The Design and Implementation of the Reactor

An Object-Oriented Framework for Event Demultiplexing (Part 2 of 2)

Douglas C. Schmidt
schmidt@ics.uci.edu
Department of Information and Computer Science
University of California, Irvine, CA 92717, (714) 856-4105

An earlier version of this paper appeared in the September 1993 issue of the C++ Report.

1 Introduction

This is the second half of the third article in a series that describes techniques for encapsulating existing operating system (OS) interprocess communication (IPC) services using object-oriented (OO) C++ wrappers. In the first half of this article, a client/server application example was presented to motivate the utility of wrappers that encapsulate *I/O demultiplexing* mechanisms [1]. I/O demultiplexing is useful for developing event-driven network servers that receive and process data arriving from multiple clients simultaneously. The previous article also examined the advantages and disadvantages of several alternative I/O demultiplexing schemes such as non-blocking I/O, multiple process or thread creation, and synchronous I/O demultiplexing (via the `select` and `poll` system calls).

This article focuses on the design and implementation of an object-oriented framework called the `Reactor`. The `Reactor` provides a portable interface to an integrated collection of extensible, reusable, and type-secure C++ classes that encapsulate and enhance the `select` and `poll` I/O demultiplexing mechanisms [2]. To help simplify network programming, the `Reactor` combines the demultiplexing of synchronous I/O-based events together with timer-based events. When these events occur, the `Reactor` automatically dispatches the method(s) of previously registered objects to perform application-specified services.

This article is organized as follows: Section 2 describes the primary features offered by the `Reactor` framework; Section 3 outlines the object-oriented design and implementation of the framework; Section 4 presents a distributed logging example that demonstrates how the `Reactor` simplifies the development of concurrent, event-driven network applications; and Section 5 discusses concluding remarks.

2 Primary Features of the Reactor

The `Reactor` provides an object-oriented interface that simplifies the development of distributed applications that utilize I/O-based and/or timer-based demultiplexing mechanisms. The following paragraphs describe the primary features offered by the `Reactor` framework:

• **Export Uniform Object-Oriented Interfaces:** Applications using the `Reactor` do not access the lower-level I/O demultiplexing system calls directly. Instead, they inherit from a common `Event_Handler` abstract base class to form composite concrete derived classes (as illustrated in Figure 1). The `Event_Handler` base class specifies a uniform interface consisting of virtual methods that handle (1) synchronous input, output, and exceptions and (2) timer-based events. Applications create objects of these derived classes and register them with instances of the `Reactor`.

• **Automate Event Handler Dispatching:** When events occur, the `Reactor` automatically invokes the appropriate virtual method event handler(s) belonging to the preregistered derived class objects. Since C++ *objects* are registered with the `Reactor` (as opposed to stand-alone *subroutines*), any context information associated with an object is retained between invocations of its methods. This is particularly useful for developing "stateful" services that retain information in between client invocations.

• **Support Transparent Extensibility:** The functionality of both the `Reactor` and its registered objects may be extended transparently *without* modifying or recompiling existing code. To accomplish this, the `Reactor` framework employs inheritance, dynamic binding, and parameterized types to decouple the lower-level I/O demultiplexing and service dispatching *mechanisms* from the higher-level application processing *policies*. Example low-level mechanisms include (1) detecting events on multiple I/O descriptors, (2) handling timer expiration, and (3) invoking the appropriate method event handler(s) in response to events. Likewise, application-specified policies include (1) connection establishment, (2) data transmission and reception, and (3) processing service requests from other participating hosts.

• **Increase Reuse:** The `Reactor`'s demultiplexing and dispatching mechanisms may be reused by many network applications. By *reusing* rather than *redeveloping* these mechanisms, developers are free to concentrate on higher-level application-related issues, rather than repeatedly wrestling with lower-level event demultiplexing details. In addition, subsequent bug-fixes and enhancements are transparently shared by all applications that utilize the Reactor's components. Conversely, developers that access select and poll directly must reimplement the same demultiplexing and dispatching code for every network application. Moreover, any modifications and improvements to this code must be replicated manually in all related applications.

• **Enhance Type-Security:** The Reactor shields application developers from error-prone, low-level details associated with programming existing I/O demultiplexing system calls. These details involve setting and clearing bitmasks, handling timeouts and interrupts, and dispatching "call-back" methods. In particular, the Reactor eliminates several subtle causes of errors with poll and select that involve the misuse of I/O descriptors and fd_set bitmasks.

• **Improve Portability:** The Reactor also shields applications from differences between select and poll that impede portability. As illustrated in Figure 5, the Reactor exports the same interface to applications, regardless of the underlying I/O demultiplexing system calls. Moreover, the Reactor's object-oriented architecture improves its own internal portability. For example, porting the Reactor from a select-based OS platform to a poll-based platform required only a few well-defined changes to the framework.

In addition to simplifying application development, the Reactor also performs its demultiplexing and dispatching tasks efficiently. In particular, its event dispatching logic improves upon common techniques that use select directly. For instance, the select-based Reactor uses a Descriptor_Set class (described in Section 3.2) that avoids examining fd_set bitmasks one bit at a time in many circumstances. An article in a future issue of the C++ Report will empirically evaluate the performance of the Reactor and compare it against a non-object-oriented solution written in C that accesses I/O demultiplexing system calls directly.

3 The Object-Oriented Design and Implementation of the Reactor

This section summarizes the object-oriented design of the Reactor framework's primary class components, focusing on the interfaces and strategic design decisions. Where appropriate, tactical design decisions and certain implementation details are also discussed. Section 3.1 outlines the OS platform-independent components; Section 3.2 covers the platform-dependent components.

The Reactor was originally modeled after a C++ wrapper for select called the Dispatcher that is available with the InterViews distribution [3]. The Reactor framework described here includes several additional features. For example, the Reactor operates transparently on top of both the System V Release 4 poll interface and select, which is available on both UNIX and PC platforms (via the WINSOCK API). In addition, the Reactor framework

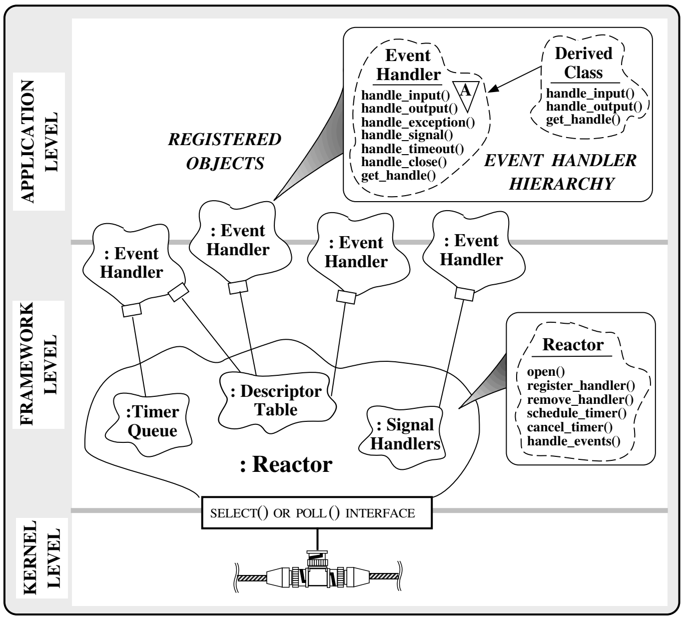

Figure 1: The Reactor Inheritance Hierarchy

contains support for multi-threading. In general, a single instance of the Reactor is active in a thread at any point in time. However, there may be multiple different instances of Reactor objects running in separate threads in a process. The framework provides the necessary synchronization operations to prevent race conditions [4].

3.1 Platform-Independent Class Components

The following paragraphs summarize the salient characteristics of the three platform-independent classes in the Reactor: the Time_Value, Timer_Queue, and Event_Handler classes:

• **Time_Value:** This class provides a C++ wrapper that encapsulates an underlying OS platform's date and time structure (such as the struct timeval data type on UNIX and POSIX platforms). The timeval structure contains two fields that represent time in terms of *seconds* and *microseconds*. However, other OS platforms use different representations, so the Time_Value class abstracts these details to provide a portable C++ interface.

The primary methods in the Time_Value class are illustrated in Figure 2. The Time_Value wrapper uses operator overloading to simplify time-based comparisons within the Reactor. Overloading permits the use of standard arithmetic syntax for relational expressions involving time comparisons. For example, the following code creates two Time_Value objects constructed by adding user-supplied command-line arguments to the current time, and then displaying the appropriate ordering relationship between the two objects:

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

```
// Time value structure from /usr/include/sys/time.h
// struct timeval { long secs; long usecs; };

class Time_Value
{
public:
  Time_Value (long sec = 0, long usec = 0);
  Time_Value (timeval t);

    // Returns sum of two Time_Values.
  friend Time_Value operator + (Time_Value,
                                Time_Value);
    // Returns difference between two Time_Values.
  friend Time_Value operator - (Time_Value,
                                Time_Value);

    // Relational and comparison operators for
    // normalized Time_Values.
  friend int operator < (Time_Value, Time_Value);
  friend int operator > (Time_Value, Time_Value);
  friend int operator <= (Time_Value, Time_Value);
  friend int operator >= (Time_Value, Time_Value);
  friend int operator == (Time_Value, Time_Value);
  friend int operator != (Time_Value, Time_Value);
private:
  // ...
};
```

Figure 2: Interface for the Time_Value Class

```
  if (argc != 3) {
    cerr << "usage: " << argv[0]
         << " time1 time2" << endl;
    return 1;
  }

  Time_Value ct =
    Timer_Queue::current_time ();
  Time_Value tv1 =
    ct + Time_Value (::atoi (argv[1]));
  Time_Value tv2 =
    ct + Time_Value (::atoi (argv[2]));

  if (tv1 > tv2)
    cout << "timer1 is greater" << endl;
  else if (tv2 > tv1)
    cout << "timer2 is greater" << endl;
  else
    cout << "timers are equal" << endl;
  return 0;
}
```

methods in the `Time_Value` class are implemented to compare "normalized" 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 `Time_Value (1, 1000000)` will compare equal to `Time_Value (2)`. Note that a direct bitwise comparison of the non-normalized class fields would not detect this equality.

- **Timer_Queue:** This class provides mechanisms that allow applications to register time-based objects derived from the `Event_Handler` base class described in the following bullet. The `Timer_Queue` ensures that the `handle_timeout` method in these objects is invoked at an application-specified time in the future. The methods of the `Timer_Queue` class illustrated in Figure 3 enable applications to schedule, cancel, and invoke the timer objects. The `Reactor`'s timer-based mechanisms may be used useful for distributed applications that require timer support. For example, reliable communication protocols require watch-dog timers that send retransmissions if their remote peers do not respond within a specific time interval. Likewise, certain daemon configuration frameworks (such as the `Service Control Manager` in Windows NT) require services under their control to periodically report their current status. These "heart-beat" messages are used to ensure that a service has not terminated abnormally.

By default, the `Timer_Queue` is implemented as a linked list of `<Time_Value, Event_Handler *, void *>` tuples. These tuples are sorted by the `Time_Value` field in ascending order of their "time-to-execute." The `Event_Handler *` field is a pointer to the timer object scheduled to be run at the time specified by the `Time_Value` field. The `void *` field is an argument supplied when a timer object is originally scheduled. When a timer expires, this argument is automatically passed to the `handle_timeout` method (described in the following bullet).

Each `Time_Value` in the linked list is stored in "absolute" time units (such as those generated by the UNIX `gettimeofday` system call). However, since virtual methods are used in the class interface, it is simple for applications to redefine the `Timer_Queue` implementation to use alternative data structures such as *delta-lists* [5] or *heaps* [6]. Delta-lists store time in "relative" units represented as offsets or "deltas" from the earliest `Time_Value` at the front of the list. Heaps, on the other hand, use a "partially-ordered, almost-complete binary tree" instead of a sorted list to reduce the average- and worst-case time complexity for inserting or deleting an entry from $O(n)$ to $O(\lg n)$. A heap representation may be more efficient for certain real-time application usage patterns [6].

- **Event_Handler:** This abstract base class specifies an extensible interface used by portions of the `Reactor` class that control and coordinate the automatic dispatching of I/O and timer mechanisms. The virtual methods in the `Event_Handler` interface are illustrated in Figure 4. The `Reactor` uses application-defined subclasses of the `Event_Handler` base class to implement its automated, event-driven call-back mechanisms. These subclasses may redefine certain virtual methods in the `Event_Handler` base class to perform application-defined processing in response to various types of events. These events include (1) different types (*e.g.,* "reading," "writing," and "exceptions") of synchronous I/O on one or more descriptors and (2) timer expiration.

An object of a subclass derived from `Event_Handler` typically supplies an I/O descriptor.[1] For example, the following `Client_Listener` class fragment represents a "passive-mode" server object based on the `SOCK_SAP` socket wrapper described in [7]. This class contains a `SOCK_Listener` object that encapsulates a socket descriptor:

```
class Client_Listener :
  public Event_Handler
```

[1] I/O descriptors may be omitted in derived class objects that are purely timer-based.

```
class Timer_Queue
{
public:
  Timer_Queue (void);

    // True if queue is empty, else false.
  int is_empty (void) const;

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

    // Returns current time of day.
  static const Time_Value &current_time (void);

    // Insert a new HANDLER and ARG that will be
    // invoked automatically at FUTURE_TIME when
    // the timer expires.  If INTERVAL is > 0, the
    // HANDLER will be reinvoked periodically
    // until explicitly canceled.
  void virtual schedule
    (const Event_Handler *handler,
     const void *arg,
     const Time_Value &future_time,
     const Time_Value &interval);

    // Locate and remove HANDLER from timer queue.
  void virtual cancel (const Event_Handler *handler);

    // Expire all timers whose values are <= EXPIRE_TIME
    // (note, this routine must be called manually since
    // it is not invoked asychronously).
  void virtual expire (const Time_Value &expire_time);
private:
  // ...
};
```

Figure 3: Interface for the Timer_Queue Class

```
{
public:
  Client_Listener (INET_Addr &addr) {
    this->listener.open (addr);
    // ...
  }

  virtual int get_handle (void) {
    return this->listener.get_handle ();
  }

  virtual int handle_input (int)
  {
    SOCK_Stream cli_handler;

    if (this->listener.accept (cli_handler) != -1) {
      char buf[BUFSIZ];
      int n;

      while ((n =
             cli_handler.recv (buf, sizeof buf)) > 0)
        ::write (1, buf, n);

      cli_handler.close ();
  }

  // ...

private:
  SOCK_Listener listener;
};

int main (int argc, char *argv[])
{
  Reactor reactor (Reactor::DEFAULT_SIZE);
  Client_Listener listener (INET_Addr (PORT_NUM));

  reactor.register_handler (&listener,
                            Event_Handler::READ_MASK);

  for (;;)
    reactor.handle_events ();
  /* NOTREACHED */
}
```

Internally, the `Reactor::register_handler` method retrieves the underlying I/O descriptor by invoking the `listener` object's `get_handle` virtual method. When the `Reactor::handle_events` method is invoked, descriptors of all registered objects are aggregated and passed to `select` or `poll`, which detect the occurrence of I/O-based events on these descriptors. When input events occur or output events become possible, the I/O descriptors become "active." At this time, the `Reactor` notifies the appropriate derived objects by invoking the method(s) that handle the event(s).[2] For instance, in the example above, when a connection request arrives the `Reactor` calls the `handle_input` method of the `Client_Listener` class. This method accepts the new connection, reads all data sent by the client, and displays the data on the standard output stream.

Time-based events are handled by the `Timer_Queue` class described above. When a timer managed by this queue expires, the `handle_timeout` method of a previously scheduled `Event_Handler` derived object is invoked. This method is passed the current time, as well as the `void` * argument passed in when the derived object was originally scheduled.

When any method in an `Event_Handler` object returns -1, the `Reactor` automatically invokes that object's `handle_close` method. The `handle_close` method may be used to perform any user-defined termination activities (such as deleting dynamic memory allocated by the object, closing log files, etc.). Upon return of this call-back function, the `Reactor` removes the associated derived class object from its internal tables.

3.2 Platform-Dependent Class Components

The `Reactor` class provides the central interface for the `Reactor` framework. On UNIX platforms, this class is the only part of the framework that contains platform-dependent code (the private representation of the `Time_Value` class may differ on non-UNIX platforms, as well).

• **Reactor:** Figure 6 illustrates the primary methods in the `Reactor` class interface, which encapsulates and extends the functionality of `select` and `poll`. These methods may be grouped into the following general categories:

- *Manager methods* – The constructor and `open` methods create and initialize objects of the `Reactor` by dynamically allocating various data structures (described in Section 3.2.1 and 3.2.2 below). The destructor and `close` methods deallocate these data structures.

[2]Note that the activated I/O descriptor is passed as an argument to the `handle_input` call-back function (though it is ignored in this case since the `listener` class instance variable keeps encapsulates the underlying descriptor).

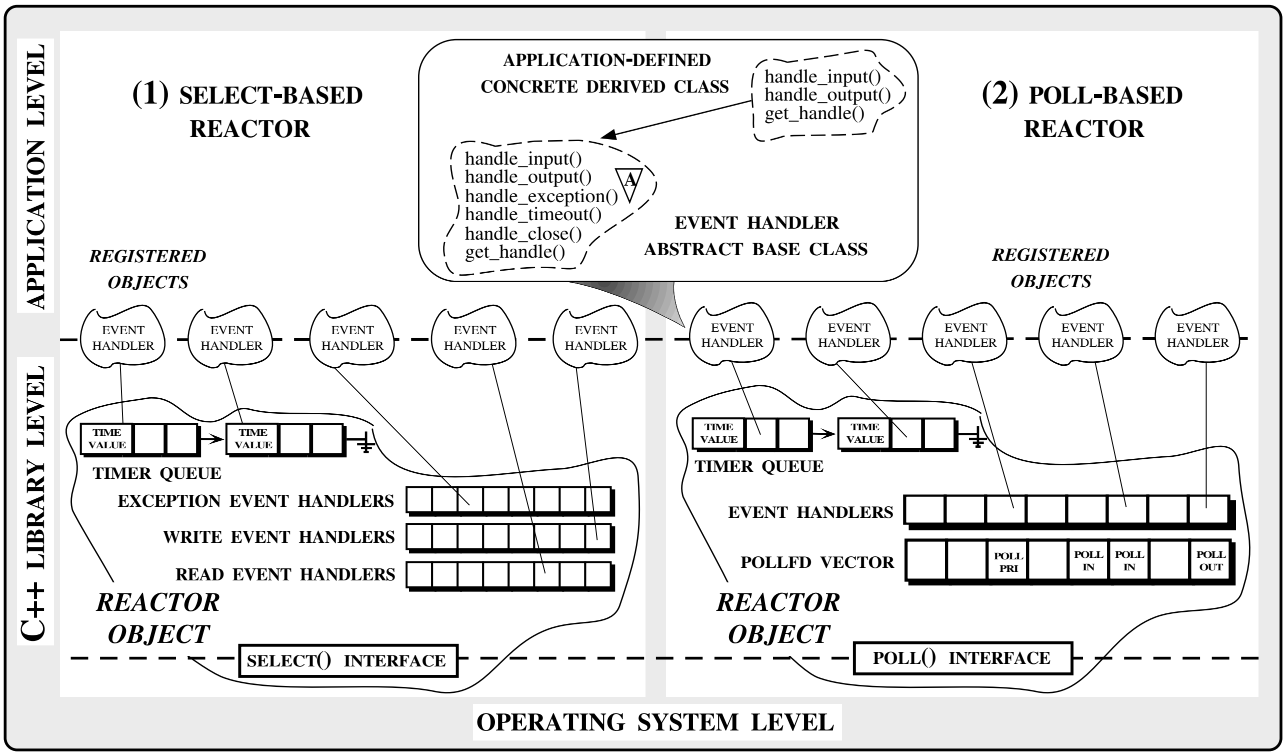

Figure 5: The Reactor's External Interfaces and Internal Data Structures

```
typedef unsigned long Reactor_Mask;
typedef int HANDLE;

class Event_Handler
{
public:
    // It is possible to bitwise "or" these values
    // together to instruct the Reactor to check for
    // multiple I/O activities on a single descriptor.
  enum {
    READ_MASK    = 01, WRITE_MASK  = 02, EXCEPT_MASK = 04,
    RWE_MASK     = READ_MASK | WRITE_MASK | EXCEPT_MASK
  };

    // Returns the I/O descriptor associated with the
    // derived object (must be supplied by a subclass).
  virtual HANDLE get_handle (void) const = 0;

    // Called when object is removed from the Reactor.n
  virtual int handle_close (HANDLE fd, Reactor_Mask);
    // Called when input becomes available on FD.
  virtual int handle_input (HANDLE fd);
    // Called when output is possible on FD.
  virtual int handle_output (HANDLE fd);
    // Called when urgent data is available on FD.
  virtual int handle_except (HANDLE fd);

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

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

Figure 4: Interface for the Event_Handler Class

- *I/O-based event handler methods* – Objects that are derived from subclasses of the `Event_Handler` class may be instantiated and registered with an instance of the `Reactor` via the `register_handle` method. Event handlers may also be removed via the `remove_handler` method.
- *Timer-based event handler methods* –

`Time_Value` arguments passed to the `Reactor`'s `schedule_timer` method are specified "relative" to the current time. For example, the following code schedules an object to print "hello world" every `interval` number of seconds, starting `delay` seconds into the future:

```
class Hello_World :
  public Event_Handler
{
public:
  virtual int handle_timeout (
      const Time_Value &tv,
      const void *arg) {
    cout << "hello world" << endl;
    return 0;
  }
  // ...
};

int main (int argc, char *argv[])
{
  if (argc != 3) {
    cerr << "usage: " << argv[0]
         << " delay interval" << endl;
    return 1;
  }
  Reactor reactor (Reactor::DEFAULT_SIZE);
```

```
  Hello_World handler; // timer object.
  Time_Value delay = ::atoi (argv[1]);
  Time_Value interval = ::atoi (argv[2]);

  reactor.schedule_timer (&handler, 0,
                          delay, interval);

  for (;;)
    reactor.handle_events ();
  /* NOTREACHED */
}
```

However, the default implementation of the underlying `Timer_Queue` stores the values in "absolute" time units. That is, it adds the scheduled time to the current time of day.

Since the interface of the `Reactor` class consists primarily of `virtual` methods it is straight-forward to extend the `Reactor`'s default functionality via inheritance. For example, modifying the `Timer_Queue` implementation to use one of the alternative representations described in Section 3.1 requires no visible changes to the `Reactor`'s `public` or `private` interface.

- *Event-loop methods –*

  After registering I/O-based and/or timer-based objects, an application enters an event-loop that calls either of the two `Reactor::handle_events` methods continuously. These methods block for an application-specified time interval awaiting the occurrence of (1) synchronous I/O events on one or more descriptors and (2) timer-based events. As events occur, the `Reactor` dispatches the appropriate method(s) of objects that the application registered to handle these events.

The following paragraphs describe the primary differences between the `poll`-based and `select`-based versions of the `Reactor`. Although the implementations of certain methods in the `Reactor` class differ across OS platforms, the method names and overall functionality remain the same. This uniformity stems from the modularity of the `Reactor`'s design and implementation, which enhances its reuse, portability, and maintainability.

3.2.1 Class Components for the select-based Reactor

As shown in Figure 5 (1), the `select`-based implementation of the `Reactor` contains three dynamically allocated `Event_Handler *` arrays. These arrays store pointers to the registered `Event_Handler` objects that process the reading, writing, exception, and/or timer-based events. The `Descriptor_Set` class provides an efficient C++ wrapper for the underlying `fd_set` bitmask data type. An `fd_set` maps the I/O descriptor name-space onto a compact bit-vector representation and provides several operations for enabling, disabling, and testing bits corresponding to I/O descriptors. One or more `fd_sets` is passed to the `select` call. The `Descriptor_Set` class optimizes several common `fd_set` operations by (1) using "full-word" comparisons to minimize unnecessary bit manipulations and

```
class Reactor
{
public:
// Manager methods

  enum { DEFAULT_SIZE = 256 };

    // Initialize a Reactor instance that may
    // contain SIZE entries (RESTART indicates
    // to restart system calls after interrupts).
  Reactor (int size, int restart = 0);

  virtual int open (int size = DEFAULT_SIZE,
                    int restart = 0);

    // Default constructor.
  Reactor (void);

    // Perform cleanup activities to close down
    // an instance of a REACTOR.
  void close (void);

// I/O-based event handler methods

    // Register an Event_Handler object according to
    // the Reactor_Mask(s) (i.e., "reading," "writing,"
    // and/or "exceptions").
  virtual int register_handler (const Event_Handler *,
             Reactor_Mask = Event_Handler::READ_MASK);

    // Remove the handler associated with the
    // appropriate Reactor_Mask(s).
  virtual int remove_handler (const Event_Handler *,
           Reactor_Mask = Event_Handler::READ_MASK);

// Timer-based event handler methods

    // Register a handler to expire at time DELTA.
    // When DELTA expires the handle_timeout()
    // method will be called with the current time
    // and ARG as parameters.  If INTERVAL is > 0
    // then the handler will be reinvolved periodically
    // at that INTERVAL.  Note that DELTA is interpreted
    // "relative" to the current time of day.
  virtual void schedule_timer (
       const Event_Handler *,
       const void *arg,
       const Time_Value &delta,
       const Time_Value &interval =
         Timer_Queue::zero);

    // Locate and cancel timer.
  virtual void cancel_timer (const Event_Handler *);

// 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 (Time_Value &tv);

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

Figure 6: Interface for the Reactor Class

(2) caching certain values to avoid recalculating bit-offsets on each call.

3.2.2 Class Components for the poll-based Reactor

The `poll` interface is more general than `select`, allowing applications to wait for a wider-range of events (such as "priority-band" I/O events). Therefore, the `poll`-based `Reactor` implementation shown in Figure 5 (2) is somewhat smaller and less complicated than the `select`-based version. For example, the `poll`-based `Reactor` requires neither the three `Event_Handler *` arrays nor the `Descriptor_Set` class. Instead, a single array of `Event_Handler` pointers and an array of `pollfd` structures are dynamically allocated and used internally to store the registered `Event_Handler` derived class objects.

4 Using and Evaluating the Reactor

The `Reactor` framework is intended to simplify the development of distributed applications, particularly network servers. To illustrate a typical usage of the `Reactor`, the following section examines the design and implementation of the distributed logging application presented in [1]. This section describes the primary C++ class components in the logging application, compares the object-oriented `Reactor`-based solution with an earlier version written in C, and discusses the influence of C++ on both the `Reactor` framework and the distributed logging facility.

4.1 Distributed Logging Facility Overview

The distributed logging facility presented below was originally designed for a commercial on-line transaction processing product. The logger facility uses a client/server architecture to provide logging services for workstations and symmetric multi-processors linked across local area and/or wide area networks. The logging facility combines the I/O demultiplexing and dispatching features of the `Reactor` together with the object-oriented interface to BSD sockets and the System V Transport Layer Interface (TLI) provided by the `IPC_SAP` wrapper library (described in [7]). Logging provides an "append-only" storage service that records diagnostic information sent from one or more applications. The primary unit of logging is the *record*. Incoming records are appended to the end of a log and all other types of write access are forbidden.

The distributed logging facility is comprised of the following three main components depicted in Figure 7:

• **Application Logging Interface:** Application processes (*e.g.,* P_1, P_2, P_3) running on client hosts use the `Log_Msg` C++ class to generate various types of logging records (such as LOG_ERROR and LOG_DEBUG). The `Log_Msg::log` method provides a `printf`-style interface. Figure 8 describes the priority levels and data format for records exchanged by the application interface and the logging dae-

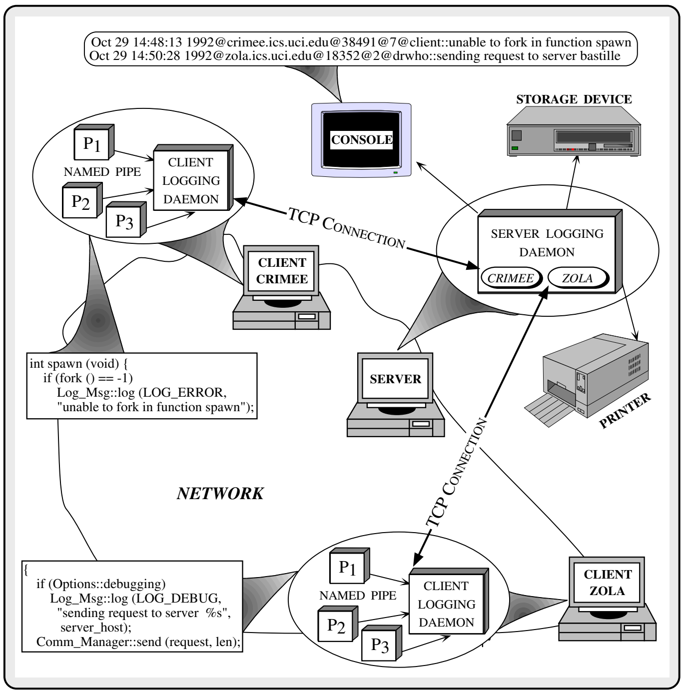

Figure 7: Run-time Activities in the Distributed Logging Facility

mons. When invoked by an application, the logging interface formats and timestamps these records and writes them to a well-known named pipe (also called a FIFO), where they are consumed by a *client logging daemon*.

• **Client Logging Daemon:** The client logging daemon is a single-threaded, iterative daemon that runs on every host machine participating in the distributed logging service. Each client logging daemon is connected to the read-side of the named pipe used to receive logging records from applications on this machine. Named pipes are used since they are an efficient form of localhost-only IPC. In addition, the semantics of named pipes in System V Release 4 UNIX have been expanded to allow "priority-band" messages that may be received in "order-of-importance," as well as in "order-of-arrival" (which is still the default behavior) [8].

The complete design of the client logging daemon and the application logging interface will appear in a subsequent C++ Report article that presents C++ wrappers for several "local-host" IPC mechanisms (such as System V Release 4 FIFOs, STREAM pipes, message queues, and UNIX-domain stream sockets). In general, a client logging daemon continuously receives the logging records in priority order from applications, converts the multi-byte record header fields into network-byte order, and forwards the records to the *server logging daemon* (which typically runs on a remote host).

• **Server Logging Daemon:** The server logging daemon is a concurrent daemon that continuously collects, reformats, and displays the incoming logging records to various external

```
// The following data type indicates the relative
// priorities of the logging messages, from lowest
// to highest priority...

enum Log_Priority
{
    // Shutdown the logger.
  LOG_SHUTDOWN = 1,
    // Messages indicating function-calling sequence
  LOG_TRACE = 2,
    // Messages that contain information normally
    // use only when debugging a program
  LOG_DEBUG = 3,
    // Informational messages
  LOG_INFO = 4,
    // Conditions that are not error condition
    // but that may require special handling.
  LOG_NOTICE = 5,
    // Warning messages
  LOG_WARNING = 6,
    // Initialize the logger
  LOG_STARTUP = 7,
    // Error messages
  LOG_ERROR = 8,
    // Critical conditions, e.g., hard device errors
  LOG_CRITICAL = 9,
    // A condition that should be corrected immediatey,
    // such as a corrupted system database.
  LOG_ALERT = 10,
    // A panic condition (broadcast to all users)
  LOG_EMERGENCY = 11,
    // Maximum logging priority + 1
  LOG_MAX = 12
};

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

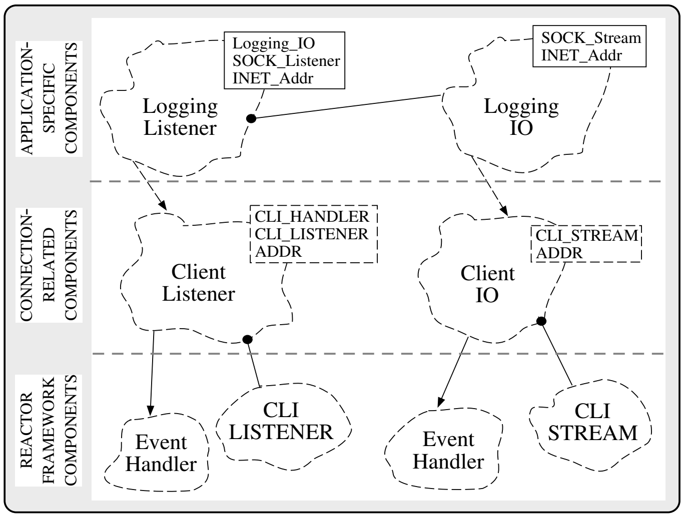

Figure 9: Class Components in the Server Logging Daemon

devices. These devices may include printers, persistent storage repositories, or logging management consoles. The remainder of this article focuses on the server logging daemon. In addition, several `Reactor` and `IPC_SAP` mechanisms are also illustrated and described throughout this example.

4.2 The Server Logging Daemon

The following section discusses the interface and implementation of the primary classes used to construct the server logging daemon. The logging server is a single-threaded, concurrent daemon that runs in a single process. Concurrency is provided by having the `Reactor` "time-slice" its attention to each active client in a round-robin fashion. In particular, during every invocation of the `Reactor::handle_events` method, a single logging record is read from each client whose I/O descriptor became active during this iteration. These logging records are written to the standard output of the server logging daemon. This output may be redirected to various devices such as printers, persistent storage repositories, or logging management consoles.

In addition to the main driver program shown below in Section 4.2.3, several other C++ class components appear in the logging facility architecture. The class inheritance and parameterization relationships between the various components are illustrated in Figure 9 using Booch notation [9]. To enhance reuse and extensibility, the component shown in this figure are designed to decouple the following aspects of the application architecture:

- `Reactor` *framework components* – The components in the `Reactor` framework discussed in Section 3 encapsulate the lowest-level mechanisms for performing the I/O demultiplexing and service dispatching.
- *Connection-related mechanisms* – The components discussed in Section 4.2.1 represent a set of generic templates that provide reusable connection-related mecha-

```
// A template class for handling connection requests from
// a remote client.

template <class CLI_HANDLER,
          class CLI_LISTENER,
          class ADDR>
class Client_Listener :
  public Event_Handler
{
public:
  Client_Listener (Reactor *r, const ADDR &a);
  ~Client_Listener (void);

private:
  virtual HANDLE get_handle (void) const;
  virtual int handle_input (HANDLE);
  virtual int handle_close (HANDLE = -1,
    Reactor_Mask = Event_Handler::READ_MASK);

  CLI_LISTENER listener;     // Accept connections
  Reactor      *reactor;     // Performs I/O demuxing
};
```

Figure 10: Class Interface for Accepting Connections

nisms. In particular, the `Client_Listener` template class is a general-purpose class designed to accept network connections with remote clients. Likewise, the `Client_IO` template class is another general-purpose class designed to send and/or receive data to/from connected clients.

- *Application-specific services* – The components discussed in Section 4.2.2 represent the application-specific portion of the distributed logging facility. In particular, the `Logging_Listener` class supplies specific parameterized types to create a connection handling instantiation that is specific for the logging application. Likewise, the `Logging_IO` class is instantiated to provide the application-specific functionality necessary to receive and process logging records from remote clients.

In general, by adopting this highly-decoupled object-oriented decomposition, the development and maintenance of the server logging daemon was simplified significantly.

4.2.1 Connection-related Mechanisms

• **The Client_Listener Class:** The `Client_Listener` parameterized type 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 `Client_Listener` class. This class inherits from `Event_Handler`, which enables it to interact with the `Reactor` framework. In addition, this template class is parameterized by a composite `CLI_HANDLER` subclass (which must understand how to perform I/O with clients), a `SVC_LISTENER` class (which must understand how to accept client connections), and `ADDR` (which is a C++ wrapper for the appropriate address family). Classes that are instantiated from the `Client_Listener` template will be capable of (1) accepting connection requests sent from remote clients, (2) dynamically allocating an object of the `CLI_HANDLER` subclass, and (3) registering this object with an instance of the `Reactor`. In turn,

```
// Useful macros to increase code clarity
#define LM_ERROR_RETURN(X,Y) \
   do { Log_Msg::log X; return (Y); } while (0)
#define LM_ERROR(X) Log_Msg::log X

// Shorthand names
#define CH CLI_HANDLER
#define CL CLI_LISTENER

template <class CH, class CL, class ADDR>
Client_Listener<CH, CL, ADDR>::Client_Listener
                       (Reactor *r, const ADDR &addr)
                       : reactor (r),
                         listener (addr)
{
  if (listener.enable (O_NONBLOCK) == -1)
    LM_ERROR ((LOG_ERROR, "%p", "non-blocking I/O"));
}

template <class CH, class CL, class ADDR> HANDLE
Client_Listener<CH, CL, ADDR>::get_handle (void) const
{
  return this->listener.get_handle ();
}

template <class CH, class CL, class ADDR> int
Client_Listener<CH, CL, ADDR>::handle_close
                       (HANDLE, Reactor_Mask)
{
  return this->listener.close ();
}

template <class CH, class CL, class ADDR>
Client_Listener<CH, CL, ADDR>::~Client_Listener (void)
{
  this->handle_close ();
}

// Generic mechanism for accepting connections from
// client hosts

template <class CH, class CL, class ADDR> int
Client_Listener<CH, CL, ADDR>::handle_input (HANDLE)
{
  ADDR client_addr;
  CH *cli_handler = new CH;

  if (this->listener.accept (*cli_handler,
                             &client_addr) != -1) {
    cli_handler->set_client_addr (client_addr);
    if (this->reactor->register_handler (cli_handler,
          Event_Handler::READ_MASK) == -1)
      LM_ERROR_RETURN ((LOG_ERROR, "%p",
        "unable to register log handler"), -1);
  }
  else
    return -1;
}
```

Figure 11: Class Implementation for Accepting Connections

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

template <class CLI_STREAM, class ADDR>
class Client_IO : public Event_Handler
{
public:
  Client_IO (void);

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

  void set_client_addr (const ADDR &);
  operator CLI_STREAM &();

  virtual HANDLE get_handle (void) const;

  virtual int handle_input (HANDLE);
  virtual int handle_close (HANDLE,
                            Reactor_Mask);

protected:
  CLI_STREAM cli_stream;
  char host_name[MAXHOSTNAMELEN + 1];

private:
  // Ensure dynamic allocation
  virtual ~Client_IO (void) {};
};
```

Figure 12: Class Interface for Client I/O

```
#define CS CLI_STREAM

template <class CS, class ADDR>
Client_IO<CS, ADDR>::Client_IO (void)
{
}

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

template <class CS, class ADDR>
Client_IO<CS, ADDR>::operator CS &()
{
  return this->cli_stream;
}

template <class CS, class ADDR>  void
Client_IO<CS, ADDR>::set_client_addr
                        (const ADDR &addr)
{
  ::strncpy (this->host_name, addr.get_host_name (),
             MAXHOSTNAMELEN + 1);
}

// Initiate the virtual function call-back
// (must be supplied by a subclass!)

template <class CS, class ADDR>  int
Client_IO<CS, ADDR>::handle_input (HANDLE)
{
  return this->recv ();
}

template <class CS, class ADDR>  HANDLE
Client_IO<CS, ADDR>::get_handle (void) const
{
  return this->cli_stream.get_handle ();
}

template <class CS, class ADDR>  int
Client_IO<CS, ADDR>::handle_close (HANDLE,
                                   Reactor_Mask)
{
  this->cli_stream.close ();
  // Must be allocated dynamically!
  delete this;
  return 0;
}
```

Figure 13: Class Implementation for Handling Client IO

the CLI_HANDLER class must know how to process data exchanged with the client.

Figure 11 depicts the Client_Listener class implementation. The class constructor sets the SVC_LISTENER data member into "non-blocking" mode, which expedites the processing of multiple connection requests that arrive consecutively from client logging daemons. When one or more connection requests arrive, the handle_input method is automatically dispatched by the Reactor. This method behaves as follows. First, it dynamically creates a separate CLI_HANDLER object, which is responsible for processing the logging records received from each new client. Next, it accepts an incoming connection into that object. Finally, it registers the newly created CLI_HANDLER object with an instance of the Reactor.

• **The Client_IO Class:** The Client_IO parameterized type provides a generic template for processing data sent from clients. In the distributed logging facility, for example, the I/O format involves logging records. However, different formats are easily substituted for other applications. Typically, objects of classes instantiated from Client_IO are dynamically created and registered with the Reactor by the handle_input routine in the Client_Listener class. The interface of the Client_IO class is depicted in Figure 12. As with the Client_Listener class, this class inherits functionality from the Event_Handler base class.

Figure 13 illustrates the Client_IO class implementation. The class constructor 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 Client_IO::handle_input method simply invokes the pure virtual method recv. This recv function must be supplied by subclasses of Client_IO; it is responsible for performing application-specific I/O behavior. Note how the combination of inheritance, dynamic binding, and parameterized types further decouples the general-purpose portions of the framework (such as connection establishment) from the application-specific functionality (such as receiving logging records).

When the Reactor removes a Client_IO object from its internal tables, the object's handle_close method is called automatically. By default, this method deallocates the object's memory (which was originally allocated by the handle_input method in the Client_Listener class). Objects are typically removed when a client logging daemon shuts down or when a serious transmission error occurs. To insure that Client_IO objects are only allocated and deallocated dynamically, the destructor is declared in the private section of the class (shown at the bottom of Figure 12).

4.2.2 Application-Specific Services

• **The Logging_Listener Class:** To implement the distributed logging application the Logging_Listener class is instantiated as follows:

```
typedef Client_Listener
  <Logging_IO,
   SOCK_Listener,
   INET_Addr> Logging_Listener;
```

The CLI_HANDLER parameter is instantiated with the Logging_IO class (described in the following bullet below), SVC_LISTENER is replaced by the SOCK_Listener class, and ADDR is the INET_Addr class.

The SOCK_* and INET_Addr instantiated types are part of a C++ wrapper called SOCK_SAP [7]. SOCK_SAP encapsulates the BSD socket interface for transferring data reliably between two processes that may run on different host machines. However, these classes could also be any other network interface that conformed to the interface used in the parameterized class (such as the TLI_SAP wrapper for the System V Transport Layer Interface (TLI)). 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 Client_IO class to use either SOCK_SAP or TLI_SAP as follows:

```
// Logging application.

#if defined (MT_SAFE_SOCKETS)
typedef SOCK_Stream PEER_STREAM;
#else
typedef TLI_Stream PEER_STREAM;
#endif // MT_SAFE_SOCKETS.

class Logging_IO
  : public Logging_IO<PEER_STREAM,
                      INET_Addr>
{
  // ...
};
```

The degree of flexibility offered by this template-based approach is extremely useful when developing applications that must run portability across multiple OS platforms. In fact, the ability to parameterize applications by transport interface is useful across variants of OS platforms (*e.g.,* SunOS 5.2 does not provide a thread-safe socket implementation).

• **The Logging_IO Class:** The Logging_IO class is created by instantiating the Client_IO class as follows:

```
class Logging_IO :
  public Client_IO<SOCK_Stream, INET_Addr>
{
public:
  virtual int recv (void);
};
```

The CLI_STREAM parameter is replaced with the SOCK_Stream class and the ADDR parameter is replaced with the INET_Addr class. The recv pure virtual method called when input arrives on the underlying SOCK_Stream is then implemented as follows:

```
// Callback routine for handling the reception of
// remote logging transmissions from clients.

int
Logging_IO::recv (void)
{
  long n;
  long len;

  n = this->cli_stream.recv (&len, sizeof len);
  switch (n) {
  case -1:
    LM_ERROR_RETURN ((LOG_ERROR, "%p at host %s\n",
      "client logger", this->host_name), -1);
    /* NOTREACHED */
  case 0:
    LM_ERROR_RETURN ((LOG_ERROR,
      "closing log daemon at host %s\n",
      this->host_name), -1);
    /* NOTREACHED */
  case sizeof (long):
    {
      Log_Record lr;

      len = ntohl (len);
      n = this->cli_stream.recv_n (&lr, len));
      if (n != len)
        LM_ERROR_RETURN ((LOG_ERROR, "%p at host %s\n",
          "client logger", this->host_name), -1);
      /* NOTREACHED */

      lr.decode ();

      if (lr.len == n)
        lr.print (this->host_name, 0, stderr);
      else
        LM_DEBUG ((LOG_DEBUG,
                   "error, lr.len = %d, n = %d\n",
                   lr.len, n));
      break;
    }
  default:
    LM_ERROR_RETURN ((LOG_ERROR, "%p at host %s\n",
      "client logger", this->host_name), -1);
    /* NOTREACHED */
  }

  return 0;
}
```

Note how this example perform two recvs to simulate a message-oriented service via the underlying TCP connection (recall that TCP provides bytestream-oriented, rather than record-oriented, service). The first recv reads the length (stored as a fixed-size integer) of the following logging record. The second recv then reads "length" bytes to obtain the actual record. Naturally, the client sending this message must also follow this message framing protocol.

4.2.3 The Main Driver Program

The following event-loop drives the Reactor-based logging server:

```
int
main (int argc, char *argv[])
{
  Reactor reactor (Reactor::DEFAULT_SIZE);
  const char *program_name = argv[0];
  Log_Msg::open (program_name, Log_Msg::STDERR);

  if (argc != 2)
    Log_Msg::log (LOG_ERROR,
                  "usage: %n port-number%a");

  unsigned short server_port = atoi (argv[1]);

  Logging_Listener
    listener (&reactor, INET_Addr (server_port));

  if (reactor.register_handler (&listener) == -1)
    Log_Msg::log (LOG_ERROR, "%p\n%a",
      "problem registering listener", -1);

  // Loop forever, handling client requests.
```

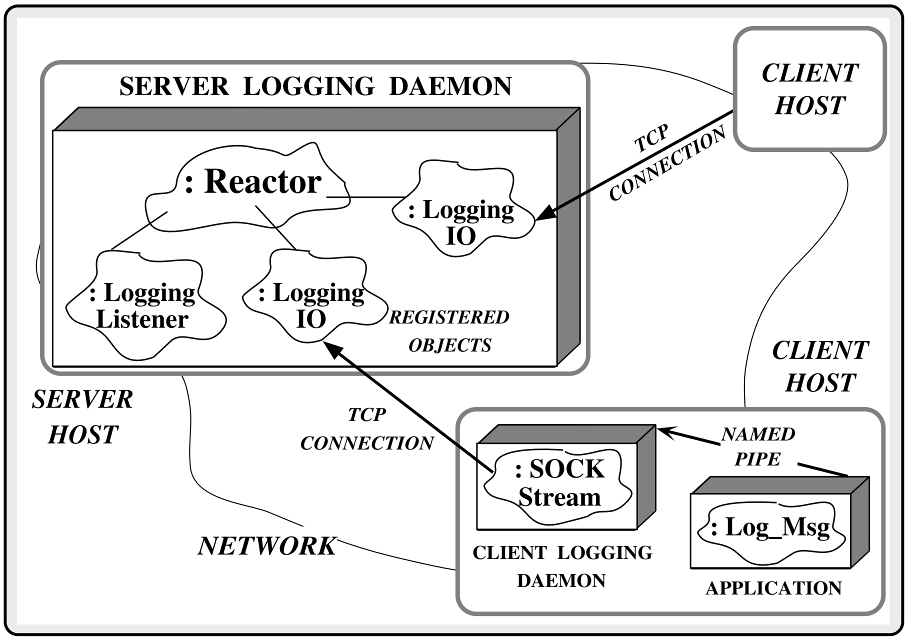

Figure 14: Run-time Configuration of the Server Logging Daemon

```
  for (;;)
    reactor.handle_events ();

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

The main program starts by opening a logging channel that directs 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. Note that since this arrangement does *not* recursively use the server logging daemon there is no danger causing of an "infinite-logging-loop."

The server then opens an instance of the `Reactor`, instantiates a `Logging_Listener` object, and registers this object with the `Reactor`. Next, 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 contacted the `Reactor` and become participants in the distributed logging service. As shown in the figure, a `Logging_IO` object has been dynamically instantiated and registered for each client. As incoming events arrive, the `Reactor` handles them by automatically dispatching (1) the `handle_input` method of the `Logging_Listener` and `Logging_IO` class. When connection requests arrive from client logging daemons, the `Logging_Listener::handle_input` function is invoked. Likewise, when logging records or shutdown messages arrive from previously connected client logging daemons, the `Logging_IO::handle_input` function is invoked.

Figure 7 portrays the entire system during execution. Logging records are generated from the application logging interface, forwarded to client logging daemons, transmitted across the network to the server logging daemon, and finally displayed on the server logging console. The logging information that is displayed indicates (1) the time the logging record was generated by the application 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) an arbitrary text string that contains the text of the logging message.

4.3 Evaluating Alternative Logging Implementations

This section compares the object-oriented and non-object-oriented versions of the distributed logging facility in terms of several software quality factors such as modularity, extensibility, reusability, and portability.

4.3.1 Non-Object-Oriented Version

The `Reactor`-based distributed logging facility is an object-oriented reimplementation of an earlier functionally equivalent, non-object-oriented logging facility. The original version was developed for a BSD UNIX-based commercial on-line transaction processing product. It was initially written in C and used BSD sockets and `select` directly. Later, it was ported to a system that provided only the System V-based TLI and `poll` interfaces.

The original C implementation is difficult to modify, extend, and port due to (1) tightly-coupled functionality and (2) an excessive use of global variables. For example, the I/O demultiplexing, service dispatching, and event processing operations in the original version are tightly-coupled with both the acceptance of client connection requests and the reception of client logging records. In addition, several global data structures are used to maintain the relationship between (1) per-client context information (such as the client hostname and current processing status) and (2) I/O descriptors that identify the appropriate context record. Therefore, any enhancements or modifications to the program directly affects the existing source code.

4.3.2 Object-Oriented Version

The object-oriented `Reactor`-based version uses data abstraction, inheritance, dynamic binding, and templates to (1) minimize the reliance on global variables and (2) decouple the application *policies* that process incoming connections and data from the lower-level *mechanisms* that perform demultiplexing and dispatching. The `Reactor`-based logging facility contains no global variables. Instead, each `Logging_IO` object registered with the `Reactor` encapsulates the client address and the underlying I/O descriptor used to communicate with clients.

By decoupling the policies and mechanisms, a number of software quality factors are enhanced. For example, the reusability and extensibility of system components was improved, which simplified both the initial development effort and subsequent modifications. Since the `Reactor` framework performs all the lower-level event demultiplexing and service dispatching, only a small amount of additional code is required to implement the server logging daemon described in Section 4.1. Moreover, the additional code is primarily concerned with application processing activities (such as accepting new connections and receiving client logging records). In addition, templates help to localize application-specific code to within a few, well-defined modules.

The separation of policies and mechanisms in the `Reactor`'s architecture facilitates extensibility and portability both "above" and "below" its public interface. For example, extending the server logging daemon's functionality (*e.g.,* adding an "authenticated logging" features) is straight-forward. Such extensions simply inherit from the `Event_Handler` base class and selectively implement the necessary virtual method(s). Likewise, by instantiating the `Client_Listener` and `Client_IO` templates, subsequent applications may be produced without redeveloping existing infrastructure. On the other hand, modifying the original non-object-oriented C version in this manner, however, would require direct changes to the existing code.

It is also possible to modify the `Reactor`'s underlying I/O demultiplexing mechanism without affecting existing application code. For example, porting the `Reactor`-based distributed logging facility from a BSD platform to a System V platform requires no visible changes to application code. On the other hand, porting the original C version of the distributed logging facility from sockets/`select` to TLI/`poll` was tedious and time consuming. It also introduced several subtle errors into the source code that did not manifest themselves until run-time. Furthermore, in certain communication-intensive applications, data is always available immediately on one or more descriptors. Therefore, polling these descriptors via non-blocking I/O may be more efficient than using `select` or `poll`. As before, extending the `Reactor` to support this alternative demultiplexing implementation not modify its public interface.

4.4 C++ Language Impact

Several C++ language features are instrumental to both the design of the `Reactor` and the distributed logging facility that utilizes its functionality. For example, the data hiding capabilities provided by C++ classes improves portability by encapsulating and isolating the differences between `select` and `poll`. Likewise, the technique of registering C++ class *objects* (rather that stand-alone *subroutines*) with the `Reactor` helps integrate application-specific context information together with multiple method that access this information. Parameterized types are useful for increasing the reusability of the `Client_Listener` class by allowing it to be instantiated with `CLI_HANDLER`s other than `Logging_IO` and `SVC_LISTENER`s other than `SOCK_Listener`. In addition, inheritance and dynamic binding facilitate transparent extensibility by allowing developers to enhance the functionality of the `Reactor` and its associated applications without modifying existing code.

Dynamic binding is used extensively in the `Reactor`. A previous C++ Report article on the `IPC_SAP` wrapper [7] discusses why avoiding dynamic binding is often advisable when designing "thin" C++ wrappers. In particular, the overhead resulting from indirect virtual table dispatching may discourage developers from using the more modular and type-secure OO interfaces. However, unlike `IPC_SAP`, the `Reactor` framework provides more than just a thin OO veneer around the underlying OS system calls. Therefore, the significant increase in clarify, extensibility, and modularity compensates for the slight decrease in efficiency. Moreover, the `Reactor` is typically used to develop distributed systems. A careful examination of the major sources of overhead in distributed systems reveals that most performance bottlenecks result from activities such as caching, latency, network/host interface hardware, presentation-level formatting, memory-to-memory copying, and process management [10]. Therefore, the additional memory reference overhead caused by dynamic binding is insignificant in comparison [11].

To justify these claims empirically, an upcoming article in the C++ Report will present the results of a benchmarking experiment that measures the performance of `IPC_SAP` and the `Reactor`. These performance results are based upon a distributed system benchmarking tool that measures client/server performance in a distributed environment. This tool also indicates the overhead of using C++ wrappers for IPC mechanisms. In particular, there are two functionally equivalent versions of the benchmarking tool: (1) an object-oriented version that uses the `IPC_SAP` and `Reactor` C++ wrappers and (2) a non-object-oriented version written in

C that uses the sockets, `select`, and `poll` system calls directly. The experiment measures the performance of the object-oriented implementation and the non-object-oriented implementation in a controlled manner.

5 Concluding Remarks

The `Reactor` is a object-oriented framework that simplifies the development of concurrent, event-driven distributed systems by making it easier to write correct, compact, portable, and efficient applications. It accomplishes this by encapsulating existing operating system demultiplexing mechanisms within an object-oriented C++ interface. In general, by separating *policies* and *mechanisms*, the `Reactor` supports reuse of existing system components, improves portability, and provides transparent extensibility.

One disadvantage with the `Reactor`-based approach is that it is somewhat difficult at first to conceptualize where an application's main thread of control occurs. This is a typical problem with event-loop-driven dispatchers such as the `Reactor` or the higher-level X-windows toolkits. However, the confusion surrounding this "indirect event-callback" dispatching model disappears quickly after writing several applications that use this approach.

The most recent version of the source code and documentation for both the `Reactor` and `IPC_SAP` C++ wrappers is located in the file `C++_wrappers.tar.Z` available via anonymous ftp from `ics.uci.edu` in the `~ftp/gnu/` directory. 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). Upcoming articles in the C++ Report will describe the design and implementation of these wrappers.

References

[1] D. C. Schmidt, "The Reactor: An Object-Oriented Interface for Event-Driven UNIX I/O Multiplexing (Part 1 of 2)," *C++ Report*, vol. 5, February 1993.

[2] W. R. Stevens, *Advanced Programming in the UNIX Environment*. Reading, Massachusetts: Addison Wesley, 1992.

[3] M. A. Linton and P. R. Calder, "The Design and Implementation of InterViews," in *Proceedings of the USENIX C++ Workshop*, November 1987.

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

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

[6] 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.

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

[8] UNIX Software Operations, *UNIX System V Release 4 Programmer's Guide: STREAMS*. Prentice Hall, 1990.

[9] G. Booch, *Object Oriented Analysis and Design with Applications (2^{nd} Edition)*. Redwood City, California: Benjamin/Cummings, 1993.

[10] 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.

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