together with the C++ socket wrappers described in [16].

The key components in the logging service are described below:

Application logging interface: Application processes, *e.g.*, P_1, P_2, P_3, running on client hosts use the `ACE_Log_Msg` C++ class to generate various types of logging records, such as LM_ERROR and LM_DEBUG. The `ACE_Log_Msg::log` method provides a `printf`-style interface. Figure 8 outlines several of the priority levels and data format for records that are exchanged between the client application, the client logging daemon, and the server logging daemon. When invoked by an application, the logging interface formats and timestamps the logging records and writes them to a well-known STREAM pipe [17]. A *client logging daemon* is responsible for processing these records, as described next.

Client logging daemon: The client logging daemon runs on every host machine participating in the distributed logging service. Each client logging daemon receives logging



```
Oct 31 14:48:13 1996@tango.cs.wustl.edu@38491@7@client::unable to fork in function spawn
Oct 31 14:50:28 1996@mambo.cs.wustl.edu@18352@2@drwho::sending request to server mambo
```

```
int spawn (void) {
   if (ACE_OS::fork () == -1)
      ACE_ERROR (LM_ERROR,
      "unable to fork in function spawn");
```

```
if (Options::instance ()->debug())
   ACE_DEBUG ((LM_DEBUG,
      "sending request to server %s",
      server_host));
```

Figure 7: Components in the Distributed Logging Service

```
enum Log_Priority
{
  // = TITLE
  //   These enumerals indicate the relative
  //   priorities of the logging messages.

  // Messages that contain information normally
  // used only when debugging a program.
  LM_DEBUG,

  // Critical conditions, e.g., hard device errors
  LM_ERROR,

  // ...
};

struct Log_Record
{
  enum {
    // Maximum number of bytes in logging record.
    MAXLOGMSGLEN = 1024
  };

  // Type of logging record.
  Log_Priority type_;
  // length of the logging record.
  long length_;
  // Time logging record generated.
  long time_stamp_;
  // Id of process that generated the record.
  long pid_;
  // Logging record data.
  char rec_data_[MAXLOGMSGLEN];
};
```

Figure 8: Logging Record Format

records from applications on this machine using some form of localhost-only IPC, such as STREAM pipes or UNIX domain sockets.

The client logging daemon continuously receives the logging records from application processes. It then converts the multi-byte record header fields into network-byte order. Finally, it forwards the records to the *server logging daemon* using TCP. The server typically runs on a remote host.

Server logging daemon: The server logging daemon continuously collects, reformats, and outputs the incoming logging records it receives from its client logging daemons. The logging information displayed by the server indicates (1) the time the logging record was generated by the client application's logging interface, (2) the host machine the application was running on, (3) the process identifier of the application, (4) the priority level of the logging record, (5) the command-line name (*i.e.*, "`argv[0]`") of the application, and (6) a string that contains the text of the logging message.

The remainder of this section focuses on the server logging daemon. Various `ACE_Reactor` and ACE C++ socket wrapper mechanisms are illustrated and described in this example.

4.2 The Server Logging Daemon

The interface and implementation of the classes used to construct the server logging daemon are described below. The logging server runs in a single process, handling logging records from clients. Event demultiplexing is provided by the `ACE_Reactor`, which dispatches incoming logging records received from clients in a round-robin fashion.

Each time the application invokes the `handle_events` method on the `ACE_Reactor`, one logging record is read from each client whose associated I/O handle became active. Logging records are written to the standard output of the server logging daemon. This output can be redirected to various devices such as printers, persistent storage repositories, or network management consoles.

Several C++ class components appear in the logging service architecture. The inheritance and template parameterization relationships between the various components are illustrated in Figure 9 using Booch notation [18]. To enhance reuse and extensibility, the components in this figure are designed to decouple the following aspects of the server logging daemon architecture, which are described from the bottom to the top of Figure 9:

Reactor framework components: The components in the Reactor framework encapsulate the lowest-level mechanisms for performing event demultiplexing and dispatching of concrete event handler hook methods. These components are discussed in Section 3.

Connection-related components: These generic templates implement the Acceptor-Connector pattern [19], which provides a reusable connection factory. The `ACE_Acceptor` implements the Acceptor component in this pattern – it accepts network connections from remote clients and creates `ACE_Svc_Handler`s that process data exchanged with the connected client. These components are discussed in Section 4.2.1.

Application-specific components: These components implement the application-specific portion of the distributed logging service. The `Logging_Acceptor` class supplies concrete parameterized types to the `ACE_Acceptor`, which creates a connection handling instantiation that is specific for the logging application. Likewise, the `Logging_Handler` class is instantiated with concrete types that provide the application-specific functionality necessary to receive and process logging records from remote clients. These components are discussed in Section 4.2.2.

Using this highly-decoupled OO decomposition enhances the design and extensibility of the server logging daemon. Each component is described below.

4.2.1 Connection-related Components

The following classes are used to implement the Acceptor component in the Acceptor-Connector pattern [19]. The Acceptor component decouples the (1) *passive* connection establishment and service initialization from (2) the processing performed by the two endpoints of a service once they are connected and initialized.

The ACE_Acceptor class: This class provides a generic template for a family of classes that standardize and automate the steps necessary to accept network connection requests from clients. Figure 10 illustrates the interface for the `ACE_Acceptor` class.

The `ACE_Acceptor` template class inherits from `ACE_Event_Handler`, which allows its `handle_input` method to be dispatched with the Reactor framework. In addition, this template class is parameterized by a concrete `SVC_HANDLER`, which understands how to perform I/O with clients, and a `PEER_ACCEPTOR` class, which understands how to accept client connections.

Classes instantiated from `ACE_Acceptor` are capable of the following behavior:

1. Accepting connection requests sent from remote clients;
2. Dynamically allocating an object of the `SVC_HANDLER` subclass;
3. Registering this object with an instance of the `ACE_Reactor`. In turn, the `SVC_HANDLER` class must know how to process data exchanged with the client.

The `ACE_Acceptor` class implementation is shown in Figure 11.[3] When one or more connection requests arrive, the `handle_input` method is dispatched automatically by the `Reactor`. This method behaves as follows. First, it dynamically creates a new `SVC_HANDLER` object,

[3]This is a simplified version of the `ACE_Acceptor`. For a complete implementation see [19].

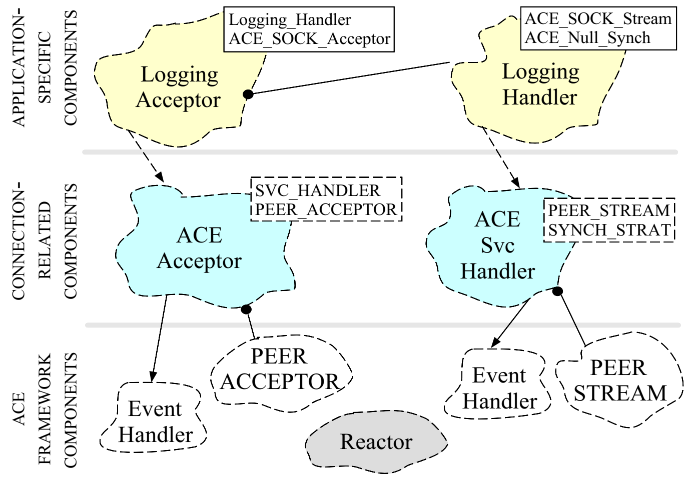


Figure 9: Components in the Server Logging Daemon

```
template <class SVC_HANDLER,
          class PEER_ACCEPTOR>
class ACE_Acceptor : public ACE_Event_Handler
{
  // = TITLE
  //   A template class that handles connection
  //   requests from a remote client.
public:
  ACE_Acceptor (ACE_Reactor *r,
                const PEER_ACCEPTOR::PEER_ADDR &a);
  ˜ACE_Acceptor (void);

protected:
  virtual ACE_HANDLE get_handle (void) const;
  virtual int handle_input (ACE_HANDLE);
  virtual int handle_close
    (ACE_HANDLE = ACE_INVALID_HANDLE,
     ACE_Reactor_Mask = ACE_Event_Handler::READ_MASK);

private:
  // Accept connections.
  PEER_ACCEPTOR acceptor_;
};
```

Figure 10: Acceptor Class Interface

```
// Shorthand names
#define SH SVC_HANDLER
#define PA PEER_ACCEPTOR

template <class SH, class PA>
ACE_Acceptor<SH, PA>::ACE_Acceptor
  (ACE_Reactor *reactor,
    const PA::PEER_ADDR &addr)
  : acceptor_ (addr)
{
  // Register to accept connections.
  reactor->register_handler
    (this,
     ACE_Event_Handler::ACCEPT_MASK);
}

template <class SH, class PA> ACE_HANDLE
ACE_Acceptor<SH, PA>::get_handle (void) const
{
  // Return the underlying I/O handle
  // when called by Reactor during
  // registration.
  return this->acceptor_.get_handle ();
}

template <class SH, class PA> int
ACE_Acceptor<SH, PA>::handle_close
  (ACE_HANDLE, ACE_Reactor_Mask)
{
  // Close down the Acceptor and
  // release the handle resources.
  return this->acceptor_.close ();
}

template <class SH, class PA>
ACE_Acceptor<SH, PA>::˜ACE_Acceptor (void)
{
  this->handle_close ();
}

// Template Method that accepts connections
// from client hosts, creates and activates
// a service handler.

template <class SH, class PA> int
ACE_Acceptor<SH, PA>::handle_input
  (ACE_HANDLE)
{
  // Create a new Svc_Handler.
  SH *svc_handler = new SH;

  // Accept connection into the handler.
  this->acceptor_.accept (svc_handler->peer ());

  // Activate the handler.
  svc_handler->open (0);
}
```

Figure 11: Acceptor Class Implementation

which is responsible for processing data sent and received from the new client. Next, it accepts an incoming connection into the SVC_HANDLER. Finally, it calls the open hook on the new SVC_HANDLER. As shown below, this hook method can register the newly created SVC_HANDLER with the ACE_Reactor.

The ACE_Svc_Handler class: This parameterized type provides a generic template for processing data exchanged with clients. In the distributed logging service, for example, the I/O format involves logging records. However, different formats can be substituted easily for other applications. The interface of the ACE_Svc_Handler class is depicted in Figure 12. As with the ACE_Acceptor class, this

```
// Receive client message from the remote clients.

template <class PEER_STREAM>
class ACE_Svc_Handler : public ACE_Event_Handler
{
public:
  ACE_Svc_Handler (void);

  // Must be filled in by subclass
  virtual int open (void *) = 0;

  PEER_STREAM &peer (void);

 // Demultiplexing hooks.
  virtual ACE_HANDLE get_handle (void) const;

protected:
  // Connection open to the client.
  PEER_STREAM peer_stream_;
};
```

Figure 12: Svc_Handler Class Interface

class inherits functionality from the ACE_Event_Handler base class. This allows concrete event handlers instantiated from ACE_Svc_Handler to be created dynamically and registered with the ACE_Reactor. The handle_input method in the ACE_Acceptor class automatically performs this behavior.

Figure 13 illustrates the ACE_Svc_Handler class implementation. Note how the combination of inheritance, dynamic binding, and parameterized types decouples (1) the general-purpose portions of the framework, *e.g.*, connection establishment from (2) the application-specific functionality, *e.g.*, receiving logging records.

```
#define PS PEER_STREAM

// Extract the underlying PS (e.g., for
// use by accept()).

template <class PS> PS &
ACE_Svc_Handler<PS>::peer (void)
{
  return this->peer_stream_;
}

template <class PS>  ACE_HANDLE
ACE_Svc_Handler<PS>::get_handle (void) const
{
  // Return the underlying I/O handle
  // when called by Reactor during
  // registration.
  return this->peer_stream_.get_handle ();
}
```

Figure 13: Svc_Handler Class Implementation

Concrete event handlers are typically closed down when a client process exits or when a serious transmission error occurs. When the ACE_Reactor is instructed to remove an ACE_Svc_Handler from its internal tables, *e.g.*, when the service handler's handle_input method return -1 upon receiving EOF from a peer service handler, it automatically invokes the service handler's handle_close method. By default, this method deallocates the service handler's memory, which was originally allocated by the handle_input method in the ACE_Acceptor class.

4.2.2 Application-specific Services

The following classes implement the application-specific portion of a service, which is the logging server daemon in this example.

The Logging_Acceptor class: To implement the server daemon portion of the distributed logging application the Logging_Acceptor class is instantiated from the generic ACE_Acceptor template, as follows:

```
typedef ACE_Acceptor <Logging_Handler,
                      ACE_SOCK_Acceptor>
        Logging_Acceptor;
```

The SVC_HANDLER template parameter is instantiated with the Logging_Handler class described below. Likewise, the PEER_ACCEPTOR template parameter is replaced by the ACE_SOCK_Acceptor class. The ACE_SOCK_* instantiated types are part of a C++ socket wrapper [16] that encapsulates the socket interface and allows data to be transmitted reliably between peer processes.

By using parameterized types, the classes that perform IPC can be any network programming interface that conforms to the API used by the ACE_Acceptor. For example, depending on certain properties of the underlying OS platform, such as whether it is a BSD or System V variant of UNIX, the logging application may instantiate the ACE_Svc_Handler class to use either SOCK_SAP or TLI_SAP, which is the ACE C++ wrapper for the System V Transport Layer Interface (TLI), as illustrated below:

```
// Logging application.

#if defined (USE_SOCKETS)
typedef ACE_SOCK_Stream PEER_STREAM;
#elif defined (USE_TLI)
typedef ACE_TLI_Stream PEER_STREAM;
#endif /* USE_SOCKETS */

class Logging_Handler
  : public ACE_Svc_Handler<PEER_STREAM>
{
  // ...
};
```

The flexibility offered by template-based extensibility is useful when developing applications that must run portably across multiple OS platforms.

The Logging_Handler class: This class is created by instantiating the `ACE_Svc_Handler` class as follows:

```
class Logging_Handler :
  public ACE_Svc_Handler<ACE_SOCK_Stream>
{
public:
  // Initialization hook called by
  // the <ACE_Acceptor>.
  virtual int open (void *) {
    ACE_SOCK_Stream::PEER_ADDR addr;

    // Cache remove host name.
    peer ().get_remote_addr (addr);
    ACE_OS::strcpy (host_name_,
                    addr.get_host_name ());

    // Register ourselves with the Reactor so it
    // will dispatch us automatically when input
    // data arrive from clients.
    ACE_Reactor::instance ()->register_handler
      (this, ACE_Event_Handler::READ_MASK);
  }

  // Demultiplexing hook called by
  // the <ACE_Reactor>.
  virtual int handle_input (ACE_HANDLE);

private:
  char host_name_[MAXHOSTNAME];
};
```

The `open` hook caches the host address of the associated client when an object of this class is dynamically allocated. As illustrated by the "console" window in Figure 7, the name of this host is printed along with the logging records received from a client logging daemon.

The `PEER_STREAM` parameter is instantiated with the `ACE_SOCK_Stream` class. The `handle_input` method is called automatically by the `ACE_Reactor` when input arrives on the underlying `ACE_SOCK_Stream`. This method can be implemented as follows:

```
// Hook method for handling the reception of
// remote logging transmissions from clients.

int
Logging_Handler::handle_input (ACE_HANDLE)
{
  // Retrieve exactly 4 bytes for the length.
  ACE_INT32 n =
    peer_stream_.recv_n (&len, sizeof len);

  if (n != sizeof len)
    // Trigger handle_close().
    return -1;
  else {
    ACE_Log_Record lr;

    size_t len = ntohl (len);
    n = this->peer_stream_.recv_n (&lr, len));

    if (n != len)
      ACE_ERROR_RETURN ((LM_ERROR,
                         "%p at host %s\n",
                         "client logger",
                         this->host_name_),
                        -1);
    lr.decode ();

    if (lr.len == n)
      lr.print (this->host_name_, 0, stderr);
    else
      ACE_ERROR_RETURN ((LM_DEBUG,
                         "lr.len = %d, n = %d\n",
                         lr.len,
                         n),
                         -1);
    return 0; // Keep handler registered.
  }
}
```

This method performs two `recv`s to simulate a message-oriented service via the underlying TCP connection. This behavior is necessary since TCP is bytestream protocol, rather than record protocol.[4] The first `recv` reads the length of the following logging record, which is stored as a fixed-size integer. The second `recv` then reads this many bytes to obtain the actual record. Naturally, the client sending this message must follow the same message framing protocol.

4.2.3 The main() Driver Program

The following event-loop drives the `ACE_Reactor`-based logging server:

```
int
main (int argc, char *argv[])
{
  // 1. Set the program name with the logger.
  ACE_LOG_MSG->open (argv[0]);

  // Ensure correct usage.
  if (argc != 2)
    ACE_ERROR_RETURN ((LM_ERROR,
                       "usage: %n port-number"),
                      -1);

  // 2. Create an addr and an acceptor.
  ACE_INET_Addr port (ACE_OS::atoi (argv[1]));
  Logging_Acceptor acceptor
    (ACE_Reactor::instance (), port);

  // 3. Loop forever, handling client requests.
  for (;;)
    ACE_Reactor::instance ()->handle_events ();

  /* NOTREACHED */
  return 0;
}
```

In Step 1, an `ACE_Log_Msg` is created to direct any logging records generated by the server to its own standard error stream. The example code in Figures 11 and 13 illustrates how the server uses the application logging interface to log its own diagnostic messages locally. Since this local configuration does not use the server logging daemon there is no danger of causing an infinitely recursive logging loop.

In step 2, the server creates a `Logging_Acceptor`, whose constructor registers itself with the `ACE_Reactor` singleton. In step 3, the server enters an endless loop that blocks in the `handle_events` method until events are received from client logging daemons.

Figure 14 illustrates the state of the logging server daemon after two clients have been dispatched by the `ACE_Reactor` and become participants in the distributed logging service. As shown in the figure, a `Logging_Handler` has been dynamically instantiated and registered for each client.

[4]Note that this implementation is not entirely robust in its handling of "short reads."

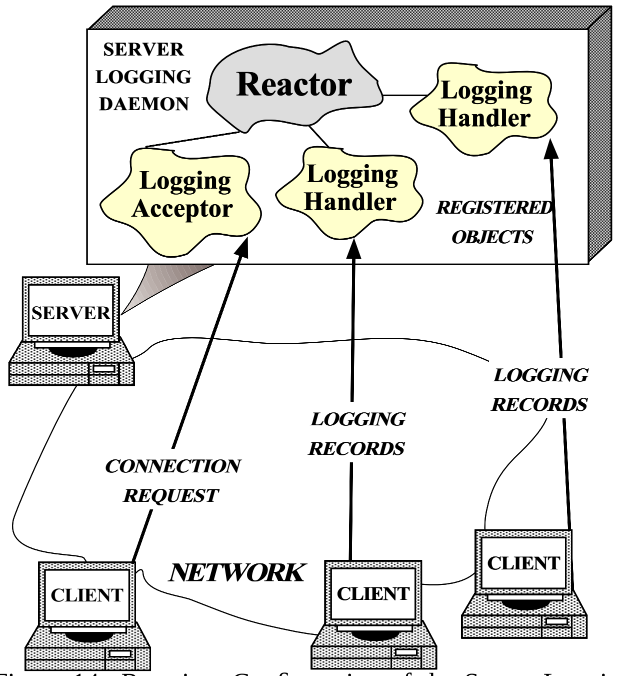


Figure 14: Run-time Configuration of the Server Logging Daemon

When events arrive at the server, the `ACE_Reactor` automatically dispatches the `handle_input` method of the `Logging_Acceptor` and `Logging_Handler`. For instance, when connection requests arrive from client logging daemons, the `ACE_Reactor` invokes the `handle_input` method of the `Logging_Acceptor`. This method accepts the new connection and creates a `Logging_Handler` that reads all data sent by the client and displays it on the standard output stream. Likewise, when logging records or shutdown messages arrive from connected client logging daemons, the `ACE_Reactor` invokes the `handle_input` method of the corresponding `Logging_Handler`.

Figure 7 portrays the entire system during execution. Logging records are generated from the client application's *logging interface* via the `ACE_Log_Msg::log` method. This method forwards the logging records to the *client logging daemon* running on the same host as the application. The client logging daemon then transmits the record across the network to the *server logging daemon*, where it is displayed on the server's logging console.

4.3 Evaluating Alternative Logger Implementations

The distributed logging service described in Section 4.2 was originally programmed in C and used in a commercial on-line transaction processing product. This section compares the C++ and C versions of the distributed logging service in terms of several software quality factors such as modularity, extensibility, reusability, and portability.

4.3.1 The non-OO C-based Logging Service

The original logging service was developed for a BSD UNIX-based commercial on-line transaction processing product. It was written in C and used BSD sockets and `select` directly. Later, it was ported to other operating systems, such as System V UNIX and Win32.

The original C implementation was hard to modify, extend, and port, due to the following problems:

Tightly-coupled functionality: In the original C logging service, the event demultiplexing, service dispatching, and event processing operations were tightly-coupled with the code that accepted client connection requests and received client logging records.

Excessive use of global variables: Several global data structures were used to maintain the relationship between (1) per-client context information, such as the client hostname and current processing status, and (2) I/O handles that identify the appropriate context record. Therefore, any enhancements or modifications to the program directly affected the existing source code.

4.3.2 The OO C++-based Logging Service

The OO `ACE_Reactor`-based version of the logging service described in this paper uses data abstraction, inheritance, dynamic binding, and templates to provide the following benefits:

Avoid global variables: The `ACE_Reactor`-based logging service contains no global variables. Instead, the `ACE_Reactor` singleton was used to register each `Logging_Handler`, which encapsulates the client address and the underlying I/O handle used to communicate with clients.

Decouple policies and mechanisms: Application *policies* that process incoming connections and data are decoupled from the lower-level *mechanisms* that perform demultiplexing and dispatching. Decoupling policies and mechanisms has the following advantages:

• **Reusability:** The ACE Reactor framework provides reusable components that perform all the lower-level event demultiplexing and service dispatching. Thus, only a small amount of application-specific code is required to implement the server logging daemon, as shown in Section 4. This code is primarily concerned with application processing activities, such as accepting new connections and receiving client logging records.

• **Extensibility:** The separation of policies and mechanisms in the `ACE_Reactor`'s architecture enhances extensibility both above and below its public interface. For example, it is straightforward to extend the server logging daemon's functionality, *e.g.,* to add an "authenticated logging" feature. Such extensions simply inherit from the

ACE_Event_Handler base class and selectively implement the necessary virtual method(s). Likewise, by instantiating the ACE_Acceptor and ACE_Svc_Handler templates, subsequent applications may be produced without redeveloping existing infrastructure. In contrast, modifying the original non-OO C version in this manner required direct changes to the existing code.

• **Portability:** It is possible to modify the underlying event demultiplexing mechanism of the ACE_Reactor without affecting existing application code. For example, porting the ACE_Reactor-based distributed logging service from a BSD platform to a System V or win32 platform requires no visible changes to application code. In contrast, porting the original C version of the distributed logging service from select to WaitForMultipleObjects was tedious and error-prone. For example, several subtle errors were introduced into the source code that did not manifest themselves until run-time.

• **Efficiency:** In certain types of applications, such as real-time embedded control systems, data is available immediately on one or more handles. Therefore, polling these handles via non-blocking I/O may be more efficient than using OS event demultiplexers like select or WaitForMultipleObjects. Extending the ACE_Reactor to support this alternative demultiplexing implementation did not require modifications to its public interface.

One consequence of the use of OO in the Reactor framework is that it uses dynamic binding extensively. [16] discusses why avoiding dynamic binding is often advisable when designing "thin" C++ wrappers for sockets. On some compilers, the overhead resulting from indirect virtual table dispatching may be fairly high. In such cases, developers may need to refrain from using dynamic binding extensively.

In general, however, the significant increase in clarity, extensibility, and modularity provided by the ACE_Reactor framework more than compensates for the slight decrease in efficiency. Furthermore, the ACE_Reactor is typically used to develop distributed applications. The major sources of overhead in distributed systems result from activities like caching, latency, network/host interface hardware, presentation-level formatting, memory-to-memory copying, and process management [20]. Therefore, the additional indirection caused by dynamic binding is typically insignificant by comparison [21]. In addition, good C++ compilers can optimize virtual method overhead away completely via the use of "adjustor thunks" [22].

5 Design Rules for Using the Reactor Effectively

The ACE_Reactor is a powerful framework for demultiplexing events and dispatching event handlers. Like other frameworks, however, learning to use the ACE_Reactor takes time and effort. One way to shorten the learning curve is to understand the *design rules* necessary to use the ACE_Reactor effectively. The design rules described below are based on extensive experience gained by helping ACE users program the Reactor framework correctly.

5.1 Understand Concrete Event Handler Return Value Semantics

Context: The return values of the various handle_* hook methods defined by concrete event handlers cause the ACE_Reactor to behave in different ways. The intent of using return values to trigger different behaviors is to reduce the complexity of the ACE_Reactor's API. However, the return values are often a source of surprise to programmers. Therefore, it is important to understand the effects of the values returned from the handle_* methods, which fall into the following three cases:

1. Zero: When a handle_* method returns zero (0) this informs the ACE_Reactor that the event handler wishes to continue being processed as before, *i.e.*, it should remain in a table in the ACE_Reactor's implementation. Thus, the ACE_Reactor will continue to include the handle of this event handler next time it invokes its event demultiplexer via handle_events. This is the "normal" behavior of event handlers whose lifetime extends beyond a single handle_* method dispatch.

2. Greater than zero: When a handle_* method returns greater than zero (> 0) this informs the ACE_Reactor that the event handler wishes to be dispatched again *before* the ACE_Reactor blocks on its event demultiplexer. This feature is useful for cooperative event handlers to enhance overall system "fairness." In particular, it allows one event handler to allow other event handlers to be dispatched before it retains control again.

3. Less than zero: When a handle_* method returns less than zero (< 0) this informs the ACE_Reactor that the event handler wants to be closed and removed from the ACE_Reactor's internal tables. To accomplish this, the ACE_Reactor invokes the event handler's handle_close cleanup method. This method can perform user-defined termination activities, such as deleting dynamic memory allocated by the object or closing log files. When the handle_close method returns, the ACE_Reactor removes the associated concrete event handler from its internal tables.

To minimize problems with handle_* return values, observe the following design rules when implementing concrete event handlers:

Design rule 0: *Do not manually delete event handler objects or call* handle_close *explicitly* – Instead, ensure the ACE_Reactor invokes the handle_close cleanup method automatically. Thus, applications must follow the proper protocol, *i.e.*, either by (1) returning a negative value from a handle_* hook method or (2) calling remove_handler.

This design rule ensures that an ACE_Reactor can cleanup its internal tables properly. If this rule is not obeyed, the ACE_Reactor will incur unpredictable memory management problems when it later tries to remove externally deleted concrete event handlers. Subsequent design rules elaborate on how to ensure that the ACE_Reactor invokes the handle_close cleanup method.

Design rule 1: *Return expressions in* handle_* *methods of classes inheriting from* ACE_Event_Handler *should be constant*. This design rule makes it easier to statically check whether the handle_* methods are returning appropriate values. If this rule must be violated, developers must precede the return statement with a comment that explains why a variable is used rather than a constant.

Design rule 2: *Return statements in* handle_* *methods of classes inheriting from* ACE_Event_Handler *that do not return 0 must be preceded by a comment stating what the return value signifies*. This design rule ensures that all non-0 return values are explicitly intended by developers.

5.2 Understand the handle_close() Cleanup Hook Semantics

Context: The handle_close cleanup hook method must be called by the ACE_Reactor either (1) *implicitly*, *i.e.*, when a handle_* method returns a negative value like -1 or (2) *explicitly*, *i.e.*, if an application calls the remove_handler method to remove a concrete event handler. In particular, the ACE_Reactor will *not* call handle_close automatically when an I/O handle is closed, either by the local application or a remote application. Thus, applications must determine when an I/O handle has closed down and must take the appropriate steps so the ACE_Reactor will trigger the handle_close cleanup method.

Example: The following Logging_Handler code fragment from Section 4.2.2 illustrates how to trigger the cleanup hook incorrectly:

```
// Hook method for handling the reception of
// remote logging transmissions from clients.

int
Logging_Handler::handle_input (ACE_HANDLE)
{
  return peer_stream_.recv (&len, sizeof len);
}
```

Note that this method will only trigger the handle_close hook when recv fails, *i.e.*, returns -1. However, it will not work correctly when recv returns 0 or > 0. To minimize problems with handle_close cleanup methods, therefore, observe the following design rules when implementing concrete event handlers:

Design rule 3: *Return a negative value from a* handle_* *method when you want to trigger the corresponding* handle_close *cleanup method on the concrete event handler*. The following revised Logging_Handler code fragment illustrates how to trigger the cleanup hook correctly:

```
// Hook method for handling the reception of
// remote logging transmissions from clients.

int
Logging_Handler::handle_input (ACE_HANDLE)
{
  ssize_t n =
    peer_stream_.recv (&len, sizeof len);

  if (n == 0)
    // Trigger handle_close().
    return -1;
  // ...

  // Keep handler registered for ‘‘normal’’ case.
  return 0;
}
```

When the handle_input method receives a 0 from recv, it returns -1. This value triggers the ACE_Reactor to call the handle_close cleanup hook.

The value -1 is typically used to trigger the cleanup hook since it's a common error code with the ACE_OS wrappers for native OS APIs. However, any negative value from a handle_* method will trigger handle_close.

Design rule 4: *Confine all* Event_Handler *cleanup activities to the* handle_close *cleanup method*. In general, it is easier to consolidate all the cleanup activities in the handle_close method, rather than dispersing them throughout the handle_* methods or other methods in an event handler. This design rule is particularly important to follow when dealing with dynamically allocated event handlers that must be cleaned up with delete this (see Rule 9).

5.3 Remember that ACE_Time_Value Arguments are Relative

Context: It's important to remember that both ACE_Time_Value arguments passed to the schedule_timer method of the ACE_Reactor are specified *relative* to the current time.

Example: The following code schedules an object to print the name of the executable program, *i.e.*, argv[0], every interval number of seconds, starting delay seconds in the future:

```
class Hello_World : public ACE_Event_Handler
{
public:
  virtual int handle_timeout
    (const ACE_Time_Value &tv,
     const void *act)
  {
    ACE_DEBUG ((LM_DEBUG,
                "%[s] %d, %d\n",
                act,
                tv.sec (),
                tv.usec ()));
    return 0;
  }
};

int main (int argc, char *argv[])
{
  if (argc != 3)
```

```
    ACE_ERROR_RETURN ((LM_ERROR,
                       "usage: %s delay interval\n",
                       argv[0]), -1);

  Hello_World handler; // timer object.

  ACE_Time_Value delay = ACE_OS::atoi (argv[1]);
  ACE_Time_Value interval = ACE_OS::atoi (argv[2]);

  // Schedule the timer.
  ACE_Reactor::instance ()->schedule_timer
    (&handler,
     (const void *) argv[0],
     delay,
     interval);

  // Run the event loop.
  for (;;)
    ACE_Reactor::instance ()->handle_events ();

  /* NOTREACHED */
}
```

A common mistake is to pass an *absolute* time value to `schedule_timer`. For instance, consider a different example:

```
  ACE_Time_Value delay = ACE_OS::atoi (argv[1]);
  delay += ACE_OS::gettimeofday ();

  // Callback every following 10 seconds.
  ACE_Time_Value interval = delay + 10;

  ACE_Reactor::instance ()->schedule_timer
    (&handler,
     0,
     delay,
     interval);
```

However, this timer will not expire for a *long* time in the future since it adds the current time of day to the `delay` and `interval` requests by the user.

The following is a design rule to follow when implementing concrete event handlers to minimize problems with absolute `ACE_Time_Value`s:

Design rule 5: *Do not use absolute times as the third or fourth arguments to* `ACE_Reactor::schedule_timer`. In general, these arguments should be less than an extremely long delay, which is significantly less than the current time.

5.4 Track ACE_Event_Handler Lifetimes Carefully

Various problems arise from failing to track the lifetimes of `ACE_Event_Handler`s that are registered with an `ACE_Reactor`. These problems are hard to track down without the use of a memory error detection tool like Purify [23], which catches some, but not all, of the following lifetime-related problems:

5.4.1 Use Non-dynamically Allocated Event Handlers Sparingly

Context: Certain types of applications, such as embedded real-time systems, try to minimize the use of dynamic memory, *e.g.*, to make system performance more predictable. It's important to be very careful to use non-dynamically allocated concrete event handlers correctly with an `ACE_Reactor`.

Example: Consider the following concrete event handler definition:

```
class My_Event_Handler : public ACE_Event_Handler
{
public:
  My_Event_Handler (const char *str = "hello")
    : str_ (ACE_OS::strnew (str)) {}

  virtual int handle_close
    (ACE_HANDLE = ACE_INVALID_HANDLE,
     ACE_Reactor_Mask = ACE_Event_Handler::READ_MASK)
  {
    // Commit suicide.
    delete this;
  }

  ~My_Event_Handler (void) {
    delete [] this->str_;
  }

  // ...
private:
  char *str_;
};
```

This class deletes itself when it's removed from the `ACE_Reactor` via its `handle_close` cleanup method. Although this may look somewhat unconventional, it is a perfectly valid C++ idiom. However, it only works as long as the object being deleted was allocated dynamically.

In contrast, if the object being deleted was *not* allocated dynamically, the global dynamic memory heap will be corrupted. The reason is that the `delete` operator will interpret `this` as a valid address in the heap. This will cause subtle memory management problems when the `delete` operator tries to insert the non-heap memory into its internal freelist.

The following example illustrates a common use-case that can corrupt the heap:

```
int main (void)
{
  // Non-dynamically allocated.
  My_Event_Handler my_event_handler;

  ACE_Reactor::instance ()->register_handler
    (&my_event_handler,
     ACE_Event_Handler::READ_MASK);

  // ...

  // Run event-loop.
  while (/* ...event loop not finished... */)
    ACE_Reactor::instance ()->handle_events ();

  // The <handle_close> method deletes an
  // object that wasn't allocated dynamically...
  ACE_Reactor::instance ()->remove_handler
    (&my_event_handler,
    ACE_Event_Handler::READ_MASK);
  return 0;
}
```

The problem with the code above is that the `ACE_Reactor` will invoke the `handle_close` method of `My_Event_Handler` when `remove_handler` is called. Unfortunately, the `handle_close` method will `delete` `this` on the `my_event_handler` object, which was not allocated dynamically.

One way to guard against this problem is to place the destructor in the private section of the `My_Event_Handler` class, *i.e.*:

```
class My_Event_Handler : public ACE_Event_Handler
{
public:
  My_Event_Handler (const char *str);
  // ...

private:
  // Place destructor into the private section
  // to ensure dynamic allocation.
  ~My_Event_Handler (void);
  // ...
};
```

In this class, the `My_Event_Handler` destructor is placed in the `private` access control section of the class. This C++ idiom ensures that all instances of this class *must* be allocated dynamically. If an instance is accidentally defined as a `static` or `auto`, it will be flagged as an error at compile-time.

The following is a design rule to follow when implementing concrete event handlers to minimize problems with concrete event handler lifetimes:

Design rule 6: *Do not delete event handlers that were not allocated dynamically*. Any `handle_close` method that contains `delete this` and whose class does not have a `private` destructor may be in violation of this design rule. In the absence of a convention checker that can identify this case statically, the `delete this` should be preceded immediately by a comment explaining why this idiom is used.

5.4.2 Remove Concrete Event Handler Appropriately

Context: As described above, determining how to remove concrete event handlers from an `ACE_Reactor` can be tricky. The following examples illustrate some other common problems.

Example: The following program illustrates another common problem related to the lifetimes of concrete event handlers:

```
ACE_Reactor reactor;

int main (void)
{
  My_Event_Handler my_event_handler;

  ACE_Reactor::instance ()->register_handler
    (&my_event_handler,
     ACE_Event_Handler::READ_MASK);

  while (/* ...event loop not finished... */)
    ACE_Reactor::instance ()->handle_events ();

  // The destructor of the ACE_Reactor singleton
  // will be called when the process exits.  It
  // removes all registered event handlers.
  return 0;
}
```

The lifetime of `my_event_handler` is defined by the lifetime of the `main` function. In contrast, the lifetime of the `ACE_Reactor` singleton is defined by the lifetime of the process. Thus, when the process exits, the destructor for the `reactor` will be called.[5] The destructor for the `ACE_Reactor` removes all event handlers that are still registered by calling their `handle_close` method. If `my_event_handler` is still registered with the `reactor`, however, its `handle_close` method will be called *after* the object has gone out of scope and been destroyed.

The following are three more design rules to follow when implementing concrete event handlers to minimize problems with concrete event handler lifetimes:

Design rule 7: *Always allocate concrete event handlers dynamically from the heap*. This is a relatively straightforward solution to many of problems related to the lifetime of concrete event handlers. If it is not possible to follow this rule, a comment must be provided when the concrete event handler is registered with the `ACE_Reactor` explaining why dynamic allocation is not used. This comment should appear immediately before the `register_handler` statement that registers the statically allocated concrete event handler with the `ACE_Reactor`.

Design rule 8: *Remove* `ACE_Event_Handler`*s from their associated* `ACE_Reactor` *before exiting the scope where they are "live"*. This rule should be used in cases where Rule 7 is not followed.

Design rule 9: *Allow the* `delete this` *idiom in the* `handle_close` *method only*, *i.e.*, do not allow `delete this` in the other `handle_*` methods or in other methods in the event handler. This rule makes it easier to check whether there are potential problems with deleting non-dynamically allocated memory. It also ensures that the `ACE_Reactor` doesn't try to access a pointer to a deleted event handler. Naturally, components unrelated to the `ACE_Reactor` may have different rules governing self-deletion.

Design rule 10: *Only* `delete this` *when the final registered event has been removed from an* `ACE_Reactor` *for a concrete event handler*. This rule avoids "dangling pointers" that can otherwise occur by prematurely deleting a concrete event handler that registered with an `ACE_Reactor` for multiple events.

For instance, the `my_event_handler` could be registered both for READ and WRITE events, as follows:

```
ACE_Reactor::instance ()->register_handler
 (&my_event_handler,
  ACE_Event_Handler::READ_MASK
  | ACE_Event_Handler::WRITE_MASK);
```

In this case, when the `handle_input` method returns -1 the `ACE_Reactor` will invoke the `handle_close` cleanup hook method. This method must *not* `delete`

[5]This is ensured by the `ACE_Object_Manager`, which destroys all singletons in ACE before a process exits.

`this` until the `WRITE_MASK` is also removed for that concrete event handler, *e.g.*, by having it return a negative value or explicitly removing it via

```
ACE_Reactor::instance ()->remove_handler
 (&my_event_handler,
  ACE_Event_Handler::WRITE_MASK);
```

The following method illustrates one way to keep track of this information:

```
class My_Event_Handler : public ACE_Event_Handler
{
public:
  My_Event_Handler (void)
  {
    // Keep track of which bits are enabled.
    ACE_SET_BITS (this->mask_,
                  ACE_Event_Handler::READ_MASK
                  | ACE_Event_Handler::WRITE_MASK);

    // Register ourselves with the Reactor for
    // both READ and WRITE events.
    ACE_Reactor::instance ()->register_handler
      (this, this->mask_);
  }

  virtual int handle_close (ACE_HANDLE h,
                            ACE_Reactor_Mask mask)
  {
    if (mask == ACE_Event_Handler::READ_MASK) {
      ACE_CLR_BITS (this->mask_,
                    ACE_Event_Handler::READ_MASK);
      // Perform READ_MASK cleanup logic.
    }
    else if (mask == ACE_Event_Handler::WRITE_MASK) {
      ACE_CLR_BITS (this->mask_,
                    ACE_Event_Handler::WRITE_MASK);
      // Perform WRITE_MASK cleanup logic.
    }

    // Only delete ourselves if we've been closed
    // down for both READ and WRITE events.
    if (this->mask_ == 0)
      delete this;
  }

  // ... handle_input() and handle_output() methods.

private:
  ACE_Reactor_Mask mask_;
  // Keep track of when to delete this.
```

The solution above maintains an `ACE_Reactor_Mask` that keeps track of when all events a concrete event handler is registered for have been removed from an `ACE_Reactor`.

5.5 Beware of WRITE_MASK Semantics

Context: The `WRITE_MASK` can be used to instruct the `ACE_Reactor` to callback to an `event_handler` whenever an application can `write` to an I/O handle without blocking. The following code illustrates how to register an event handler using the `WRITE_MASK`:

```
ACE_Reactor::instance ()->mask_ops
  (event_handler,
   ACE_Event_Handler::WRITE_MASK,
   ACE_Reactor::ADD_MASK);
```

It is always ok to write to a handle *unless* the connection is flow controlled. Thus, this `ACE_Reactor` will keep calling back the `handle_output` method of the `event_handler` until (1) the connection flow controls or (2) the `mask_ops` method is instructed to clear the `WRITE_MASK`.

Example: A common programming mistake is to forget to clear the `WRITE_MASK` once there's no longer any more data to write to the connection. This omission will cause the `ACE_Reactor` to continuously invoke the `handle_output` method of the concrete event handler that's registered with the `WRITE_MASK`. Therefore, the following design rule should be followed to avoid this problem:

Design rule 11: *Clear the* `WRITE_MASK` *when you no longer want the concrete event handler's* `handle_output` *method to get called back.*

The following code illustrates how to ensure the `handle_output` method is no longer called back:

```
ACE_Reactor::instance ()->mask_ops
  (event_handler,
   ACE_Event_Handler::WRITE_MASK,
   ACE_Reactor::CLR_MASK);
```

The `ACE_Reactor` defines a short-hand method for accomplishing the same thing:

```
ACE_Reactor::instance ()->cancel_wakeup
  (event_handler,
   ACE_Event_Handler::WRITE_MASK);
```

These methods are typically called when there are no more output messages pending on a concrete event handler.

To facilitate automated checking of this rule, programmers must insert comments into their `handle_output` methods. These comments will indicate which return paths are *not* intended to clear the `WRITE_MASK`, *i.e.*, the event handler wants to continue to be called back when it's "ok to write." Likewise, programmers should also comment the path(s) where the `WRITE_MASK` *is* removed. If there are no paths in the `handle_output` method that clear the `WRITE_MASK`, this indicates a potential violation of this design rules.

For example, the following `handle_output` method illustrates a potential application of this design rule:

```
int
My_Event_Handler::handle_output (ACE_HANDLE)
{
  if (/* output queue is now empty */) {
    /* Removing WRITE_MASK */
    ACE_Reactor::instance ()->cancel_wakeup
      (event_handler,
       ACE_Event_Handler::WRITE_MASK);
    return 0;
  } else {
    // ... continue to transmit messages
    // from the output queue.
    /* Not removing WRITE_MASK */
    return 0;
  }
}
```

If there were no comments indicating the `WRITE_MASK` is removed this would be a violation of the design rule.

5.6 Register Concrete Event Handlers Appropriately

Context: There are two ways to register a concrete event handler with an `ACE_Reactor` for I/O operations:

- **Explicitly pass the handle:** This approach uses the following `ACE_Reactor` method:

```
int register_handler
  (ACE_HANDLE io_handle,
   ACE_Event_Handler *event_handler,
   ACE_Reactor_Mask mask);
```

and passes the `ACE_HANDLE` of the I/O device explicitly, *i.e.*:

```
void register_socket (ACE_HANDLE socket,
                      ACE_Event_Handler *handler)
{
  ACE_Reactor::instance ()->register_handler
  (socket,
   handler,
   ACE_Event_Handler::READ_MASK);
  // ...
}
```

Note that this `register_handler` method allows the same concrete event handler to be registered with multiple `ACE_HANDLE`s. This feature of the Reactor framework makes it possible to minimize the amount of state required to handle many clients that are connected simultaneously to the same event handler.

- **Implicitly pass the handle:** This approach uses the other `register_handler` method on the `ACE_Reactor`:

```
int register_handler
  (ACE_Event_Handler *event_handler,
   ACE_Reactor_Mask mask);
```

In this case, a "double-dispatch" [6] is performed by the `ACE_Reactor` to obtain the underlying `ACE_HANDLE` from the concrete event handler via its `get_handle` method. This method is defined with the following signature in the `ACE_Event_Handler` base class:

```
virtual ACE_HANDLE get_handle (void) const;
```

Example: When using implicit registration, a common mistake is to omit the `const` on the `get_handle` method when deriving from `ACE_Event_Handler`. Omitting the `const` prevents the compiler from properly overriding the `get_handle` method in the subclass. Instead, it will *hide* the method in the subclass, thereby generating code that will call the *base class* `get_handle` method, which returns -1 by default. Therefore, it is important to obey the following design rule:

Design rule 12: *Make sure the signature of the* `get_handle` *method is consistent with the one in the* `ACE_Event_Handler` *base class.* If you don't follow this rule, and you "implicitly" pass the `ACE_HANDLE` to the `ACE_Reactor`, the default `get_handle` in the `ACE_Event_Handler` base class will return -1, which is erroneous.

5.7 Remove Closed Handles/Handlers from the Reactor by Centralizing Cleanup Tasks

Context: When a connection is closed down, the handle is no longer valid for I/O. In this case, `select` will continue to report that the handle is "ready for reading" so that the handle can be closed, which is typically done by calling `ACE_OS::close` or `ACE_OS::closesocket` in a concrete event handler's `handle_close` cleanup method.

Example: A common mistake when writing applications with the Reactor is to fail to remove defunct handles and their associated event handlers from the `ACE_Reactor`. If you fail to do this, however, the `ACE_Reactor` will continually callback the `handle_input` method on the event handler until the handle and its handler are removed from the `ACE_Reactor`. The following design rule helps to avoid this problem:

Design rule 13: *Return a negative value from the* `handle_*` *methods when a connection closes down (or when an error occurs on the connection) and centralize cleanup activities in the* `handle_close` *method.*

Code that follows this design rule is usually structured as follows:

```
int handle_input (ACE_HANDLE handle)
{
  // ...
  ssize_t result =
    ACE_OS::read (handle, buf, bufsize);

  if (result <= 0)
    // Connection has closed down or an
    // error has occurred.
    return -1;
  else
    // ...
```

When the -1 is returned, the `ACE_Reactor` will call your `handle_close` cleanup method. To avoid resource leaks, make sure this method gives the event handler a chance to delete itself and close its handle, *e.g.*, `ACE_OS::close (handle)`. Once `handle_close` returns, the `ACE_Reactor` will automatically remove the `handle`/handler tuple from its internal table if the handle is no longer registered for any types of events.

5.8 Use the DONT_CALL Flag to Avoid Recursive handle_close() Callbacks

Context: Earlier rules covered how an event handler's `handle_close` hook is automatically invoked by an `ACE_Reactor` when the handler is removed either *explicitly*, *e.g.*, when the `remove_handler` method is called, or *implicitly*, *e.g.*, by returning a negative value from a `handle_*` method. Applications must be careful, however, if they call `remove_handler` within the `handle_close` cleanup method since this can trigger infinite recursion.

Example: The following `handle_close` method will infinitely recurse since the `remove_handler` method will reinvoke `handle_close` again:

```
int
My_Event_Handler::handle_close
  (ACE_HANDLE,
   ACE_Reactor_Mask)
{
  // ...

  ACE_Reactor::instance ()->remove_handler
    (this->get_handle (),
     // Remove all the events for which we're
     // registered.
     ACE_Event_Handler::RWE_MASK);
  // ...
}
```

The following design rule prevents infinite recursion from occurring.

Design rule 14: *Always pass the DONT_CALL flag to* `remove_handler` *when calling it in a* `handle_close` *method.* This rule ensures that the `ACE_Reactor` will not invoke the `handle_close` method recursively. The following code illustrates how to do this:

```
int
My_Event_Handler::handle_close
  (ACE_HANDLE,
   ACE_Reactor_Mask)
{
  // ...

  ACE_Reactor::instance ()->remove_handler
    (this->get_handle (),
     // Remove all the events for which we're
     // registered.  We must pass the DONT_CALL
     // flag here to avoid infinite recursion.
     ACE_Event_Handler::RWE_MASK |
     ACE_Event_Handler::DONT_CALL);
  // ...
}
```

Incidentally, `remove_handler` is typically called from within `handle_close` in situations where (1) a concrete event handler is registered for multiple events and (2) the first time `handle_close` is called should trigger a complete shutdown of the event handler. Thus, it's essential that `handle_close` also remove the other events this event handler is registered for with the `ACE_Reactor`.

5.9 Don't Overload the ACE_Event_Handler handle_*() methods

Context: It is generally not a good idea for subclasses to overload virtual methods inherited from a base class. In C++, this overloading "hides" the inherited function in subclass, *i.e.*,

```
class My_Event_Handler :
  public ACE_Event_Handler
{
  // Overload the base classes' method:
  // virtual int handle_input (ACE_HANDLE)
  virtual int handle_input (void);
}
```

In this example, the `My_Event_Handler` class overloads the `handle_input` method, which is defined in the `ACE_Event_Handler` base class. However, this function has a side-effect of "hiding" the method in the base class, which is usually undesirable.

Example: Although overloading of base class methods is problematic with C++ applications in general, it is particularly tricky for classes that subclass from `ACE_Event_Handler`. In particular, application developers may not realize that the `handle_input` method on an `My_Event_Handler` object will not be dispatched when input events occur on the `ACE_Reactor` that this object is registered with because the signatures do not match exactly. To avoid this problem altogether, simply abide by the following design rule:

Design rule 15: *Do not overload any* `handle_*` *methods in* `ACE_Event_Handler` *subclasses.* Naturally, it is fine to *override* these methods as necessary in order to customize the behavior of the desired `handle_*` hooks. Many C++ compilers warn about this behavior, but it's generally good to avoid it whenever possible to avoid confusion.

5.10 Beware of Deadlock in Multi-threaded Reactor Applications

Context: Although Reactor's are generally used to implement single-threaded concurrent applications, they also can be used to in multi-threaded applications. In this context, it's important to beware of deadlock between multiple threads that are sharing a common `ACE_Reactor`.

Example: When the `ACE_Reactor` dispatches a callback to an event handler's `handle_*` method it holds an `ACE_Token` for the duration of the callback. This `ACE_Token` is defined as a *recursive mutex* [24], which keeps track of the identify of the thread that holds the mutex in order to avoid "self-deadlock." If the dispatched `handle_*` method directly or indirectly calls back into the `ACE_Reactor` *within the same thread of control*, the `ACE_Token`'s `acquire` method detects this automatically and simply increases its count of the lock recursion nesting depth, rather than deadlocking the thread.

Even with recursive mutexes, however, it is still possible to incur deadlock if (1) the original `handle_*` callback method makes a blocking call to a method in another thread and (2) that method in the second thread directly or indirectly calls into the same `ACE_Reactor`. In this case, deadlock occurs since the `ACE_Reactor`'s `ACE_Token` does not realize that the second thread is calling on behalf of the first thread where the `handle_*` hook method was originally dispatched.

To avoid deadlock when using the `ACE_Reactor` in a multi-threaded application, try to apply the following design rule:

Design rule 16: *Do not make blocking calls to other threads in* `handle_*` *methods if these threads will directly* *or indirectly call back into the same* ACE_Reactor. It may be necessary to use an ACE_Message_Queue to exchange information asynchronously if a handle_* callback method must communicate with another thread that accesses the same ACE_Reactor.

5.11 Ensure Controlling Thread Owns Event-loop in Multi-threaded Reactor Applications

Context: Only one thread of control at a time can invoke the handle_events method on an ACE_Reactor. To ensure this, each ACE_Reactor keeps track of the thread of control that "owns" its event loop. By default, the owner of an ACE_Reactor is the identifier of the thread that initialized it. There are certain use-cases, however, where the thread that initializes an ACE_Reactor is different from the thread that ultimately runs its event-loop via the handle_events method.

Example: Consider the following erroneous C++ code fragment:

```
class My_Server :
  public ACE_Task<ACE_NULL_SYNCH>
{
public:
  // Initializer.
  int open (size_t size) {
    // This call sets the owner
    // of the event-loop to
    // <ACE_OS::thr_self>.
    reactor_.open (size);

    // Make this an Active Object
    // (inherited from <ACE_Task>).
    this->activate ();
    // ...
  }

  // Hook method entry point for
  // the Active Object.  This runs
  // in a new thread of control.
  int svc (void) {
    // ...

    // This call will return -1
    // since the new thread of
    // control doesn't ''own'' this
    // <reactor_> instance.
    reactor_.handle_events ();
  }

private:
  ACE_Reactor reactor_;
};
```

Since My_Server inherits from ACE_Task, calling its activate method will make the My_Server object become an Active Object [25], which runs its svc hook method entry point in a separate thread of control. Therefore, the handle_events method will fail since it still thinks the owner of this ACE_Reactor is the original thread that initialized it.

It is straightforward to fix this code by simply having the new thread of control become the owner of the ACE_Reactor, as follows:

```
int svc (void) {
  // ...

  // Have this thread become the
  // new owner of <reactor_>'s event-loop.
  reactor_.owner (ACE_OS::thr_self ());

  // Now this call works correctly.
  reactor_.handle_events ();
  // ...
}
```

Design rule 17: *Ensure that the thread running an* ACE_Reactor*'s event loop becomes the controlling thread by assigning its thread identifier via the* owner *method.* The one exception to this rule is if the thread running the handle_events method is the same thread that created this particular instance of the ACE_Reactor.

6 Concluding Remarks

The ACE_Reactor is an OO framework designed to simplify the development of concurrent, event-driven distributed applications. By encapsulating low-level OS event demultiplexing mechanisms within an OO C++ interface, the ACE_Reactor makes it easier to develop correct, compact, portable, and efficient applications. Likewise, by separating *policies* and *mechanisms*, the ACE_Reactor enhances reuse, improves portability, and provides transparent extensibility.

The following C++ language features are used to enhance to the design of the ACE_Reactor and its applications.

Classes: The encapsulation provided by C++ classes improves portability. For instance, the ACE_Reactor class shields applications from differences between OS event demultiplexers like WaitForMultipleObjects and select.

Objects: Registering concrete event handler *objects*, rather than stand-alone *functions*, with the ACE_Reactor helps integrate application-specific state together with the methods that use this state.

Inheritance and dynamic binding: These features facilitate transparent extensibility by allowing developers to enhance the functionality of the ACE_Reactor and its associated applications without modifying existing code.

Templates: C++ parameterized types help increase the reusability by factoring variability into uniform classes, which can be "plugged" into the generic templates. For instance, the ACE_Acceptor can be instantiated with SVC_HANDLERs other than Logging_Handler and PEER_ACCEPTORs other than ACE_SOCK_Acceptor.

Perhaps the greatest challenge to using the ACE_Reactor is that its "inversion of control" programming model makes it hard to determine where an application's main flow of control is executing. This is a common challenge with other callback-based dispatcher mechanisms, such as the X-windows event loop.

The C++ source code and documentation for the `ACE_Reactor` and ACE socket wrappers is available at `www.cs.wustl.edu/~schmidt/ACE.html`. Also included with this release are a suite of test programs and examples, as well as many other C++ wrappers that encapsulate named pipes, STREAM pipes, `mmap`, and the System V IPC mechanisms (*i.e.*, message queues, shared memory, and semaphores).

Acknowledgements

Thanks to Hans Rohnert and Bill Landi of Siemens for their helpful comments on this paper.

References

[1] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching," in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), pp. 529–545, Reading, Massachusetts: Addison-Wesley, 1995.

[2] D. C. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications," in *Proceedings of the* 6th *USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, April 1994.

[3] W. R. Stevens, *UNIX Network Programming, Volume 1: Networking APIs: Sockets and XTI, Second Edition*. Englewood Cliffs, NJ: Prentice Hall, 1998.

[4] H. Custer, *Inside Windows NT*. Redmond, Washington: Microsoft Press, 1993.

[5] D. C. Schmidt and C. Cleeland, "Applying Patterns to Develop Extensible ORB Middleware," *IEEE Communications Magazine*, vol. 37, April 1999.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison-Wesley, 1995.

[7] D. C. Schmidt and P. Stephenson, "Experiences Using Design Patterns to Evolve System Software Across Diverse OS Platforms," in *Proceedings of the* 9th *European Conference on Object-Oriented Programming*, (Aarhus, Denmark), ACM, August 1995.

[8] D. C. Schmidt, "Transparently Parameterizing Synchronization Mechanisms into a Concurrent Distributed Application," *C++ Report*, vol. 6, July/August 1994.

[9] I. Pyarali, T. H. Harrison, and D. C. Schmidt, "Asynchronous Completion Token: an Object Behavioral Pattern for Efficient Asynchronous Event Handling," in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, Massachusetts: Addison-Wesley, 1997.

[10] R. E. Barkley and T. P. Lee, "A Heap-Based Callout Implementation to Meet Real-Time Needs," in *Proceedings of the USENIX Summer Conference*, pp. 213–222, USENIX Association, June 1988.

[11] D. C. Schmidt, D. L. Levine, and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, pp. 294–324, Apr. 1998.

[12] D. E. Comer and D. L. Stevens, *Internetworking with TCP/IP Vol II: Design, Implementation, and Internals*. Englewood Cliffs, NJ: Prentice Hall, 1991.

[13] G. Varghese and T. Lauck, "Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility," in *The Proceedings of the* 11th *Symposium on Operating System Principles*, November 1987.

[14] J. Hu, S. Mungee, and D. C. Schmidt, "Principles for Developing and Measuring High-performance Web Servers over ATM," in *Proceedings of INFOCOM '98*, March/April 1998.

[15] J. Hu, I. Pyarali, and D. C. Schmidt, "Measuring the Impact of Event Dispatching and Concurrency Models on Web Server Performance Over High-speed Networks," in *Proceedings of the* 2nd *Global Internet Conference*, IEEE, November 1997.

[16] D. C. Schmidt, "IPC_SAP: An Object-Oriented Interface to Interprocess Communication Services," *C++ Report*, vol. 4, November/December 1992.

[17] D. L. Presotto and D. M. Ritchie, "Interprocess Communication in the Ninth Edition UNIX System," *UNIX Research System Papers, Tenth Edition*, vol. 2, no. 8, pp. 523–530, 1990.

[18] G. Booch, *Object Oriented Analysis and Design with Applications (*2nd *Edition)*. Redwood City, California: Benjamin/Cummings, 1994.

[19] D. C. Schmidt, "Acceptor and Connector: Design Patterns for Initializing Communication Services," in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, Massachusetts: Addison-Wesley, 1997.

[20] D. C. Schmidt and T. Suda, "Transport System Architecture Services for High-Performance Communications Systems," *IEEE Journal on Selected Areas in Communication*, vol. 11, pp. 489–506, May 1993.

[21] A. Koenig, "When Not to Use Virtual Functions," *C++ Journal*, vol. 2, no. 2, 1992.

[22] T. H. Harrison, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-time CORBA Event Service," in *Proceedings of OOPSLA '97*, (Atlanta, GA), pp. 184–199, ACM, October 1997.

[23] P. S. Inc., *Purify User's Guide*. PureAtria Software Inc., 1996.

[24] D. C. Schmidt, "An OO Encapsulation of Lightweight OS Concurrency Mechanisms in the ACE Toolkit," Tech. Rep. WUCS-95-31, Washington University, St. Louis, September 1995.

[25] R. G. Lavender and D. C. Schmidt, "Active Object: an Object Behavioral Pattern for Concurrent Programming," in *Pattern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), Reading, Massachusetts: Addison-Wesley, 1996.