Achieving Reuse Through Design Patterns

A Case Study of Evolving Object-Oriented
System Software Across OS Platforms

Douglas C. Schmidt† and Paul Stephenson‡
schmidt@ics.uci.edu and ebupsn@ebu.ericsson.se
†Department of Information and Computer Science, University of California, Irvine, CA 92717
‡Ericsson/GE Mobile Communications, Inc., Cypress, CA 90630

An earlier version of this paper appeared in the 3^{rd} SIGS C++ World Conference in Austin, Texas, November, 1994.

Abstract

Building system software that is reusable across OS platforms presents developers with many challenges. It is often difficult to reuse existing interfaces and implementations directly due to portability, functionality, and efficiency constraints imposed by different platforms and applications. It may still be possible, however, to leverage prior development effort by reusing design patterns. Design patterns embody recurring architectural themes that underlie solutions to requirements in particular problem domains. This paper presents a case study of an object-oriented framework written in C++ that was ported from several UNIX platforms to the Windows NT platform. The framework supports concurrent event demultiplexing and event handler dispatching. Fundamental differences in the event demultiplexing and I/O mechanisms on the UNIX and Windows NT platforms precluded direct reuse of many algorithms and interfaces in the framework. However, it was possible to reuse the underlying design patterns, which greatly reduced development effort and project risk.

1 Introduction

System software provides low-level services and mechanisms used by higher-level application software. System software is comprised of components such as libraries, frameworks, and utility programs. These components access and manipulate hardware devices (such as network adapters and disk drives) and software mechanisms residing within an OS kernel (such as alarm timers, synchronization objects, communication ports, and signal handlers).

The distinction between system software and application software is rather blurry (*e.g.,* is a debugger a system software artifact or application software artifact?). In general, system programmers design software components (such as dynamic linkers, device drivers, communication protocol stacks, and distributed file systems) that interface with lower-level hardware devices and operating system mechanisms. The direct consumers of system software are typically application programmers and other system programmers. In contrast, end-users are generally the consumers of application software (such as databases, graphical user interfaces, CAD/CAM products, and video-on-demand servers).

1.1 Challenges of Cross-Platform System Software Reuse

Developing system software that is capable of being directly reused on different OS platforms is challenging. Several key factors that complicate cross-platform reuse of system software are outlined below:

- *Efficiency* – Since other components will be layered upon system software, the techniques used to develop system software must not degrade performance significantly. Otherwise, developers may be inclined to build new, more efficient special-purpose code, rather than reuse existing components.
- *Portability* – In order to meet performance and functionality requirements, system software often must access highly non-portable mechanisms and interfaces (such as device registers within a network link-layer driver) in the underlying OS and hardware platform.
- *Lack of Functionality* – many OS platforms do not provide adequate functionality to develop portable, reusable system components. For example, the lack of kernel-level multi-threading, explicit dynamic linking, and asynchronous exception handling (as well as up-to-date C++ compilers that interact correctly with these features) greatly increases the complexity of developing and porting reusable system software.
- *Need to Master Complex Concepts* – Successfully developing robust, efficient, and portable system software requires intimate knowledge of complex mechanisms (such as concurrency control, interrupt handling, and interprocess communication) offered by one or more operating systems. It is also essential to understand the relative performance costs associated with using alternative mechanisms (such as shared memory vs. message passing) on different OS platforms.

There are trade-offs among the factors described above that further complicate the reuse of system software across platforms. Often, it may be difficult to develop portable system software that does not significantly degrade efficiency or subtly alter the semantics and robustness of commonly used operations. For instance, many traditional operating system kernels do not support pre-emptive multi-threading, and writing a portable user-level threads mechanism is often less efficient than programming with thread mechanisms supported by the kernel. Likewise, user-level threads may reduce robustness by restricting the use of OS features such as signals or synchronous I/O operations.

1.2 Reuse Techniques for System Software

Several techniques are useful for enhancing the reusability of system software. One relatively straightforward approach is to develop C++ wrappers that hide minor syntactic and semantic differences that exist among OS system call interfaces. A C++ wrapper transforms an existing interface to make it more object-oriented, without providing additional functionality. For example, the `IPC_SAP` class library described in [1] encapsulates the differences between BSD UNIX sockets and the System V UNIX Transport Layer Interface (TLI) within a type-safe C++ wrapper. Writing programs that utilize reusable `IPC_SAP` C++ wrapper interfaces facilitates modular development of portable network services such as remote login, file transfer, and distributed logging. C++ classes and parameterized types are useful programming language features for developing C++ wrappers.

A more sophisticated method for enhancing system software reuse is to develop object-oriented frameworks that shield applications from complex semantic differences between OS platforms. A framework is an integrated collection of components that collaborate to produce a reusable architecture for a family of related applications [2]. Object-oriented frameworks have been used to implement large-scale component reuse in domains such as graphical user interfaces [3], databases [4], operating system kernels [5], and communication subsystems [6, 7].

Like the C++ wrapper approach described above, a framework may be used to provide a common interface that is portable and reusable across OS platforms. Unlike the wrapper approach, however, a framework enables more extensible, larger-scale reuse of software. Frameworks provide integrated functionality that decouple the application-specific components in a system from the reusable application-independent components. Inheritance, dynamic binding, and object composition are C++ language features that help enforce dependencies and decouple implementations within a framework.

This paper presents a case study that describes the evolution of an object-oriented framework called the `Reactor` [8, 9]. The `Reactor` framework supports concurrent event demultiplexing and event handler dispatching. Event handlers are triggered by various types of events (such as timers, synchronization objects, signals, or I/O events). The primary contribution of this paper is to describe a reuse strategy that facilitates the development of efficient system software across OS platforms, even when the platforms possess fundamentally different functionality. In particular, the paper examines how we ported the components in the `Reactor` framework from several UNIX platforms to the Windows NT platform. UNIX and Windows NT provide significantly different mechanisms for event demultiplexing and I/O. To satisfy our performance requirements, it was not possible to directly reuse implementations or interfaces of the `Reactor` framework across OS platforms. However, it was possible to reuse the underlying *design patterns* that were embodied in the `Reactor` framework.

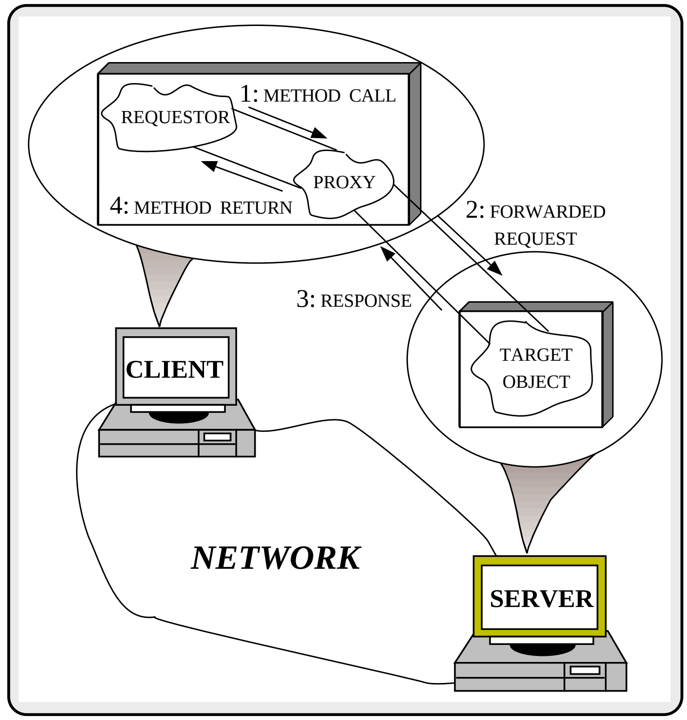

Figure 1: An Example of the Proxy Pattern

2 Design Patterns

A design pattern is a recurring architectural theme that provides a solution to a particular set of requirements within a problem domain [10]. For example, the remote object method invocation mechanism of a CORBA Object Request Broker (ORB) [11] is based on the *Proxy* design pattern [10] illustrated in Figure 1. In this pattern, one or more servers implement the methods and attributes of a remote object (such as an object that provides a distributed logging service). A client accesses a remote object indirectly via a local *proxy object*, which is a surrogate for the remote object. The proxy object defines the same interface as the remote object. Each method in the proxy simply marshals parameters and forwards method invocations to the remote object residing on a server.

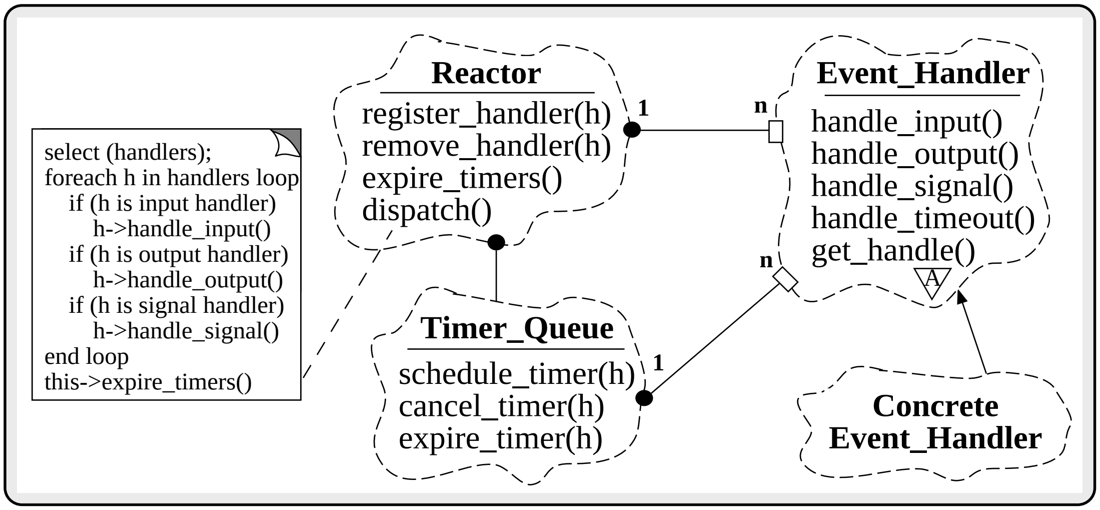

Figure 2: The Reactor Pattern

Design patterns facilitate reuse at an architectural level by providing "blueprints" or guidelines for defining and composing the components in a software system. In general, a large amount of reuse is possible at this level of abstraction, though there is often less reuse of existing components. In particular, reusing design patterns may not result in direct reuse of implementations, interfaces, or even detailed designs. For instance, the design and implementation of the CORBA remote object method invocation mechanism may vary radically across different vendors' ORBs and different OS platforms. Nevertheless, the Proxy design pattern is a recurring architectural theme throughout CORBA implementations, regardless of the vendor or OS platform.

Object-oriented frameworks typically embody a wide range of design patterns. For example, the ET++ graphical user-interface (GUI) framework [12] incorporates design patterns (such as Abstract Factory [10]) that hide the details of creating user-interface objects. This enables an application to be portable across different window systems (such as X windows and SunView). Likewise, the InterViews [3] GUI framework contains design patterns (such as Strategy and Iterator [10]) that allow algorithms and/or application behavior to be decoupled from mechanisms provided by the reusable GUI components.

In the distributed application domain, many components in the ADAPTIVE Service eXecutive (ASX) framework [7] represent design patterns. These design patterns address recurring distributed application software development themes (such as event demultiplexing, connection establishment, message routing, and flexible composition of hierarchically-related services). The ASX framework has been implemented on several UNIX platforms and on Windows NT. This paper focuses on two specific design patterns (the Reactor [13] and Accepter patterns) that are provided by the ASX framework.

2.1 The Reactor Pattern

The Reactor pattern is an object behavioral pattern. This pattern simplifies the development of event-driven applications (such as a CORBA ORB, an X-windows host resource manager, or a UNIX remote login service). The Reactor pattern provides a common infrastructure that integrates event demultiplexing and the dispatching of *event handlers*. Event handlers perform application-specific processing operations in response to various types of events. An event handler may be triggered by different operating system entities (such as timers, communication ports, synchronization objects, and signal handlers) that are monitored by an application.

The Reactor pattern provides several benefits to distributed applications:

- It enables multiple event handlers to wait simultaneously for events to occur on multiple entities monitored by an application *without* blocking or continuously polling for events on any single monitored entity.
- It decouples the application-specific portions of a service from the reusable application-independent mechanisms that implement event demultiplexing. This decoupling allows the event handlers to evolve independently of the event demultiplexing mechanisms provided by the underlying OS platform.

Figure 2 illustrates the participants in the Reactor pattern.[1] The `Reactor` class defines an interface for registering, removing, and dispatching `Event_Handler` objects. A `Reactor` implementation provides application-independent mechanisms that automatically perform demultiplexing and dispatching of application-specific concrete

[1] Relationships between components are illustrated throughout the paper via Booch notation [14]. Dashed clouds indicate classes; directed edges indicate inheritance relationships between classes; and an undirected edge with a solid bullet at one end indicates a composition relation between two classes. Solid clouds indicate objects; nesting indicates composition relationships between objects; and undirected edges indicate some type of link exists between two objects.

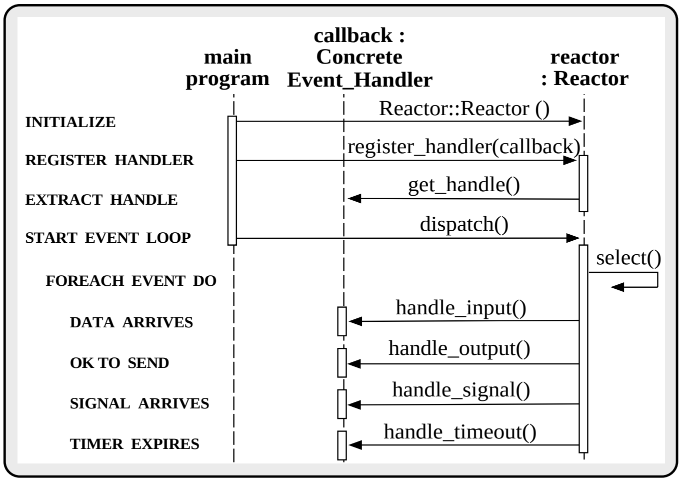

Figure 3: Object Interactions within the Reactor Pattern

event handlers. Time-based event dispatching is performed by a `Timer_Queue` object contained within the `Reactor`. Both the `Timer_Queue` and the `Reactor` class contain references to objects of `Concrete_Event_Handler` subclasses. These subclasses are derived from the `Event_Handler` abstract base class, which defines virtual methods for handling input, output, signal, and timeout events. A `Concrete_Event_Handler` subclass may override these virtual methods to perform application-specific functionality in response to the corresponding events.

When events occur at run-time, the `Reactor` dispatches the associated application-specific methods on pre-registered objects that are derived from the `Event_Handler` base class. This collaboration takes the form of the method callbacks depicted by the object interaction diagram shown in Figure 3. After initializing the `Reactor` by registering one or more event handlers, an application calls the `Reactor`'s `dispatch` method to perform its main event-loop.

Certain event handlers are triggered by the occurrence of events on descriptors that represent OS entities (such as I/O ports or synchronization objects). To bind the `Reactor` together with these descriptors, a subclass of `Event_Handler` must define the `get_handle` method. This method returns a descriptor that is used internally within the `Reactor`'s `dispatch` method to wait for certain events to occur. When these events occur, the `Reactor` dispatches the event handler associated with the descriptor. The code annotation in Figure 2 outlines the behavior of the `dispatch` method.

An alternative way to implement event demultiplexing is to use multi-threading. In this approach, an application spawns a separate thread for each entity it monitors. Every thread blocks until the entity it monitors receives an event. At this point, the appropriate event handler code is executed within the thread. Using multi-threading to implement event demultiplexing has several drawbacks, however. It may require the use of complex concurrency control schemes; it may lead to poor performance on uni-processors [7]; and it may not be available on many popular OS platforms.

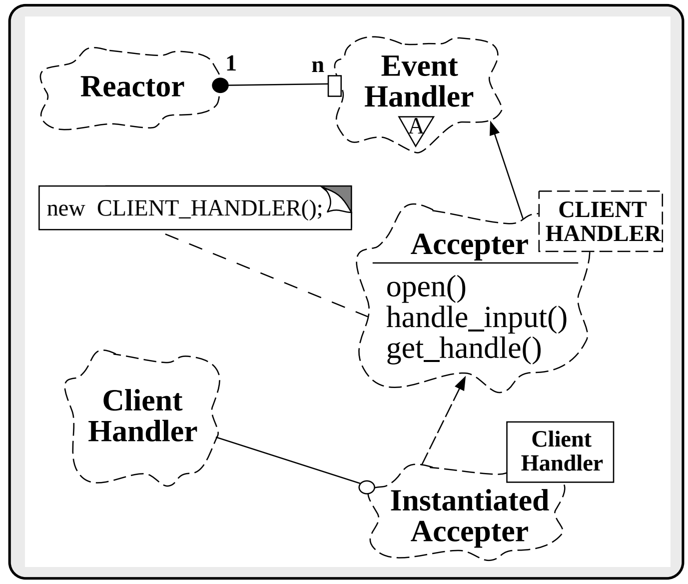

Figure 4: The Accepter Pattern

On the other hand, certain types of applications perform long-duration services such as file transfer or remote login. For these types of applications, multi-threading and/or multi-processing may be useful techniques for reducing development effort, improving application robustness, and transparently leveraging off of available multi-processor capabilities. Therefore, it is often useful to use the Reactor pattern in conjunction with OS multi-threading or multi-processing mechanisms, as described in Section 2.2.

2.2 The Accepter Pattern

The Accepter pattern is an object creational pattern. This pattern decouples the act of establishing a connection from the service(s) provided after a connection is established. Connection-oriented services (such as file transfer, remote login, distributed logging, and video-on-demand) are particularly amenable to this pattern. The Accepter pattern simplifies the development of these types of services by allowing the application-specific portion of a service to be modified independently of the mechanism used to establish a connection. Furthermore, by using other object-oriented language features (such as templates, inheritance, and dynamic binding) that are based on design patterns (such as Factory Method or Abstract Factory [10]), it is possible to completely parameterize the type of service offered by an instance of the Accepter pattern [13].

Figure 4 illustrates the participants in the Accepter pattern, which leverages off the interfaces and mechanisms provided by the Reactor pattern. The `open` method in template class `Accepter` initializes a communication endpoint to listen for incoming connection requests from clients. The `get_handle` method returns the I/O descriptor corresponding to the communication end-point. The Accepter class inherits the demultiplexing and dispatching interface from the Event_Handler base class. When a connection request arrives from a client, this interface is used by a Reactor object to trigger a callback on the Accepter's virtual handle_input method.

Each new connection request from a client causes the Accepter's handle_input method to create a new CLIENT_HANDLER object dynamically. In the example illustrated in Figure 4, CLIENT_HANDLER is a formal parameterized type argument defined in the Accepter template class. The instantiated Accepter class supplies an actual Client_Handler class parameter. This class parameter is responsible for implementing a particular application-specific service (*i.e.*, transferring a file, receiving logging records, etc.) that will interact with a client.

Note that the Accepter pattern does not dictate the behavior of Client_Handler objects it creates. In particular, a dynamically created Client_Handler object may be executed in any of the following ways:

1. *Run in the same thread of control* – This approach is implemented by inheriting the Client_Handler from Event_Handler and registering each newly created Client_Handler object with the Reactor. Thus, each Client_Handler object is dispatched in the same thread of control as an Accepter object.
2. *Run in a separate thread of control* – In this approach, a Reactor serves as the central event dispatcher within an application. When a client connects, the Accepter's handle_input method will spawn a separate thread of control and arrange for the Client_Handler object to process the connection within that new thread. Threads are useful for cooperating services that frequently reference common memory-resident data structures shared by the threads.
3. *Run in a separate OS process* – This approach is similar to the previous bullet. However, a separate process is created rather than a separate thread. Network services that base their security mechanisms on process ownership (such as the standard Internet ftp and telnet services) are typically executed in separate processes to prevent accidental or intentional access to unauthorized resources.

The ASX framework described in [7, 15] provides mechanisms that support all three of these types of behavior. The implementation described in the following section uses the single-threaded behavior described in the first bullet above.

3 Implementing the Design Patterns

This section outlines how the Reactor and Accepter design patterns were implemented on BSD and System V UNIX, as well as on Windows NT. The discussion emphasizes the relevant functional differences between the various OS platforms and describes how these differences affected the design and implementation of the patterns.

The implementation of the Reactor pattern was significantly affected by the semantics of the event demultiplexing and I/O mechanisms provided in the underlying operating system. In general, there are two types of demultiplexing and I/O semantics: *reactive* and *proactive*. Reactive I/O semantics (which are provided on standard BSD and System V UNIX systems [16]) allow an application to indicate to the OS which I/O descriptors to notify it about when an I/O-related operation (such as a read, write, and connection request/accept) may be performed without blocking. Subsequently, when the OS detects that the desired operation may be performed without blocking on any of the indicated descriptors, it informs the application that the descriptor(s) are ready. The application then "reacts" by handling the descriptor(s) accordingly (such as reading or writing data, accepting connections, etc.).

In contrast, proactive I/O semantics (which are provided on Windows NT [17] and VMS) allow an application to proactively initiate I/O-related operations (such as a read, write, or connection request/accept) or general-purpose event-signaling operations (such as a semaphore lock being acquired or a thread terminating). The invoked operation proceeds asynchronously and does not block the caller. When an operation completes, it signals the application. At this point, the application runs a completion routine that determines the exit status of the operation and potentially starts up another asynchronous operation.

For performance reasons, it was not feasible to completely encapsulate the variation in behavior between the UNIX and Windows NT I/O semantics. Therefore, we could not directly reuse existing C++ code, algorithms, or detailed designs. However, it was possible to capture and reuse the concepts that underlay the Reactor and Accepter design patterns.

We reduced our project's risk by reusing existing design patterns. These patterns provided a concise set of architectural blueprints that guided our porting effort from UNIX to Windows NT. In particular, we did not have to rediscover the key collaborations between participants in the patterns. Instead, our task was to determine a suitable mapping of the pattern participants onto the mechanisms provided by the different OS platforms. Finding an appropriate mapping was non-trivial, as we describe below. Nevertheless, our knowledge of the design patterns greatly reduced the amount of redevelopment effort.

3.1 UNIX Implementations

3.1.1 Implementing the Reactor Pattern on UNIX

The standard demultiplexing mechanisms on UNIX operating systems provide reactive I/O semantics. In particular, the UNIX select and poll event demultiplexing system calls inform an application which subset of descriptors within a set of I/O descriptors may send/receive messages or request/accept connections without blocking. Implementing the Reactor pattern using UNIX reactive I/O is straightforward. After `select` or `poll` indicate which I/O descriptors have become ready, the `Reactor` object reacts by invoking the appropriate `Event_Handler` callback methods (*i.e.*, `handle_input` or `handle_output`).

One advantage of the UNIX reactive I/O scheme is that it decouples (1) event detection and notification from (2) the operation performed in response to the triggered event. This allows an application to optimize its response to an event by using context information available when the event occurs. For example, a network server might check to see how many bytes are in a socket receive queue in order to determine the size of a buffer it allocates for a `recv` system call. A disadvantage of UNIX reactive I/O is that operations may not be invoked asynchronously to run in parallel with subsequent operations (unless threads are used).

The original implementation of the Reactor pattern provided by the ASX framework was derived from the `Dispatcher` class category available in the InterViews object-oriented GUI framework [3]. The `Dispatcher` is an object-oriented interface to the UNIX `select` system call. InterViews uses the `Dispatcher` to define an application's main event loop and to manage connections to one or more physical window displays. The `Reactor` framework's first modification to the original `Dispatcher` framework added support for signal-based event dispatching. The `Reactor`'s signal-based dispatching mechanism was modeled closely on existing mechanisms for timer-based and I/O descriptor-based event demultiplexing and event handler dispatching.

The next major modification to the `Reactor` occurred when porting it from SunOS 4.x (which is based primarily on BSD 4.3 UNIX) to SunOS 5.x (which is based primarily on System V release 4 (SVR4) UNIX). SVR4 provides another event demultiplexing interface via the `poll` system call. `Poll` is similar to `select`, though it uses a different interface and provides a broader, more flexible model for event demultiplexing that supports SVR4 features such as STREAM pipe band-data [18].

The SunOS 5.x port of the `Reactor` was enhanced to support either `select` or `poll` as the underlying event demultiplexer. Although portions of the `Reactor`'s internal implementation changed, its external interface remained the same for both the `select`-based and the `poll`-based versions. This common interface facilitates networking application portability between System V and BSD UNIX [9].

A portion of the public interface for the UNIX implementation of the Reactor pattern is shown below:

```
// Bit-wise or these values to check for
// multiple activities per-descriptor
enum Reactor_Mask {
  READ_MASK = 01, WRITE_MASK = 02, EXCEPT_MASK = 04,
  RWE_MASK = READ_MASK | WRITE_MASK | EXCEPT_MASK
};

class Reactor
{
public:
  // 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);

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

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

// ...
};
```

Likewise, the `Event_Handler` interface for UNIX is defined as follows:

```
typedef int HANDLE; // I/O descriptor

class Event_Handler
{
protected:
  // Returns the I/O descriptor associated with the
  // derived object (must be supplied by a subclass)
  virtual HANDLE get_handle (void) const;

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

  // 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);
};
```

The next major modification to the `Reactor` enabled it to be used in conjunction with multi-threaded applications on SunOS 5.x using Solaris threads [19]. Adding multi-threading support required changes to the internals of both the `select`-based and `poll`-based versions of the `Reactor`. These changes involved a SunOS 5.x mutual exclusion mechanism known as a "mutex." A mutex serializes the execution of multiple threads by defining a critical section where only one thread executes the code at a time [20]. Critical sections of the `Reactor`'s code that concurrently access shared resources (such as the `Reactor`'s internal table of `Event_Handler` objects) are protected by a mutex.

The standard SunOS 5.x synchronization type (`mutex_t`) provides support for *non-recursive* mutexes. The SunOS 5.x non-recursive mutex provides a simple and efficient form of mutual exclusion based on adaptive spin-locks. However, non-recursive mutexes possess the restriction that the thread currently owning a mutex may not reacquire the mutex without releasing it first. Otherwise, deadlock will occur immediately.

While developing the multi-threaded `Reactor`, it quickly became obvious that the default implementation of SunOS 5.x mutex variables was inadequate to support the synchronization semantics required by the `Reactor`. In particular, as described in Section 2.1, the Reactor's `dispatch` interface performs callbacks to methods of pre-registered, application-specific event handler objects. The following C++ pseudo-code illustrates the `dispatch` logic:

```
void Reactor::dispatch (void)
{
  for (;;) {
    // Block until one or more events occur
    this->wait_for_events (this->handler_set);

    this->lock->acquire (); // Obtain the mutex

    // Dispatch all the callback methods on
    // handlers who contain active events
    foreach active handler in this->handler_set {
      if (handler is an input handler)
         handler->handle_input (handler);
      if (handler is an output handler)
         handler->handle_output (handler);
      if (handler is a signal handler)
         handler->handle_signal (handler);
    }

    this->expire_timers (); // Handle timers
    this->lock->release (); // Release the mutex
  }
}
```

Callback methods (such as the `handle_input` and the `handle_output` methods) defined by the event handler objects may subsequently re-enter the `Reactor` object via its `register_handler` and `remove_handler` methods, as shown in the following C++ pseudo-code:

```
// Global per-process instance of the Reactor.
extern Reactor reactor;

// Application-specific callback method.

int Accepter::handle_input (HANDLE handle)
{
  Concrete_Event_Handler *new_handler =
    new Concrete_Event_Handler;

  *new_handler = this->accept (handle);

  // Re-enter the Reactor object.
  reactor.register_handler (new_handler,
                            READ_MASK);

  // ...
}
```

In this case, using non-recursive mutexes will result in deadlock since (1) the mutex within the `Reactor`'s `dispatch` method is locked throughout the callback and (2) the `Reactor`'s `register_handler` method tries to acquire the same mutex.

One solution to this problem involved recoding the `Reactor` to release its mutex lock before invoking callbacks to application-specific `Event_Handler` methods. However, this solution was tedious and error-prone. It also increased synchronization overhead by repeatedly releasing and reacquiring mutex locks. A more elegant and efficient solution used *recursive* mutexes to prevent deadlock and to avoid modifying the `Reactor`'s concurrency control scheme. A recursive mutex allows calls to its `acquire` method to be nested as long as the thread that owns the lock is the one attempting to re-acquire it. A portable C++ implementation of recursive mutexes for SunOS 5.x appears in [21].

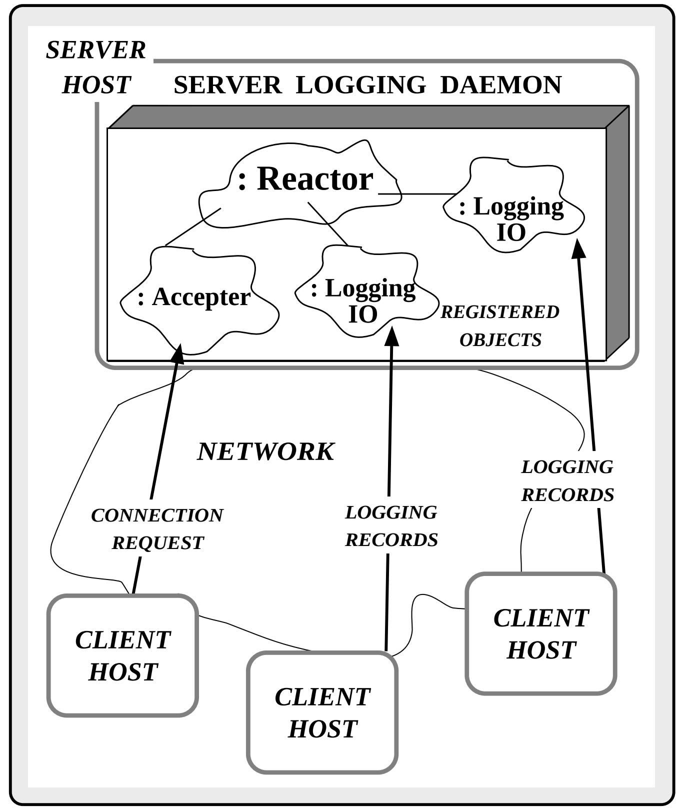

Figure 5: The Distributed Logging Facility

The current implementation of the UNIX-based Reactor pattern is about 1,400 lines of C++ code (not including comments or extraneous whitespace). This implementation is portable between both System V and BSD UNIX variants.

3.1.2 Implementing the Accepter Pattern on UNIX

To motivate and illustrate the Accepter pattern, consider the event-driven server for a distributed logging facility shown in Figure 5 [8, 9]. A distributed logging facility enables multiple applications running on client hosts to forward logging records to a server logging daemon[2] running on a designated host in a network. Logging records are event messages that contain error notifications, debugging information, status reporting, etc. Centralizing the logging activities of distributed applications within a single server daemon is useful since it consolidates status reporting and serializes access to shared output devices (such as consoles, printers, files, or network management information bases).

As shown in Figure 5, a client logging daemon communicates with the server logging daemon via an interprocess communication (IPC) channel (such as a TCP/IP connection). The server logging daemon processes logging records

[2] A daemon is an OS process that runs continuously "in the background" [16].

that arrive concurrently on multiple I/O descriptors (*i.e.*, one descriptor for each client connection). Moreover, the server also listens on a designated I/O descriptor that accepts connection requests from new clients who would like to participate in the logging service. Therefore, it is not feasible for the server process to perform long-duration operations by blocking on any individual I/O descriptor. Using blocking I/O would significantly delay the response time to handle logging records or connection requests arriving from clients that are bound to other descriptors.

A highly modular and extensible way to design the server logging daemon is to combine the Reactor and Accepter patterns. Together, these patterns decouple (1) the application-independent mechanisms that demultiplex and dispatch pre-registered `Event_Handler` objects from (2) the application-specific connection establishment and logging record transfer functionality performed by methods in these objects.

Within the server logging daemon, two subclasses of the `Event_Handler` base class (`Logging_IO` and `Accepter`) perform the actions required to process the different types of events arriving on different I/O descriptors. The `Logging_IO` event handler is responsible for receiving and processing logging records transmitted from a client. Likewise, the `Accepter` event handler is a factory that is responsible for accepting a new connection request from a client, dynamically allocating a new `Logging_IO` event handler to handle logging records from this client, and registering the new handler with an instance of a `Reactor` object.

The following code illustrates an implementation of a portion of the server logging daemon. An instance of the `Logging_IO` template class performs I/O between the server logging daemon and a particular instance of a client logging daemon. As shown in the code below, the `Logging_IO` class inherits from both `XPORT_IO` and `Event_Handler`. Inheriting from the template parameter `XPORT_IO` provides the reliable TCP capabilities used to transfer logging records between an application and the server. The use of templates removes the reliance on a particular IPC interface (such as BSD sockets or System V TLI). Inheriting from `Event_Handler` enables a `Logging_IO` object to be registered with the `Reactor`. This inheritance also allows a `Logging_IO` object's `handle_input` method to be dispatched automatically by a `Reactor` object to process logging records when they arrive from clients.

```
template <class XPORT_IO>
class Logging_IO :
  public Event_Handler, public XPORT_IO
{
public:
  // Callback method that handles the reception of
  // logging transmissions from remote clients.
  // Two recv()'s are used to maintain logging
  // record framing across a TCP bytestream.

  virtual int handle_input (HANDLE)
  {
    long len;
    long n;

    // Determine length of a logging record.
    n = this->XPORT_IO::recv (&len, sizeof len);

    if (n <= 0)
      return n;
    else {
      Log_Record log_record;

      // Convert from network to host byte-order.
      len = ntohl (len);
      this->XPORT_IO::recv (&log_record, len);

      // Format the logging record
      log_record.format ();
      // Print logging record to output device.
      log_record.print ();
      return 0;
    }
  }

  // Retrieve the underlying I/O descriptor (called
  // by the Reactor when a Logging_IO object is
  // first registered).

  virtual HANDLE get_handle (void) const
  {
    return this->XPORT_IO::get_handle ();
  }


  // Close down the I/O descriptor and delete
  // the object when a client closes down the
  // connection.

  virtual int handle_close (HANDLE, Reactor_Mask) {
    delete this;
    return 0;
  }

private:
  // Must be private to ensure dynamic allocation.
  ~Logging_IO (void) {
    this->XPORT_IO::close ();
  }
}
```

The `Accepter` template class is shown in the C++ code below. It is a generic factory that performs the steps necessary to (1) accept connection requests from client logging daemons and (2) create `CLIENT_HANDLER` objects that are used to perform an actual application-specific service on behalf of clients. Note that the `Accepter` object and the `CLIENT_HANDLER` objects it creates run within a single thread of control.

The `Accepter` subclass inherits from both the `XPORT_LISTENER` class and from the `Event_Handler` class. Inheriting from the `XPORT_LISTENER` class provides the capability to listen for connection requests on a communication port, as well as to accept connection requests on that port when they arrive from clients. Inheriting from the `Event_Handler` class enables an `Accepter` object to be registered with the `Reactor`. This inheritance also allows the `Reactor` to dispatch the `Accepter` object's `handle_input` method. In turn, the `handle_input` method invokes the `SOCK_Listener::accept` method to accept a new client's connection.

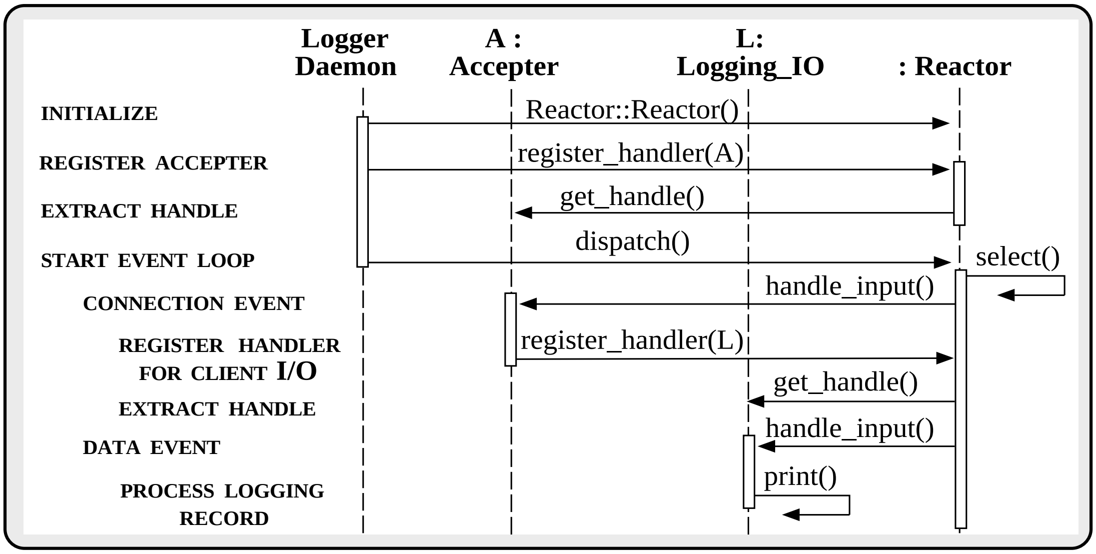

Figure 6: Server Logging Daemon Interaction Diagram

```
// Global per-process instance of the Reactor.
extern Reactor reactor;

// Handles connection requests from a remote client.

template <class CLIENT_HANDLER,
          class XPORT_LISTENER,
          class XPORT_ADDR>
class Accepter
  : public Event_Handler, public XPORT_LISTENER
{
public:

  // Initialize the accepter endpoint.

  Accepter (XPORT_ADDR &addr)
            : XPORT_LISTENER (addr) {}

  // Callback method that accepts a new connection,
  // creates a new CLIENT_HANDLER object to perform
  // I/O with the client connection, and registers
  // the new object with the Reactor.

  virtual int handle_input (HANDLE)
  {
    CLIENT_HANDLER *handler = new CLIENT_HANDLER;

    this->XPORT_LISTENER::accept (*handler);
    reactor.register_handler (handler, READ_MASK);
    return 0;
  }

  // Retrieve the underlying I/O descriptor (called
  // by the Reactor when a Accepter object is
  // first registered).

  virtual HANDLE get_handle (void) const {
    return this->XPORT_LISTENER::get_handle ();
  }

  // Close down the I/O descriptor when the
  // Accepter is shut down.

  virtual int handle_close (HANDLE, Reactor_Mask)
  {
    return this->XPORT_LISTENER::close ();
  }
};
```

The C++ code shown below illustrates the main entry point into the server logging daemon. This code creates a `Reactor` object and an `Accepter` object and registers the `Accepter` with the `Reactor`. Note that the `Accepter` template is instantiated with the `Logging_IO` class, which performs the distributed logging service on behalf of clients. Next, the main program enters the `Reactor`'s event-loop by calling `dispatch`. The `dispatch` method continuously handles connection requests and logging records that arrive from clients. The interaction diagram shown in Figure 6 illustrates the collaboration between the various objects in the server logging daemon at run-time. Note that once the `Reactor` object is initialized, it becomes the primary focus of the control flow within the server logging daemon. All subsequent activity is triggered by callback methods on the event handlers controlled by the `Reactor`.

```
// Global per-process instance of the Reactor.
Reactor reactor;

// Server port number.
const unsigned int PORT = 10000;

// Instantiate the Logging_IO template
typedef Logging_IO <SOCK_Stream> LOGGING_IO;

// Instantiate the Accepter template
typedef Accepter<LOGGING_IO,
                 SOCK_Listener,
                 INET_Addr> ACCEPTER;

int
main (void)
{
  INET_Addr addr (PORT);
  ACCEPTER accepter (addr);

  reactor.register_handler (&accepter,
                            READ_MASK);
```

```
  // Main event loop that handles client
  // logging records and connection requests.
  reactor.dispatch ();

  return 0;
}
```

The C++ code example shown above uses templates to decouple the reliance on the particular type of IPC interface used for connection establishment and communication. The `SOCK_Stream`, `SOCK_Listener` and `INET_Addr` classes used in the template instantiations are part of the `SOCK_SAP` C++ wrapper library [1]. `SOCK_SAP` encapsulates the `SOCK_STREAM` semantics of the socket transport layer interface within a type-secure, object-oriented interface. `SOCK_STREAM` sockets support the reliable transfer of bytestream data between two processes, which may run on the same or on different host machines in a network [16].

By using templates, it is relatively straightforward to instantiate a different IPC interface (such as the `TLI_SAP` C++ wrappers that encapsulate the System V TLI interface [22]). Templates trade additional compile-time and link-time overhead for improved run-time efficiency. Note that a similar degree of decoupling also could be achieved via inheritance and dynamic binding by using the Abstract Factory or Factory Method patterns described in [10]. [21] provides a detailed discussion of the trade-offs between templates and inheritance/dynamic binding.

3.2 Windows NT Implementation

This section describes the Windows NT implementation of the Reactor and Accepter design patterns. The Windows NT port was performed at the Ericsson/GE Mobile Communications facility in Cypress, California. The design patterns and framework described in this paper are currently being applied at Ericsson on a family of client/server applications as part of the External Operating Systems project [23]. This project uses the Reactor and Accepter patterns as the basis for a network management framework. This framework enhances the flexibility and reuse of applications that monitor and manage telecommunication switch performance across multiple hardware and software platforms.

We initially attempted to port the existing `Reactor` implementation from UNIX to Windows NT using the `select` function from the Windows Sockets (WinSock) library.[3] This approach failed because the WinSock version of `select` does not interoperate with standard Win32[4] I/O HANDLEs. Our applications required the use of Win32 I/O HANDLEs to support network protocols (such as Microsoft's NetBIOS Extended User Interface (NetBEUI)) that are not supported by WinSock version 1.1. Next, we tried to reimplement the `Reactor` interface using the Win32 API system call `WaitForMultipleObjects`. The goal was to maintain the original UNIX interface, but transparently supply a different implementation.

Transparent reimplementation failed to work due to fundamental differences in the proactive vs. reactive I/O semantics on Windows NT and UNIX outlined in Section 3. We initially considered circumventing these differences by using a technique that asynchronously initiated a 0-sized `ReadFile` request on an overlapped I/O HANDLE. Overlapped I/O is an Win32 mechanism that supports asynchronous input and output. With this technique, the overlapped event would signal when data arrives and a synchronous `ReadFile` would then be invoked to receive the data. Unfortunately, this solution would have doubled the number of system calls for every input operation, which caused unacceptable performance overhead. In addition, this approach still did not adequately emulate the output semantics provided by the UNIX reactive I/O mechanisms.

At this point it became clear that the direct reuse of class method interfaces, attributes, or algorithms was not a feasible type of reuse under the circumstances. Instead, we needed to elevate the level of abstraction for reuse to the level of design patterns. Regardless of the underlying OS event demultiplexing I/O semantics, the Reactor pattern is applicable for event-driven applications that must process multiple event handlers triggered concurrently by various types of events. Although differences between OS platforms precluded direct reuse of implementations or interfaces, the design knowledge we had invested in the Reactor and Accepter patterns was reusable.

The remainder of this section describes the modifications we made to the implementations of the Reactor and Accepter design patterns in order to port them to Windows NT.

3.2.1 Implementing the Reactor Pattern on Windows NT

Windows NT provides proactive I/O semantics that are typically used in the following manner. First, an application creates a HANDLE that corresponds to an I/O channel for the type of networking mechanism being used (such as named pipes or sockets). The overlapped I/O attribute is specified to the HANDLE creation system call (WinSock sockets are created for overlapped I/O by default). Next, an application creates a HANDLE to a Win32 event object and uses this event object HANDLE to initialize an overlapped I/O structure. The HANDLE to the I/O channel and the overlapped I/O structure are then passed to the `WriteFile` or `ReadFile` system calls when initiating either a send or receive operation, respectively. The initiated operation proceeds asynchronously and does not block the caller. When the operation completes, the event object specified inside the overlapped I/O structure is set to the "signaled" state. Subsequently, Win32 system calls such as `WaitForSingleObject` or `WaitForMultipleObjects` may be used to detect the signaled state of the Win32 event object, thereby determining when an outstanding asynchronous operation has completed.

The Win32 `WaitForMultipleObjects` system call

[3] WinSock is a Windows-oriented transport layer programming interface based on the BSD socket paradigm [24].

[4] Win32 is the 32-bit Windows subsystem of the Windows NT operating system.

is functionally similar to the UNIX select and poll system calls. It blocks on an array of HANDLEs waiting for one or more of them to signal. Unlike the two UNIX system calls (which wait only for I/O descriptors), WaitForMultipleObjects is a general purpose routine that may be used to wait for any type of Win32 object (such as a thread, process, synchronization object, I/O handle, named pipe, socket, or timer). It may be programmed to return to its caller either when any one of the HANDLEs becomes signaled or when all of the HANDLEs become signaled. WaitForMultipleObjects returns the index location in the HANDLE array of the lowest signaled HANDLE.

The generality of WaitForMultipleObjects is both a strength and a weakness. While it provides the flexibility to synchronize on a wide range Win32 objects, it is also more complicated to program for applications that must synchronize simultaneous send and receive operations on the same I/O channel. For example, in order to distinguish the completion of a send operation from a receive operation, separate overlapped I/O structures and Win32 event objects must be allocated for input and output. Furthermore, two elements in the WaitForMultipleObjects HANDLE array (which is currently limited to a rather small maximum of 64 HANDLEs) are consumed by the separate send and receive event object HANDLEs.

An advantage of the Windows NT proactive I/O scheme is that it may improve performance by allowing I/O operations to execute asynchronously with respect to other functions performed by the operating system. In contrast, the reactive I/O semantics offered by UNIX do not support asynchronous I/O directly (threads may be used instead). However, designing and implementing the Reactor pattern using proactive I/O on Windows NT turned out to be more difficult than using reactive I/O on UNIX.

Two characteristics of WaitForMultipleObjects significantly complicated the implementation of the Windows NT version of the Reactor pattern:

1. Each Win32 WaitForMultipleObjects call only returns notification on a single HANDLE. Therefore, to achieve the same behavior as the UNIX select and poll system calls (which return a set of descriptors), multiple WaitForMultipleObjects must be performed.
2. The semantics of WaitForMultipleObjects do not result in a fair distribution of notifications. In particular, the lowest signaled HANDLE in the array is always returned, regardless of how long other HANDLEs further back in the array may have been pending.

The implementation techniques required to handle these characteristics of Windows NT were rather complicated. Therefore, a Handler_Repository class was created to shield the Reactor from this complexity. This class provides a container for Event_Handler objects registered with a Reactor. This container class implements standard operations for inserting, deleting, suspending, and resuming Event_Handlers. Each Reactor object contains a Handler_Repository object in its private data portion. A Handler_Repository maintains the array of HANDLEs passed to WaitForMultipleObjects and it also provides methods for inserting, retrieving, and "re-prioritizing" the HANDLE array. Re-prioritization alleviates the inherent unfairness in the way that the Windows NT WaitForMultipleObjects system call notifies applications when HANDLEs become signaled.

The Handler_Repository's re-prioritization method is invoked by specifying the index of the HANDLE which has signaled and been dispatched by the Reactor. The method's algorithm moves the signaled HANDLE toward the end of the HANDLE array. This allows signaled HANDLEs that are further back in the array to be returned by subsequent calls to WaitForMultipleObjects. Over time, HANDLEs that signal frequently migrate to the end of the HANDLE array. Likewise, HANDLES that signal infrequently migrate to the front of the HANDLE array. This algorithm ensures a reasonably even distribution of HANDLE dispatching.

The implementation techniques described in the previous paragraph did not affect the external interface of the Reactor. Unfortunately, certain aspects of Windows NT proactive I/O semantics, coupled with the desire to fully utilize the flexibility of WaitForMultipleObjects, forced visible changes to the Reactor's external interface. In particular, Windows NT overlapped I/O operations must be initiated *immediately* (rather than waiting until it becomes *possible* to perform an operation, as with the UNIX reactive I/O scenario described above). Therefore, it was necessary for the Windows NT Event_Handler interface to distinguish between I/O HANDLEs and synchronization object HANDLES, as well as to supply additional information (such as message buffers and event HANDLEs) to the Reactor.

The following modifications to the Reactor were required to support Windows NT I/O semantics. The Reactor_Mask enumeration was modified to include a new SYNC_MASK value to allow the registration of an Event_Handler that is dispatched when a general Win32 synchronization object signals. The send method was added to the Reactor class to proactively initiate output operations on behalf of an Event_Handler.

```
// Bit-wise or these values to check for
// multiple activities per-descriptor
enum Reactor_Mask {
  READ_MASK = 01, WRITE_MASK = 02, SYNC_MASK = 04,
  RWS_MASK = READ_MASK | WRITE_MASK | SYNC_MASK
};

class Reactor
{
public:
  // Same as UNIX Reactor

  // Initiate an asynchronous send operation
  virtual int send (const Event_Handler *,
                    const Message_Block *);

// ...
};
```

Likewise, the Event_Handler interface for Windows NT was also modified, as follows:

```
class Event_Handler
{
protected:
  // Returns the Win32 I/O HANDLE associated with the
  // derived object (must be supplied by a subclass)
  virtual HANDLE get_io_handle (void) const;
  // Returns the Win32 synchronization HANDLE
  // associated with the derived object (must be
  // supplied by a subclass)
  virtual HANDLE get_sync_handle (void) const;

  // Called when object is removed from the Reactor
  virtual int handle_close (Message_Block *,
                            Reactor_Mask);
  // Called when input operation has completed
  virtual int handle_input (Message_Block *);
  // Called when output operation has completed
  virtual int handle_output (Message_Block *);
  // Called when a synchronization object has signaled
  virtual int handle_sync (void);

  // 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);

  // Allocates a message for the Reactor
  virtual Message_Block *get_message (void);

  // Get/set input and output events
  virtual HANDLE input_event (void);
  virtual void input_event (HANDLE in_event);
  virtual HANDLE output_event (void);
  virtual void output_event (HANDLE out_event);

// ...
};
```

When a derived `Event_Handler` is registered for input with the `Reactor` an overlapped input operation is immediately initiated on its behalf. In order to do this, the `Reactor` must obtain an I/O mechanism HANDLE, destination buffer, and a Win32 event object HANDLE for synchronization from the derived `Event_Handler`. A derived `Event_Handler` returns the I/O mechanism HANDLE via its `get_io_handle` method and returns the destination buffer location and length information via the `Message_Block` abstraction described in [7].

An event HANDLE for input synchronization is returned by the first overloaded `input_event` definition shown above. Since the creation of Win32 event objects is a common operation, the derived `Event_Handler` may choose to defer the operation to the `Reactor`. This is done by returning a NULL HANDLE from `input_event`. A NULL HANDLE signals the `Reactor` to allocate a Win32 event object for use with input operations for the derived `Event_Handler`. The allocated event object HANDLE is returned to the derived `Event_Handler` via the second overloaded `input_event` definition. The derived `Event_Handler` then assumes responsibility for properly closing the event object HANDLE when it is deleted. Each time an input operation completes and is successfully dispatched, the `Reactor` acquires a new `Message_Block` and proactively initiates the next input operation. The `output_event` methods are similar to the `input_event` methods, though they handle output semantics rather than input semantics.

When a derived `Event_Handler` object is registered for synchronization with the `Reactor` the object's `get_sync_handle` method is invoked automatically to obtain the Win32 synchronization object HANDLE. The synchronization object HANDLE is placed directly in the `WaitForMultipleObject` HANDLE array (in contrast, an I/O mechanism HANDLE is triggered indirectly via an event object HANDLE). Note that the `Reactor` performs no proactive operation that will cause the Win32 synchronization object to signal. Moreover, the `Reactor` does not perform any operation to reset or re-arm the synchronization object once it has signaled and been dispatched. The `Reactor` simply registers a synchronization object HANDLE and dispatches its derived `Event_Handler` when it signals.

Given the behavior of `Event_Handler` objects that are registered for synchronization, together with the semantics of the `WaitForMultipleObject` system call, one may question the need for specialized input and output processing within the `Reactor`. In other words, why not simply implement the Win32 version of the `Reactor` to handle only synchronization objects and push I/O handling functionality into the derived `Event_Handler`s? Our motivation for maintaining this distinction is that the `Reactor` is intended to perform I/O multiplexing within server applications. By encapsulating the details of initiating, completing, and dispatching I/O operations within the `Reactor`, derived `Event_Handler`s are able to reuse this functionality and to focus on data pre- and postprocessing (rather than focusing on I/O operation details).

The current implementation of the Windows NT-based Reactor pattern is about 1,600 lines C++ code (not including comments or extraneous whitespace). This code is approximately 200 lines longer than the UNIX version. The additional code is required primarily to handle the complex `WaitForMultipleObjects` event demultiplexing semantics discussed above. Although Windows NT event demultiplexing is more complex than UNIX, the behavior of Win32 mutex objects eliminated the need for the separate `Mutex` interface with recursive-mutex semantics discussed in Section 3.1.1. Under Win32, a thread will not be blocked if it attempts acquire a mutex specifying the HANDLE to a mutex that it already owns. However, to release its ownership, the thread must release a Win32 mutex once for each time that the mutex was acquired.

3.2.2 Implementing the Accepter Pattern on Windows NT

The following example C++ code illustrates an implementation of the Accepter pattern based on the Windows NT version of the Reactor pattern. The application is the same server logging daemon presented in Section 3.1.2. However, the example below uses a C++ wrapper for Win32 named pipes in place of the `SOCK_SAP` C++ wrappers for the socket interface.

```
template <class XPORT_IO>
```

```
class Logging_IO :
  public Event_Handler, public XPORT_IO
{
public:
  // Callback method that handles the reception of
  // logging transmissions from remote clients.  Note
  // the use of the Message_Block data structure, which
  // stores an incoming message received from a client.

  virtual int handle_input (Message_Block *msg)
  {
    Log_Record *log_record =
      (Log_Record *) msg->get_rd_ptr ();

      // Format record in preparation for printing.
      log_record.format ();
      // Print logging record to output device.
      log_record.print ();
      delete msg;
      return 0;
    }
  }

  // Retrieve the underlying I/O HANDLE (called
  // by the Reactor when a Logging_IO object is
  // first registered).

  virtual HANDLE get_handle (void) const
  {
    return this->XPORT_IO::get_handle ();
  }

  // Return a dynamically allocated buffer
  // to store an incoming logging message.

  virtual Message_Block *get_message (void) {
    return new Message_Block (sizeof (Log_Record));
  }

  // Close down the I/O descriptor and delete
  // the object when a client closes down the
  // connection.

  virtual int handle_close (Message_Block *msg,
                            Reactor_Mask) {
    delete msg;
    delete this;
    return 0;
  }

private:
  // Must be private to ensure dynamic allocation.
  ~Logging_IO (void) {
    this->XPORT_IO::close ();
  }
}
```

The `Accepter` class is essentially the same as the one illustrated in Section 3.1.2, though it uses the `handle_sync` method to complete connection acceptance rather than the `handle_input` method. Likewise, the interaction diagram that describes the collaboration between objects in the server logging daemon is also very similar to the one shown in Figure 6. The primary difference is that Win32 Named Pipe C++ wrappers are used in place of the socket C++ wrappers in the main program, as shown in the code below:

```
// Global per-process instance of the Reactor.
Reactor reactor;

// Server endpoint
const char ENDPOINT[] = "logger";

// Instantiate the Logging_IO template
typedef Logging_IO <NPipe_IO> LOGGING_IO;

// Instantiate the Accepter template
typedef Accepter<LOGGING_IO,
                 NPipe_Listener,
                 Local_Pipe_Name> ACCEPTER;

int
main (void)
{
  Local_Pipe_Name addr (ENDPOINT);
  ACCEPTER accepter (addr);

  reactor.register_handler (&accepter,
                            SYNC_MASK);

  accepter.initiate ();

  // Main event loop that handles client
  // logging records and connection requests
  reactor.dispatch ();

  return 0;
}
```

The named pipe Accepter object (`accepter`) is registered with the Reactor to handle asynchronous connection establishment. Due to the semantics of Windows NT proactive I/O, the `accepter` object must explicitly initiate the acceptance of a named pipe connection via its `initiate` method. Each time a connection acceptance is completed, the Reactor dispatches the `handle_sync` method of the named pipe Accepter to create a new `Client_Handler` that will receive logging records from the client. The `Reactor` will also initiate the next connection acceptance sequence asynchronously.

4 Concluding Remarks

Design patterns facilitate the reuse of an abstract architecture that is independent from any concrete realization of this architecture. Design patterns are particularly useful when developing system software components and frameworks that are reusable across OS platforms. This paper describes two design patterns (Reactor and Accepter) that are commonly used in distributed system software. These design patterns characterize the collaboration between objects that are used to automate common activities (such as event demultiplexing, event handler dispatching, and connection establishment) used to implement distributed systems.

This case study describes how a framework based on the Reactor and Accepter design patterns were ported from several UNIX platforms to the Windows NT Win32 platform. It was difficult to directly reuse the implementations, interfaces, or detailed designs of these frameworks across the different OS platforms. In particular, performance constraints and fundamental differences in the I/O mechanisms available on Windows NT and UNIX platforms prevented us from encapsulating event demultiplexing functionality within a completely reusable framework. However, we were able to reuse the underlying design patterns, which significantly reduced project risk.

Our experiences also underscore that the transition from object-oriented analysis to object-oriented design and implementation may be challenging. Often, the constraints of the underlying OS and hardware platform influence design and implementation details significantly. This is particularly problematic for system software, which is frequently targeted for particular platforms with particular non-portable characteristics. In such circumstances, reuse of design patterns may be the only viable means to leverage previous development expertise.

The UNIX version of the `ASX` framework components described in this paper are freely available via anonymous ftp from the Internet host `ics.uci.edu` (128.195.1.1) in the file `gnu/C++_wrappers.tar.Z`. This distribution contains complete source code, documentation, and example test drivers for the C++ components developed as part of the ADAPTIVE project [25] at the University of California, Irvine. Components in the `ASX` framework have been ported to both UNIX and Windows NT and are currently being used in a number of commercial products including the AT&T Q.port ATM signaling software product and the Ericsson EOS family of network management applications for telecommunication switches.

References

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

[2] R. Johnson and B. Foote, "Designing Reusable Classes," *Journal of Object-Oriented Programming*, vol. 1, pp. 22–35, June/July 1988.

[3] M. A. Linton, J. Vlissides, and P. Calder, "Composing User Interfaces with InterViews," *IEEE Computer*, vol. 22, pp. 8–22, February 1989.

[4] D. Batory and S. W. O'Malley, "The Design and Implementation of Hierarchical Software Systems Using Reusable Components," *ACM Transactions on Software Engineering and Methodology*, vol. 1, Oct. 1992.

[5] R. Campbell, V. Russo, and G. Johnson, "The Design of a Multiprocessor Operating System," in *Proceedings of the USENIX C++ Workshop*, pp. 109–126, USENIX Association, November 1987.

[6] J. M. Zweig, "The Conduit: a Communication Abstraction in C++," in *Proceedings of the* 2^{nd} *USENIX C++ Conference*, pp. 191–203, USENIX Association, April 1990.

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

[8] 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.

[9] D. C. Schmidt, "The Object-Oriented Design and Implementation of the Reactor: A C++ Wrapper for UNIX I/O Multiplexing (Part 2 of 2)," *C++ Report*, vol. 5, September 1993.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1994.

[11] S. Vinoski, "Distributed Object Computing with CORBA," *C++ Report*, vol. 5, July/August 1993.

[12] A. Weinand, E. Gamma, and R. Marty, "ET++ - an object-oriented application framework in C++," in *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference*, pp. 46–57, ACM, Sept. 1988.

[13] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Dispatching," in *Proceedings of the* 1^{st} *Annual Conference on the Pattern Languages of Programs*, (Monticello, Illinois), August 1994.

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

[15] D. C. Schmidt and T. Suda, "The Service Configurator Framework: An Extensible Architecture for Dynamically Configuring Concurrent, Multi-Service Network Daemons," in *Proceedings of the Second International Workshop on Configurable Distributed Systems*, (Pittsburgh, PA), pp. 190–201, IEEE, Mar. 1994.

[16] W. R. Stevens, *UNIX Network Programming*. Englewood Cliffs, NJ: Prentice Hall, 1990.

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

[18] S. Rago, *UNIX System V Network Programming*. Reading, MA: Addison-Wesley, 1993.

[19] J. Eykholt, S. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams, "Beyond Multiprocessing... Multithreading the SunOS Kernel," in *Proceedings of the Summer USENIX Conference*, (San Antonio, Texas), June 1992.

[20] A. D. Birrell, "An Introduction to Programming with Threads," Tech. Rep. SRC-035, Digital Equipment Corporation, January 1989.

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

[22] D. C. Schmidt, "The ADAPTIVE Communication Environment: Object-Oriented Network Programming Components for Developing Client/Server Applications," in *Proceedings of the* 12^{th} *Annual Sun Users Group Conference*, (San Francisco, CA), pp. 214–225, SUG, June 1994.

[23] D. C. Schmidt and P. Stephenson, "An Object-Oriented Framework for Developing Network Server Daemons," in *Proceedings of the* 2^{nd} *C++ World Conference*, (Dallas, Texas), SIGS, Oct. 1993.

[24] *Windows Sockets - An Open Interface for Network Programming under Microsoft Windows*, Version 1.1 ed., January 1993.

[25] D. C. Schmidt, D. F. Box, and T. Suda, "ADAPTIVE: A Dynamically Assembled Protocol Transformation, Integration, and eValuation Environment," *Journal of Concurrency: Practice and Experience*, vol. 5, pp. 269–286, June 1993.