The Reactor

An Object-Oriented Framework for Event Demultiplexing and Event Handler Dispatching

Douglas C. Schmidt

Overview

- The Reactor is an object-oriented framework that encapsulates OS event demultiplexing mechanisms
  - *e.g.*, the Reactor API runs transparently atop both `WaitForMultipleObjects` and `select`
- The Reactor simplifies the development of network servers that handle different types of I/O from multiple sources
  - It handles many low-level details involved with event demultiplexing and event handler dispatching
  - When events occur, the Reactor automatically dispatches previously-registered "call-back" member functions that perform application-specific services

Overview (cont'd)

- The Reactor framework integrates a collection of extensible, reusable, and type-secure C++ classes that address many limitations with existing UNIX and Win32 event demultiplexing facilities, while preserving the benefits they offer
  - *e.g.*, enables development of distributed applications (particularly servers) using higher-level OO design and programming abstractions *without* sacrificing efficiency
- The Reactor interoperates seamlessly with both
  1. I/O handles (*e.g.*, sockets)
  2. ACE IPC_SAP communication objects
- Originally inspired by the Dispatcher facility in the InterViews OO window system release

Reactor Features

- *Added Functionality*
  - Combines synchronous I/O events together with timer-based events and asynchronous signals
- *Improve Portability*
  1. Application portability
     - Shield applications from differences between `select` and `WaitForMultipleObjects`
  2. Reactor portability
     - The modularity of the internal OO structure simplifies porting effort between UNIX and Win32

OO Design Overview

- The Reactor's object-oriented design is based upon domain analysis of typical client/server event demultiplexing structures and functionality
- A primary design goal is to decouple (1) the *sensing*, *demultiplexing*, and *dispatching* of the I/O-based and timer-based events from (2) the actual application-specific service being performed
- This decomposition forms the basis for more advanced OO daemon configuration, port demultiplexing, and service dispatching frameworks
  - *e.g.*, The ACE ORB (TAO) and JAWS

OO Design Overview (cont'd)

- Once the key abstractions were identified the next steps involved:
  1. Encapsulating design decisions behind abstract, extensible APIs
  2. Localizing platform dependencies in well-defined classes
  3. Implementing the design in a language (such as C++) that supports the Reactor's functionality cleanly and efficiently
     - *e.g.*, object-oriented, rather than function-oriented call-backs

Reactor Features (cont'd)

- *Simplify Network Programming*
  1. Export uniform OO interfaces
     - Based on inheritance and dynamic binding
  2. Automate dispatching of application-specific member functions
     - Also enables object-based "context retention" for stateful services
  3. Support transparent extensibility
     - *e.g.*, separate application-specific *policies* from general-purpose Reactor *mechanisms*
  4. Increase reuse
     - *i.e.*, demultiplexing and dispatching code
  5. Enhance type-security
     - *e.g.*, hide I/O handles

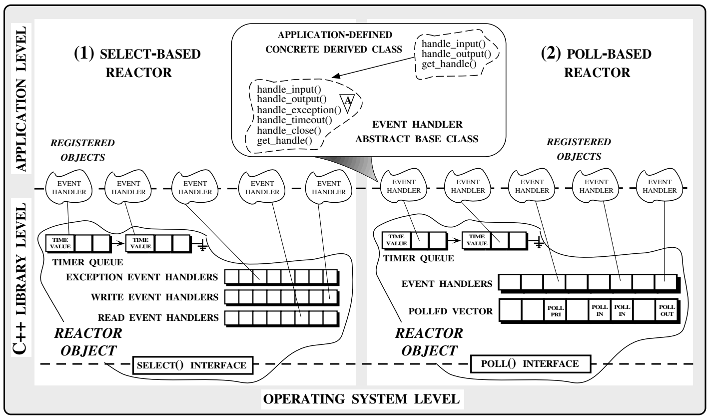
OO Design Overview (cont'd)

- Key abstractions in event-driven applications
  1. *Event sensors*
     - Detect and retrieve events from various I/O ports
     - Timer queue
  2. *Event loop*
     - Central control flow driver that structures the application
  3. *Event dispatchers*
     - Demultiplex each event to appropriate pre-registered handlers
  4. *Event handler*
     - Application-supplied code that processes incoming events and subsequently returns control to the framework's event loop

OO Design Overview (cont'd)

- *Platform Dependent* classes in the Reactor include:
  - ACE_Reactor
    - The main API for the C++ wrapper library
    - Uses the Bridge pattern to encapsulate `select` and `WaitForMultipleObjects`
    - Supplies synchronous event-driven I/O demultiplexing
    - Also handles timer-based events
    - The implementation of the `ACE_Reactor` class is structually similar across platforms...
      - *e.g.*, OO techniques are used both externally *and* internally
- Note, end-users directly interact with the `ACE_Reactor` and subclasses of `ACE_Event_Handler`

OO Design Overview (cont'd)

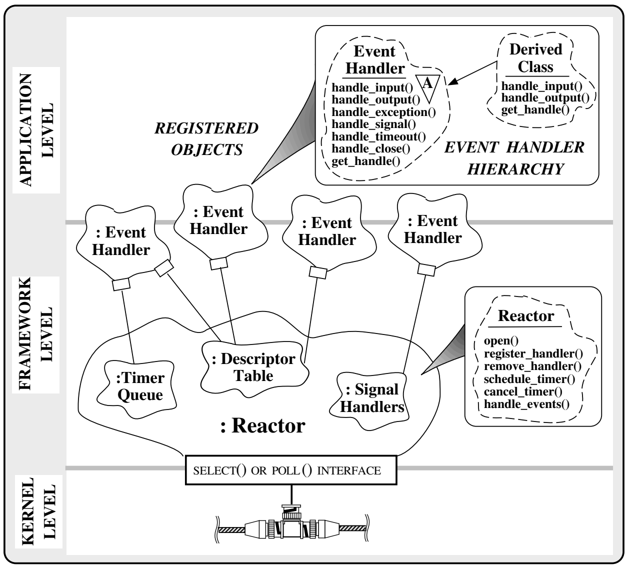


- Internal data structures used for the `select`-based versions of the Reactor

OO Design Overview (cont'd)

- *Platform Independent* classes in the Reactor include:
  - ACE_Time_Value
    - A set of routines for manipulating *normalized* `struct timeval` quantities
  - ACE_Timer_Queue
    - Provides an interface for scheduling, canceling, and expiring periodic and aperiodic time-based events
  - ACE_Event_Handler
    - Abstract base class that specifies a well-defined dispatching interface used by the Reactor to provide an extensible call-back facility
    - Application developers derive subclasses from `ACE_Event_Handler`, which are typically:
      1. Bound to I/O handles
      2. Registered with an instance of the `ACE_Reactor`

OO Design Overview (cont'd)



ACE_Time_Value Class

- A C++ wrapper around the underlying OS timer routines
  - *e.g.*, the UNIX `timeval` structure

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

- ACE_Time_Value provides routines to manipulate "normalized" time quantities
  - *e.g.*, normalization ensures that

    ACE_Time_Value (1, 1000000) == ACE_Time_Value (2)

- Simplifies time-based comparisons within the Reactor
  - *i.e.*, forms the basis for "timer-based event dispatching"

ACE_Time_Value Class (cont'd)

- Class interface

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

  friend ACE_Time_Value operator + (const ACE_Time_Value &,
                                    const ACE_Time_Value &);
  // Returns the sum of two ACE_Time_Values,
  friend ACE_Time_Value operator - (const ACE_Time_Value &,
                                    const ACE_Time_Value &);
  // Returns the difference between two ACE_Time_Values.

  // = Relational and comparison operators for normalized
  //   ACE_Time_Values.
  friend int operator < (const ACE_Time_Value &,
                         const ACE_Time_Value &);
  friend int operator > (....);
  friend int operator <= (....);
  friend int operator >= (....);
  friend int operator == (....);
  friend int operator != (....);
private:
  /* ... */
};
```


ACE_Timer_Queue Class

- Provides services for scheduling, canceling, and expiring periodic and aperiodic time-based events
  - *e.g.*, to detect timeouts and send retransmissions

- Default implementation is a sorted list of `ACE_Time_Value` and `ACE_Event_Handler` tuples
  - Time is stored in "absolute" time units

- Since all member functions are declared **virtual** it is trivial to use alternative timer-based dispatching schemes
  - *e.g.*, *delta-lists*, *heaps*, or *timing wheels*

ACE_Timer_Queue Class (cont'd)

- Class interface:

```
class ACE_Timer_Queue
{
public:
  int is_empty (void) const;
  // True if queue is empty, else false.
  const ACE_Time_Value &earliest_time (void) const;
  // Returns earliest time in queue.
  static const ACE_Time_Value &current_time (void);
  // Returns current time of day.

  void virtual schedule (const ACE_Event_Handler *handler,
                         const void *arg,
                         const ACE_Time_Value &future_time,
                         const ACE_Time_Value &interval);
  // Insert a new handler that expires at a future_time.
  // If interval is > 0, the handler will be reinvoked
  // periodically until explicitly canceled.

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

  void virtual expire (const ACE_Time_Value &);
  // Expire all Timers whose values are <= expire_time.
  // Note, this must be called manually (i.e., it is
  // non-preemptive).
private:
  // ...
};
```

ACE_Event_Handler Class

- Abstract base class that specifies an extensible interface used by portions of the `ACE_Reactor` class that control and coordinate the automatic dispatching of I/O and timer mechanisms
- The Reactor uses application-defined subclasses of the `ACE_Event_Handler` to implement its automated event-driven call-back services
- Each subclass may redefine certain virtual member functions that perform application-defined actions in response to events such as:
  1. Different types of I/O events on handles
     - *e.g.*, *reading*, *writing*, and *exceptions*
  2. Timer-expiration

ACE_Event_Handler Class (cont'd)

- Class interface:

```
typedef int ACE_Reactor_Mask;
class ACE_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 handle.
  enum {
    READ_MASK   = 01,
    WRITE_MASK  = 02,
    EXCEPT_MASK = 04,
    RWE_MASK    = READ_MASK | WRITE_MASK | EXCEPT_MASK
  };

  virtual ACE_HANDLE get_handle (void) const = 0;
  // Returns the I/O handle associated with
  // the derived object
```



```
  virtual int handle_close (ACE_HANDLE handle,
                            ACE_Reactor_Mask mask);
  // Called when object is removed from the Reactor.

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

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

  virtual int handle_exception (ACE_HANDLE handle);
  // Called when urgent data is available on handle.

  virtual int handle_timeout (const ACE_Time_Value &tv,
                              const void *arg = 0);
  // Called when timer expires (tv stores the
  // current time).
};
```


ACE_Event_Handler Class (cont'd)

- An `ACE_Event_Handler` subclass typically has an I/O handle associated with the derived object
  - Unless it is purely time-based...
- A return value of -1 from an `ACE_Event_Handler` member function in a subclass instructs the `ACE_Reactor` to remove the associated derived class object from its internal tables
- When an object is removed, its `handle_close` member function is automatically invoked
  - This function typically performs any user-defined termination activities
    - *e.g.*, delete dynamic memory, flush logging files, close communication end-points, etc.

ACE_Reactor Class

- A C++ wrapper around `select` and `poll`
- The `ACE_Reactor` class forms the basis for "event-driven" application service processing
  - *i.e.*, it waits an application-specified time interval for different types of events to occur on multiple I/O handles
    - These handles are associated with registered objects derived from `ACE_Event_Handler`
- The `ACE_Reactor` supports
  1. Synchronous I/O events
  2. Timer-based events

ACE_Reactor Class (cont'd)

- Objects registered in the `ACE_Reactor` are easily extended via subclasses derived from the `ACE_Event_Handler` abstract base class
- Typically, there is only one active instance of a `ACE_Reactor` per-process or per-thread
- However, other `ACE_Reactor` "configurations" may be stored in variables and swapped in and out as necessary

ACE_Reactor Class (cont'd)

- Class interface:

```
class ACE_Reactor
{
public:
  virtual int register_handler (const ACE_Event_Handler *,
                                ACE_Reactor_Mask =
                                  ACE_Event_Handler::READ_MASK);
  // Registers an ACE_Event_Handler object according to the
  // Reactor_Mask (i.e., "reading," "writing," and/or
  // "exceptions").

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



```
virtual void schedule_timer (const ACE_Event_Handler *,
                             const void *arg,
                             const ACE_Time_Value &delta,
                             const ACE_Time_Value &interval);
// Register a handler to expire at time delta.  When
// delta expires the handle_timeout() member function
// 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 cancel_timer (const ACE_Event_Handler *handler);
// Locate and cancel timer associated with handler.

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

// ...
```

Reactor Timer-based Example

- Simple program that periodically prints "hello world" until user inputs data.

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

  virtual int handle_input (ACE_HANDLE handle) {
    char buf[BUFSIZ];
    ssize_t n = read (handle, buf, sizeof buf);
    write (STDOUT, buf, n);
    return -1;
  }

  virtual ACE_HANDLE get_handle (void) const {
    return STDIN;
  }

  virtual int handle_close (ACE_HANDLE,
                            ACE_Reactor_Mask)
  {
    // ...
    exit (1);
  }
};
```



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

  ACE_Reactor reactor;
  Hello_World handler; // timer object.
  ACE_Time_Value delay = ::atoi (argv[1]);
  ACE_Time_Value interval = ::atoi (argv[2]);

  reactor.register_handler (&handler,
                            ACE_Event_Handler::READ_MASK);
  reactor.schedule_timer (&handler, 0,
                          delay, interval);

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


Summary

- The Reactor framework decouples application-specific policies from a suite of reusable event demultiplexing and service dispatching mechanisms
  - This improves reusability and extensibility
- Moreover, the Reactor's internal OO design makes it much easier to extend and port to other systems
  - Note, applications also become more portable
    - *e.g.*, no changes to move from UNIX to Win32
- The Reactor also forms the basis for the ACE ORB (TAO) and JAWS