A Portable, Extensible and Efficient Implementation of Proactor Pattern

Alexander Babu Arulanthu, Irfan Pyarali and Douglas C. Schmidt

{alex, irfan, schmidt}@cs.wustl.edu

http://www.cs.wustl.edu/{∼alex, irfan, schmidt}

Department of Computer Science

Washington University, St. Louis 63130

Abstract

The Proactor pattern [1] describes how to structure applications and systems that effectively utilize asynchronous mechanisms supported by operating systems. When an application invokes an asynchronous operation, the OS performs the operation on behalf of the application. This allows the application to have multiple operations running simultaneously without requiring the application to have a corresponding number of threads. Therefore, the Proactor pattern simplifies concurrent programming and improves performance by requiring fewer threads and leveraging OS support for asynchronous operations.

The Adaptive Communications Environment (ACE) [2] has implemented a Proactor framework that encapsulates I/O Completion Ports of Windows NT operating system. This ACE Proactor abstraction provides an OO interface to the standard C APIs supported by Windows NT. We ported this Proactor framework to Unix platforms that support POSIX4 asynchronous I/O calls and real-time signals. This paper describes the design and implementation of this new `Portable Proactor` *framework and explains how the design and the implementation have been made so that the framework can be extensible, scalable and efficient. We explain how our design took care of keeping the old interfaces of the framework intact, still making the design highly extensible and efficient. The source code for this implementation can be acquired from the ACE website at* `www.cs.wustl.edu/∼schmidt/ACE.html`.

1 The Proactor Pattern

1.1 Intent

The Proactor pattern presented in [1] supports the demultiplexing and dispatching of multiple event handlers, which are triggered by the *completion* of asynchronous events. This pattern simplifies asynchronous application development by integrating the demultiplexing of completion events and the dispatching of their corresponding event handlers.

1.2 Motivation

The Proactor pattern should be applied when applications require the performance benefits of executing operations concurrently, without the constraints of synchronous multi-threaded or reactive programming. To illustrate these benefits, consider a networking application that needs to perform multiple operations concurrently. For example, a high-performance Web server must concurrently process HTTP requests sent from multiple clients. Figure 1 shows a typical interaction between Web browsers and a Web server. When a user instructs a

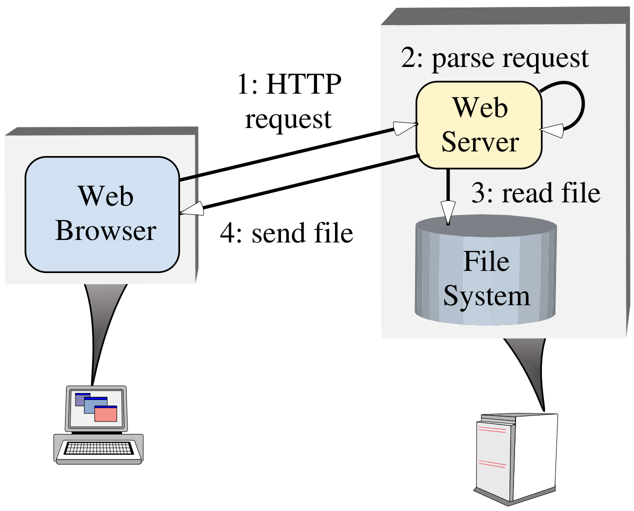

Figure 1: Typical Web Server Communication Software Architecture

browser to open a URL, the browser sends an HTTP `GET` request to the Web server. Upon receipt, the server parses and validates the request and sends the specified file(s) back to the browser.

Developing high-performance Web servers requires the resolution of the following forces:

- *Concurrency* – The server must perform multiple client requests simultaneously;
- *Efficiency* – The server must minimize latency, maximize throughput, and avoid utilizing the CPU(s) unnecessarily.
- *Programming simplicity* – The design of the server should simplify the use of efficient concurrency strategies;
- *Adaptability* – Integrating new or improved transport protocols (such as HTTP 1.1 [3]) should incur minimal maintenance costs.

A Web server can be implemented using several concurrency strategies, such as multiple synchronous threads and reactive synchronous event dispatching. But such conventional approaches have drawbacks as discussed in [1]. The Proactor pattern provides a powerful technique that supports an efficient and flexible asynchronous event dispatching strategy for high-performance concurrent applications.

1.3 Concurrency Through Proactive Operations

When the OS platform supports asynchronous operations, an efficient and convenient way to implement a high-performance Web server is to use *proactive event dispatching*. Web servers designed using a proactive event dispatching model handle the *completion* of asynchronous operations with one or more threads of control. Thus, the Proactor pattern *simplifies asynchronous Web servers by integrating completion event demultiplexing and event handler dispatching*.

An asynchronous Web server would utilize the Proactor pattern by first having the Web server issue an asynchronous operation to the OS and registering a callback with a `Completion Dispatcher` that will notify the Web server when the operation completes. The OS performs the operation on behalf of the Web server and subsequently queues the result in a well-known location. The `Completion Dispatcher` is responsible for de-queuing completion notifications and executing the appropriate callback that contains application-specific Web server code.

Figures 2 and 3 show how a Web server designed using proactive event dispatching handles multiple clients concurrently within one or more threads. Figure 2 shows the sequence of steps taken when a client connects to the Web Server.

1. The Web Server instructs the `Acceptor` to initiate an asynchronous accept;

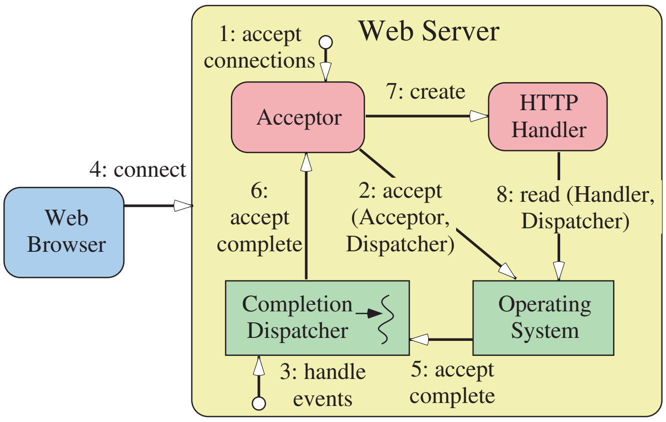

Figure 2: Client connects to a Proactor-based Web Server

2. The Acceptor initiates an asynchronous accept with the OS and passes itself as a `Completion Handler` and a reference to the `Completion Dispatcher` that will be used to notify the `Acceptor` upon completion of the asynchronous accept;
3. The Web Server invokes the event loop of the `Completion Dispatcher`;
4. The client connects to the Web Server;
5. When the asynchronous accept operation completes, the `Operating System` notifies the `Completion Dispatcher`;
6. The `Completion Dispatcher` notifies the Acceptor;
7. The `Acceptor` creates an `HTTP Handler`;
8. The `HTTP Handler` initiates an asynchronous operation to read the request data from the client and passes itself as a `Completion Handler` and a reference to the `Completion Dispatcher` that will be used to notify the `HTTP Handler` upon completion of the asynchronous read.

Figure 3 shows the sequence of steps that the proactive Web Server takes to service an HTTP `GET` request. These steps are explained below:

1. The client sends an HTTP `GET` request;
2. The read operation completes and the `Operating System` notifies the `Completion Dispatcher`;
3. The `Completion Dispatcher` notifies the `HTTP Handler` (steps 2 and 3 will repeat until the entire request has been received);
4. The `HTTP Handler` parses the request;

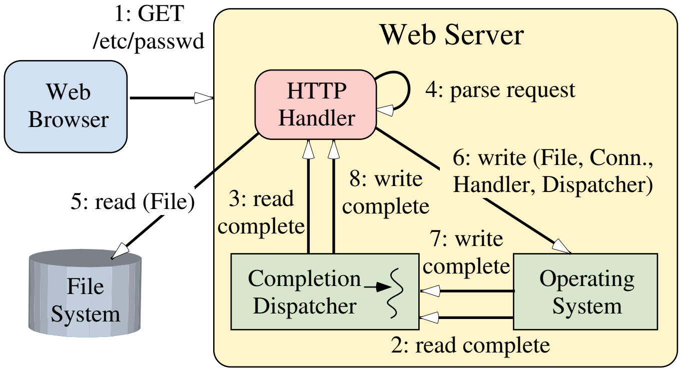

Figure 3: Client Sends requests to a Proactor-based Web Server

5. The `HTTP Handler` synchronously reads the requested file;
6. The `HTTP Handler` initiates an asynchronous operation to write the file data to the client connection and passes itself as a `Completion Handler` and a reference to the `Completion Dispatcher` that will be used to notify the `HTTP Handler` upon completion of the asynchronous write;
7. When the write operation completes, the Operating System notifies the `Completion Dispatcher`;
8. The `Completion Dispatcher` then notifies the `Completion Handler` (steps 6-8 continue until the file has been delivered completely).

The primary advantage of using the Proactor pattern is that multiple concurrent operations can be started and can run in parallel without necessarily requiring the application to have multiple threads. The operations are started asynchronously by the application and they run to completion within the I/O subsystem of the OS. The thread that initiated the operation is now available to service additional requests.

In the example above, for instance, the `Completion Dispatcher` could be single-threaded. When HTTP requests arrive, the single `Completion Dispatcher` thread parses the request, reads the file, and sends the response to the client. Since the response is sent asynchronously, multiple responses could potentially be sent simultaneously. Moreover, the synchronous file read could be replaced with an asynchronous file read to further increase the potential for concurrency. If the file read is performed asynchronously, the only synchronous operation performed by an `HTTP Handler` is the HTTP protocol request parsing.

The primary drawback with the Proactive model is that the programming logic is at least as complicated as the Reactive model. Moreover, the Proactor pattern can be difficult to debug since asynchronous operations often have a non-predictable and non-repeatable execution sequence, which complicates analysis and debugging. But Patterns such as the Asynchronous Completion Token [4] can be applied to simplify the asynchronous application programming model [1].

1.4 Applicability

The Proactor pattern is used when one or more of the following conditions hold:

- An application needs to perform one or more asynchronous operations without blocking the calling thread;
- The application must be notified when asynchronous operations *complete*;
- The application needs to vary its concurrency strategy independent of its I/O model;
- The application will benefit by decoupling the application-dependent logic from the application-independent infrastructure;
- An application will perform poorly or fail to meet its performance requirements when utilizing either the multi-threaded approach or the reactive dispatching approach.

1.5 Structure and Participants

The structure of the Proactor pattern is illustrated in Figure 4 using UML notation.

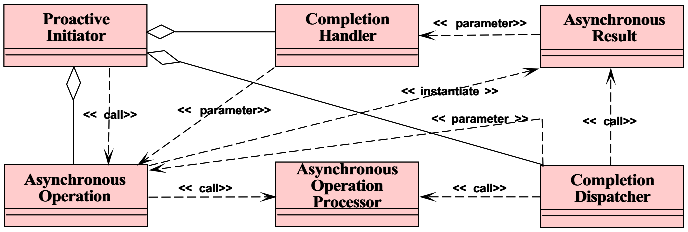

Figure 4: Participants in the Proactor Pattern

The key participants in the Proactor pattern include the following:

Proactive Initiator (`Web server application's main thread`):

- A `Proactive Initiator` is any entity in the application that initiates an `Asynchronous Operation`. The `Proactive Initiator` registers a `Completion Handler` and a `Completion Dispatcher` with an `Asynchronous Operation Processor`, which notifies it when the operation completes.

Completion Handler (the `Acceptor` and `HTTP Handler`):

- The Proactor pattern uses `Completion Handler` interfaces that are implemented by the application for `Asynchronous Operation` completion notification.

Asynchronous Operation (the methods `Asynchronous Read`, `Asynchronous Write`, `Asynchronous Accept` and `Asynchronous Transmit File`):

- `Asynchronous Operations` are used to execute requests (such as I/O and timer operations) on behalf of applications. When applications invoke `Asynchronous Operations`, the operations are performed *without* borrowing the application's thread of control.[1] Therefore, from the application's perspective, the operations are performed *asynchronously*. When `Asynchronous Operations` complete, the `Asynchronous Operation Processor` delegates application notifications to a `Completion Dispatcher`.

Asynchronous Operation Processor (the `Operating System`):

- `Asynchronous Operations` are run to completion by the `Asynchronous Operation Processor`. This component is typically implemented by the OS.

Asynchronous Result (the object passed to the `Completion Handler` on *completion* of an `Asynchronous Operation`)

- For each `Asynchronous Operation` class, there is one `Asynchronous Result` class. The `Asynchronous Operation` classes use the `Asynchronous Result` classes to specify all the parameters needed to carry out the Asynchronous Operations. The `Asynchronous Result` objects are created by the `Asynchronous Operation` classes and are passed to the `Asynchronous Operation Processor` on issuing the asynchronous calls. The `Asynchronous Result` objects also contain the results of the asynchronous operations and are used to pass the results to the `Completion Handlers`. In addition, the `Asynchronous Result` objects have information such as `Asynchronous Completion Tokens ACTs` [4] which can be used by the applications to uniquely associate the asynchronous method completions with their invocations.

[1] In contrast, the reactive event dispatching model [5] steals the application's thread of control to perform the operation synchronously.

Completion Dispatcher (the `Notification Queue`)

- The `Completion Dispatcher` is responsible for calling back to the application's `Completion Handlers` when `Asynchronous Operations` complete. When the `Asynchronous Operation Processor` completes an asynchronously initiated operation, the `Completion Dispatcher` performs an application callback on behalf of the `Asynchronous Operation Processor`. The `Completion Dispatcher` fills the result of the asynchronous operation in the `Asynchronous Result` object and passes that to the `Completion Handler`.

2 WIN32 Implementation of the Proactor Pattern

In this section, we will discuss the design of the Proactor framework that was built only for the WIN32 I/O Completion Ports. We will discuss how each of the participant of the pattern discussed in the previous section was implemented.

2.1 Asynchronous Operation Processor

The WIN32 I/O subsystem is the *Asynchronous Operation Processor*.

It defines `OVERLAPPED` structure which contains information used in asynchronous input and output (I/O). This structure is passed to the asynchronous APIs to specify the address of the data, such as `Offset` etc, in a file.

The WIN32 I/O subsystem provides the following APIs to execute asynchronous I/O calls.

- *ReadFile:* To issue asynchronous read on a stream or a file handle.
- *WriteFile:* To issue asynchronous accept on a stream or a file handle.
- *AcceptEx:* To issue asynchronous accept from a socket handle.
- *TransmitFile:* To initiate transmitting a file asynchronously.
- *CancelIO:* To cancel all pending input and output (I/O) operations that were issued by the calling thread for the specified file handle.
- *GetCompletionStatus:* To query the OS for completions of asynchronous I/O.
- *PostQueuedCompletionStatus:* To post a completion to a `Completion Port`

2.2 Asynchronous Operation

`Asynchronous Read Stream`, `Asynchronous Write Stream`, `Asynchronous Read File`, `Asynchronous Write File`, `Asynchronous Accept` and `Asynchronous Transmit File` are the different asynchronous operations that are currently supported in the framework. The UML diagram 5 shows the family of classes which implement the various `Asynchronous Operations`.

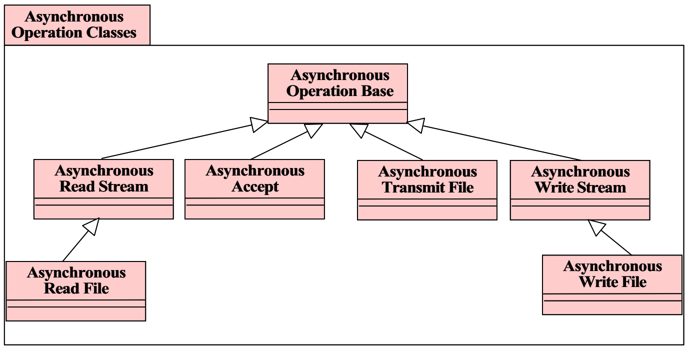

Figure 5: Asynchronous Operation Classes

2.2.1 Asynchronous Operation Base

The class `Asynchronous Operation Base` abstracts out all the common code found in the individual asynchronous operation classes. This class provides the following APIs.

- `open`: Applications use the `open` method to register the `I/O handle` and the `Completion Handler` with the `Completion Dispatcher`. This API calls the `CreateIoCompletionPort` API of the WIN32 operating system to register the `I/O handle` with the `Completion Port` of the I/O subsystem.

- `cancel`: This function cancels all the asynchronous calls issued on a completion port on a particular I/O handle. This is implemented using the `CancelIO`.

2.2.2 Asynchronous Read Stream

After the `open` method of the `Asynchronous Operation Base` is called, applications call the *read* API of this class to issue an asynchronous read on a stream. The `handle` to be read from is given via the `open` call. `read` API takes the following parameters

- `Handler:` The completion handler to be called when the operation is completed. This `Handler` defines the call back method `handle_read_stream`.
- `message block:` Buffer where the data has to be read.
- `bytes to read:` Maximum number of bytes that could be read from the socket.
- `ACT (Asynchronous Completion Token):` Magic cookie to identify the asynchronous read invocation when it completes.

The following steps take place when `read` API is called.

- `read` creates an `Asynchronous Read Stream Result` object with all the information that are needed to carry out the asynchronous operation such as `I/O handle` and `bytes to read` and also the information needed to call back the application such as `Handler` and `ACT`.
- `read` passes this result object to a worker method called `shared_read`. This worker method is shared by the `Asynchronous Read File` operation class.
- `shared_read` calls the `ReadFile` WIN32 API to issue the asynchronous read. `ReadFile` takes pointer to the *OVERLAPPED* structure. Since we want to preserve more information such as `ACT` and `Handler` along with `OVERLAPPED` structure, we pass the pointer to `Asynchronous Read Stream Result` object to the `ReadFile` call. To achieve this, all the Asynchronous Result classes derive from the `OVERLAPPED` structure (refer to Figure 6).

2.2.3 Asynchronous Write Stream

After the `open` method of the `Asynchronous Operation Base` is called, applications should call the *write* API of this class to issue an asynchronous write on a stream. The `handle` to write to is already given via the `open` call. The `write` API takes the following parameters

- `Handler:` The completion handler to be called when the operation is completed. This `Handler` defines the call back method `handle_write_stream`.
- `message block:` Buffer that contains the data to be sent.
- `bytes to write:` Number of bytes to be written to the socket.
- `ACT:` Magic cookie for this asynchronous write invocation.

The following steps take place when `write` is called.

- `write` creates an `Asynchronous Write Stream Result` object with all the information that are needed to carry out the asynchronous operation such as I/O handle and bytes to write and also the information needed to call back the application such as Handler and ACT.

- write passes this result object to a worker method called shared_write. This worker method is shared by the Asynchronous Write File Operation class.
- shared_write calls the *WriteFile* WIN32 API to issue the asynchronous write. WriteFile takes pointer to the *OVERLAPPED* structure. As discussed in Asynchronous Read Stream operation, we pass the pointer to *Asynchronous Write Stream Result* object to the WriteFile so that additional information such as ACT, Handler can be passed around along with the OVERLAPPED structure.

2.2.4 Asynchronous Read File

This class extends the functionality of the Asynchronous Read Stream class to do asynchronous read on a file. This operation is very similar to Asynchronous Read Stream except that it reads from a file handle instead of steam handle. The handle to be read from is given via the open call. The Handler's handle_read_file is called on completion. When read is called, a Asynchronous Read File Result class object is created and passed to the shared_read method of the Asynchronous Read Stream.

2.2.5 Asynchronous Write File

This class extends the functionality of the Asynchronous Write Stream class to do asynchronous write to a file. This operation class is very similar to the Asynchronous Write Stream except that it writes from a file handle instead of steam handle. The stream handle is given via the open call. The Handler's *handle_write_file* is called on completion. When write is called, a Asynchronous Write File Result class object is created and passed to the shared_read method of the Asynchronous Write Stream.

2.2.6 Asynchronous Accept

After the open method of the Asynchronous Operation Base is called, applications call the *accept* API of this class to issue an asynchronous write on a stream. The listen handle where the connection is to be accepted is given via the open call.The accept API takes the following parameters:

- Handler: This class defines call back hook method handle_asynch_accept which gets called when the accept completes.
- message block: Buffer to accept initial data that is read on the socket.
- bytes to read: Number of initial bytes to be read from the socket.
- ACT: Magic cookie for this asynchronous accept invocation.

The following steps take place when accept is called.

- accept creates an Asynchronous Accept Result object with all the information that are needed to carry out the asynchronous operation which are I/O handle and number of bytes to read and also the information needed to call back the application which are Handler and ACT.
- accept calls the *AcceptEx* WIN32 API to issue the asynchronous accept. AcceptEx takes pointer to the OVERLAPPED structure. Since we want to preserve the more information such as ACT and Handler along with OVERLAPPED structure, we pass the pointer to Asynchronous Accept Result object to the AcceptEx call.

2.2.7 Asynchronous Transmit File

After the open method of the Asynchronous Operation Base is called, applications should call the *transmit_file* API of this class to issue an asynchronous transmit file operation. The stream handle is given via the open call.transmit file API takes the following parameters

- Handler: This class defines call back hook method handle_asynch_transmit_file which gets called when the transmission completes.
- header and trailer: Header and the trailer data for the transmission.
- The offset of the file from where the data has to be read.
- bytes per send: Size of the block that is sent on the socket.
- ACT: Magic cookie for this asynchronous transmission.

The following steps take place when transmit is called.

- The transmit file creates an Asynchronous Transmit File Result object with all the information that are needed to carry out the asynchronous operation such as I/O handle and bytes to read and also the information needed to call back the application which are `Handler` and `ACT`.

- `transmit file` calls the *TransmitFile* WIN32 API to initiate the asynchronous transmission. `TransmitFile` takes pointer to the *OVERLAPPED* structure. Since we want to preserve additional information such as `ACT` and `Handler` along with `OVERLAPPED` structure, we pass the pointer to `Asynchronous Transmit File Result` object to the `TransmitFile` call.

2.3 Asynchronous Result

The Asynchronous Result classes derive from the `OVERLAPPED` structure and make more useful classes. For each `Asynchronous Operation` class, there is an `Asynchronous Result` class which carries around the additional information besides the information in the `OVERLAPPED` structure. This additional information is needed to execute that operation. The result classes also contain fields to hold the results of the asynchronous operation. The `Asynchronous Result` objects are finally passed to the `Completion Handlers` when the completion is dispatched.

Refer to the UML diagram 6 for the family of `Asynchronous Result` classes.

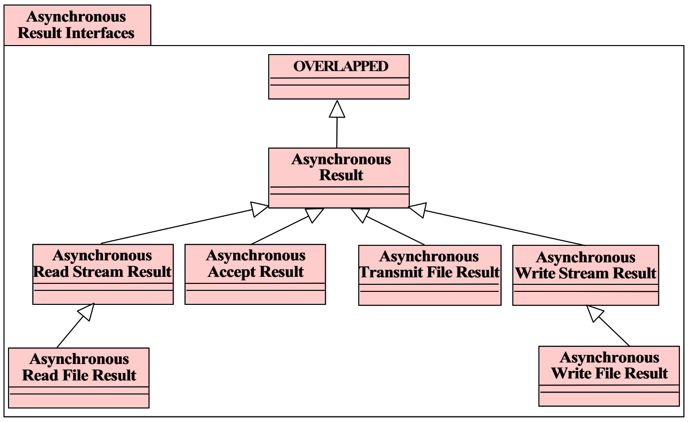

Figure 6: Asynchronous Result Classes

The `Asynchronous Result` base class derives from the `OVERLAPPED` and also abstracts out all the commonalities found in the individual result classes. The individual result classes derive from the `Asynchronous Result` class and add more information pertaining to those corresponding asynchronous operations. The Asynchronous Result classes contain the following items.

- `Handler`: This is application handler that handles completions.
- `ACT`: The magic cookie given by the application when the asynchronous operation is issued.
- `complete method`: This method is defined in the base class `Asynchronous Result` as a pure virtual method. The individual Result classes override this method and call the correct call back methods for that operation.

  For example, the `Asynch Read Stream Result` class calls `handle_read_stream` call back method from the `complete` method. This is very useful because the `Completion Dispatcher` when it gets back a `Asynchronous Result` object from the OS, can call the `complete` method on it without knowing the exact type of the asynchronous operation that has completed.

2.4 Completion Handler

The *Handler* class defines the call back methods which are called by the Proactor framework on completions of asynchronous events.

For each `Asynchronous Operation`, a call back method is defined and default implementations are provided. Therefore, currently, the `Handler` class provides the following call back methods.

- `handle_read_stream`
- `handle_write_stream`
- `handle_read_file`
- `handle_write_file`
- `handle_accept`
- `handle_transmit_file`

Applications define their own handlers deriving from the `Handler` class and fill in the application logic appropriately in the call back methods.

2.5 Completion Dispatcher

The *Proactor* class implements the `Completion Dispatcher` role of the Proactor pattern. Henceforth, we will be using the terms Proactor and Completion Dispatcher interchangeably.

The `Proactor` executes the following steps.

- The event loop on the Proactor executes `GetQueuedCompletionStatus` to get the completions of asynchronous I/O. When there is a completion of an asynchronous event, it gets back an `OVERLAPPED` pointer from the OS. This pointer is down cast to the `Asynchronous Result` type object.

This `Asynchronous Result` object might come from a completion event or it might have been posted through `PostQueuedCompletionStatus`.

- The completion status of the asynchronous operation, which includes the `number of bytes transferred, error code` are obtained from the OS and filled in the `Asynchronous Result` object.
- The `complete` method is invoked on the `Asynchronous Result` object, which calls back the application handler.
- When the call back completes, the `Asynchronous Result` object is deleted by the `Proactor`.

3 Design of the Portable, Extensible and Efficient Proactor Framework

In this section, we will explain how we designed the WIN32 specific Proactor framework to be a poratable, extensible and efficient one. We will first explain the features that the POSIX4 operating systems provide to do asynchronous I/O. We will then explain how we ported each of the participant of the Proactor pattern to the POSIX platforms. We present a straight forward solution to show how the POSIX implementation can be integrated to the existing WIN32 implementation. Then, we show how we architected the design to make it more extensible, scalable and efficient.

3.1 Goals

The following are the goals that we have kept in our mind to guide our design and implementation decisions.

- `Backward Compatibility:` We should keep the existing APIs of the framework intact and extend all the functionalities to work on POSIX platforms.

- `Separation of Interface and Implementation:` Irrespective how many different implementations are provided for the framework, the APIs of the framework should be simple and common across all the platforms or implementations. As far as possible, applications should be freed from worrying about which implementation of the framework they are using etc. But applications should have control over configuring which implementation should be used by the frame work.

- `Scalability:` The design of the frame work is in such a way that it is easier to extend the framework, for example to have more asynchronous operations or to port the framework to more `Asynchronous Processor` implementations or platforms.

- `Flexibility:` The framework APIs should be flexible enough so that applications can easily exploit features which may be special to some particular platforms. For example, assigning *priority* to an asynchronous call is possible on POSIX systems, which is not present on WIN32. The framework should be flexible enough so that applications can make use of such features portably.

3.2 POSIX Asynchronous Operation Processor

The POSIX I/O subsystem provides the following APIs to execute asynchronous operations.

- *aiocb:* This is a structure defined by the POSIX operating system. This is used to pass the parameters to the various *aio_* calls, to issue the asynchronous calls and also to query for the completions of the asynchronous calls. The structure has the following parameters.
  - `int file:` File descriptor or stream descriptor on which an asynchronous operation is done.
  - `void buf:` Location of the buffer that contains the data for the I/O.
  - `size_t nbytes:` Length of the data transfer.
  - `offset:` Offset for the file from where the file operation is done.
  - `int priority:` Priority of the asynchronous operation.
  - `struct signal_event:` Signaling option, signal number and signal information for the asynchronous call.
  - `struct results:` Error code and return value of the asynchronous operation.

- *aio_read:* This call issues an asynchronous read on a stream or a file handle. The `aiocb` structure is used to specify the various parameters to do the read operation.
- *aio_write:* This system call issues asynchronous write on a stream or a file handle.
- *aio_cancel:* This system call is used to cancel all or a particular asynchronous operation issued on a handle.
- *aio_suspend:* This system call is used to wait on an array of `aiocbs` that were used to issue `aio_reads` or `aio_writes`, for their corresponding asynchronous operations to complete.
- *aio_sigtimedwait* or *sigwaitinfo*: This call is used to wait for a set of real-time signals. It also delivers the signal information used when the signal was raised.

- *aio_error:* This system call retrieves the error status for an synchronous operation.
- *aio_return:* This retrieves the return status for an asynchronous operation.

3.3 Signal and Asynchronous I/O Blocks Based Completion Queue Strategies

On POSIX systems, the asynchronous I/O completions can be obtained from the Operating system in two different ways either by using the real-time signals or through the Asynchronous I/O Control Blocks (`aiocbs`) that are used to issue the asynchronous calls.

We have made use of both of these mechanisms and provided two different strategies for the `Completion Dispatcher` i.e. the `Proactor` class and the `Asynchronous Operation` classes.

We call the `aiocb` based completion notification/dispatching mechanism as `AIOCB strategy` and the real-time signals based mechanism as `SIG strategy`. They are implemented as follows.

- `AIOCB Strategy:` The asynchronous I/O control blocks (`aiocbs`), that are used to issue asynchronous I/O calls (`aio_read` or `aio_write`) are stored with the Proactor class.

  When the `aio_read / aio_write` are issued, the signaling option is disabled in the `aiocb` so that the operating system will not raise and queue up the real-time signal when that operation completes.

  The array of `aiocbs` that were stored with the `Proactor` class are then queried for one or more completions using the `aio_suspend` system call.

  This approach needs the `Completion Dispatcher`, i.e., the `Proactor` class, to keep track of all the `aiocbs` which are used to issue the `aio_` calls. This implementation adds more complexity on part of the `Completion Dispatcher`, since they need to maintain the list of `aiocbs` and query for completions on them. Also, this approach may not scale well, since the number of pending asynchronous operations are limited by the size of the *aiocb array* which should be decided at compile time.

- `SIG Strategy:` This completion notification/dispatching strategy is based on the real-time signaling feature of the POSIX4 operating systems.

  A *real-time signal number* and *signal information* is specified in the `aiocb` structure, when the asynchronous I/O call (`aio_read` or `aio_write`) is invoked. On completion of the operation, the operating system raises the real-time signal along with the *signal information* that was specified when the `aio_` call was issued. If that real-time signal is masked, that signal is queued up which can then be received through `sigtimedwait` or `sigwaitinfo` system calls.

  In our framework implementation, we want the completions to be dispatched only when the *event loop* of the `Proactor` is invoked. We do not want the control to move around arbitrarily between the `signal handler` and the rest of the code. Therefore, we mask the real-time signals used for issuing the `aio_` calls and use `sigtimedwait` and `sigwaitinfo` in the `event loop` of the Proactor class, to receive the real-time signals that are queued by the Operating System. In this approach, there is no overhead of maintaining the list of `aiocbs` with the `Proactor`.

In the following sections, we will explain how we have made use of the POSIX4 features to port the framework to POSIX platforms. We will explain how the various components in the Asynchronous Operation and Asynchronous Result classes were ported to POSIX platforms.

3.4 Asynchronous Operation Classes

The functionalities of the various Asynchronous Operation classes shown in Figure 5, are implemented for the POSIX platforms as follows.

3.4.1 Asynchronous Operation Base

The APIs of this class have been ported as follows.

- `open:` POSIX subsystem does not have the concept of the completion port as in WIN32. Therefore, the *open* method initializes the data members for `I/O handle`, `Handler` and `Proactor` so that they can be used while issuing asynchronous calls.
- `cancel:` `aio_cancel` system call is used to implement this API.

3.4.2 Asynchronous Read Stream

The `read` API implementation looks similar to the WIN32 implementation, except the `shared_read` method, calls the `aio_read` to initiate an asynchronous read. `aio_read` operation takes a pointer to the structure `aiocb`. Since we want to preserve information such as `ACT` and `Handler` along with `aiocb` structure for each asynchronous call, we pass the pointer to the `Asynchronous Read Stream Result` object to the `aio_read` call. This is possible since the result classes derive from the `aiocb` structure (refer to Figure 10).

The `shared_read` has been implemented for the two different completion strategies as follows.

- AIOCB strategy: The `aiocb` objects used to issue the *aio_read* are stored with the *Proactor* so that the Proactor can do *aio_suspend* on them, to query for the completion.

- SIG strategy: In this strategy, the parameter `signal number` provided through the `read` API, is used to issue the `aio_read` operation. Thus, applications can specify signal numbers on a per operation basis. But the signal numbers used to issue the asynchronous operations should have been already specified to the Proactor class, so that it can wait for completions with that signal number. The `Asynchronous Read Stream Result` pointer is passed as the signal information, which is received again on completion.

3.4.3 Asynchronous Write Stream

The `write` API implementation works similar to the WIN32 implementation. The `shared_write` makes the call to `aio_write` passing the `Asynchronous Write Stream Result` object pointer and initiates the asynchronous call. In the case of signal based completion strategy, the `Asynchronous Write Stream Result` pointer is passed as the signal information. In the AIOCB strategy, the `aiocb` object, that is used to issue the asynchronous operation, is stored with the Proactor so that it can be used to query for completions.

3.4.4 Asynchronous Read File

This class extends the functionality of the `Asynchronous Read Stream` class to do asynchronous read on a file. `Asynchronous Read File Result` class is passed on to the `shared_read` method of the `Asynchronous Read Stream` class, which invokes the `aio_read` call.

3.4.5 Asynchronous Write File

This class extends the functionality of the `Asynchronous Write Stream` class to do asynchronous write on a file. `Asynchronous Write File Result` class is passed on to the `shared_write` method of the `Asynchronous Write Stream` class, which invokes the `aio_write` call.

3.4.6 Asynchronous Accept

Unlike the WIN32 subsystem, which has *AcceptEx* system call to queue an *accept* operation with the Operating System, POSIX platforms do not provide a system call to do *accept* asynchronously on a *socket*.

When the `accept` API of this class is called it should be carried out asynchronously. We looked at the following approaches for implementing this class.

Thread per Accept: For each asynchronous `accept` the application makes, an `Asynchronous Accept Result` object is created with all the information needed for that invocation. Then, a separate thread is spawned and the `Asynchronous Accept Result` object is passed to that thread.

The thread will block on the `accept` system call. It *notifies* the `Proactor`, when it comes out of the system call. The *post_completion* API of the `Proactor` is used to notify the completions to the `Proactor`.

Refer to the Figure 7 for how this is implemented. But this approach does not scale well since it requires one thread for each pending accept call.

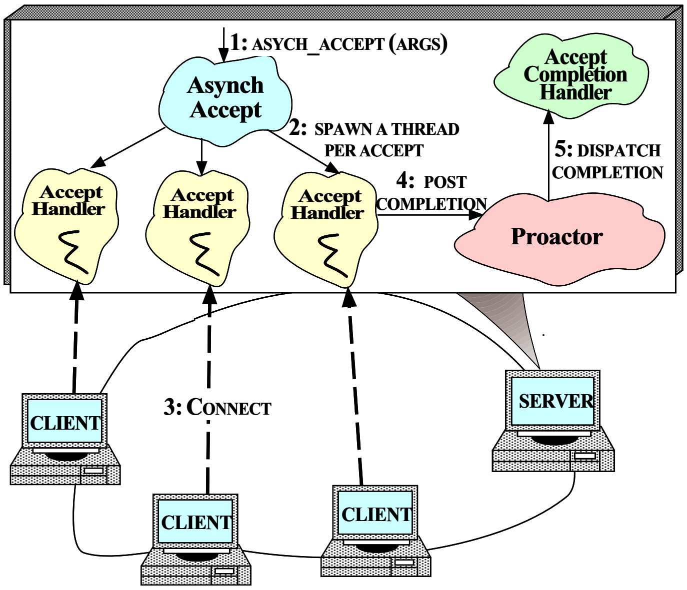

Figure 7: Asynchronous Accept by Thread Per Accept

Reactor with an auxillary thread: Having multiple threads is overkill for doing asynchronous accept. But we have to execute the `accepts` without borrowing the thread that issued the asynchronous `accept` call.

In this implementation, We have an auxillary thread running the *Reactor* [6] event loop. We also maintain the Queue of `Asynchronous Accept Result` objects to keep track of the `asynchronous accepts` issued by the application. Refer to the Figure 8 which explains this implementation.

This model consists of the following components.

- `Auxillary Thread:` When the `open` method on the `Asynchronous Accept` operation class is called, it spawns the thread which always runs the Reactor's event loop. But initially when there is no `accept` call issued by the application, the `handle` on which `accept` is done is disabled in the Reactor, so that it does not accept

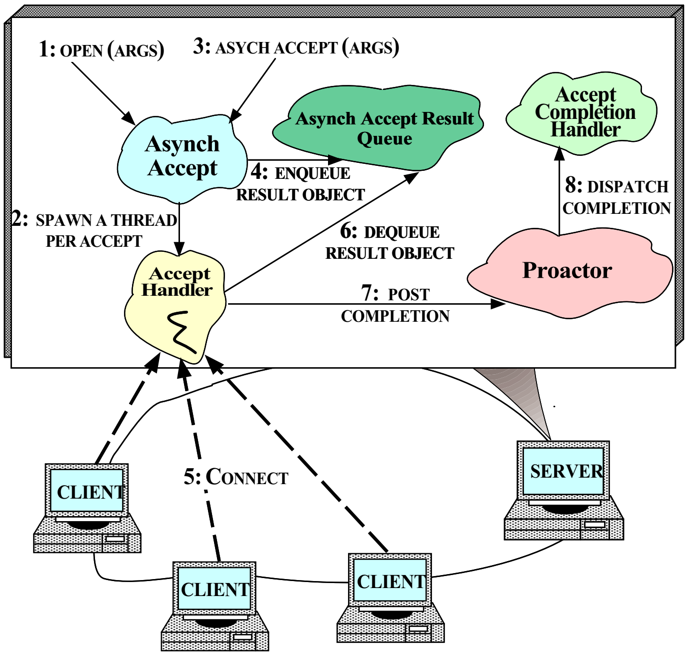

Figure 8: Asynchronous Accept using Reactor with an Auxillary Thread

any connections when there are no `asynchronous accept` calls issued by the application.

- `Event Handler for the Reactor:` The `Asynchronous Accept Handler` is the helper class for implementing the `Asynchronous Accept` operation. It also acts as the Event Handler for the Reactor running in the auxillary thread. This class manages the Queue of `Asynchronous Accept Result` objects.

  When an asynchronous accept is issued by the application, an `Asynchronous Accept Result` object is created for that invocation and enqueued in the `Asynchronous Accept Result Queue`. Then the `handle` on the Reactor is enabled so that connections can be accepted.

  When there is a connection at the handle, the `Reactor` which runs in the auxillary thread calls the `handle_input` of the `Asynchronous Accept Handler` class. The `handle_input` method does a non-blocking `accept` system call on the handle and completes the accept call.

- `Notifying completions to the Proactor:` Once the `accept` system gets completed in the auxillary thread in the `handle_input` method, the completion should be notified to the `Proactor`.

An `Asynchronous Accept Result` object dequeued from the Queue and the result of the `accept` is filled in the object. The order in which the `Asynchronous Accept Result` objects are dequeued, can be decided based on the `priority` of the asynchronous operation.

The `Asynchronous Accept Result` object is then *posted* to the `Proactor` using the `post_completion` API in the `Proactor` class. The implementation of this API is discussed in section 3.8.

3.4.7 Asynchronous Transmit File

WIN32 Operating System provides an API for transmitting a file asynchronously on a socket. It is not so on POSIX platforms. `Asynchronous Transmit File` operation class has been implemented using an `Asynchronous Read File` object which reads from the file and an `Asynchronous Write Stream` object which writes on data read from the file to the socket. The Figure 9 depicts this model.

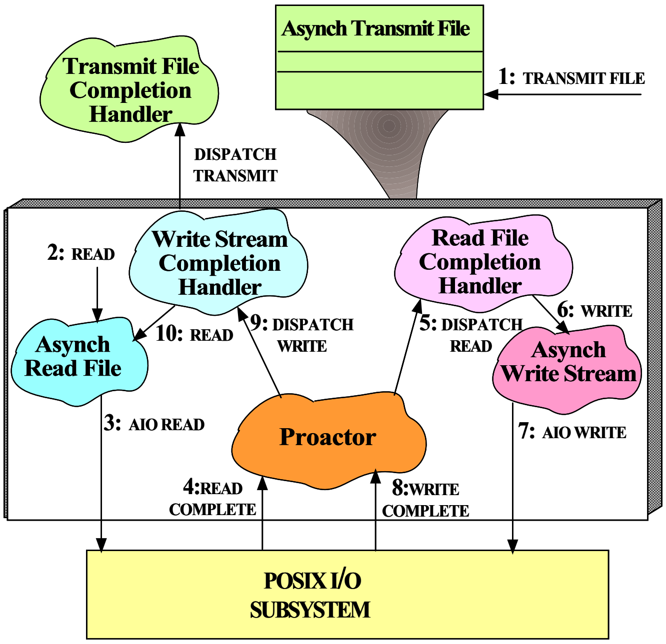

Figure 9: Asynchronous Transmit File

The `Asynchronous Transmit File` class has a helper class called *Asynchronous Transmit File Handler* which the does the transmission on behalf of the `Asynchronous Transmit File` operation class. It contains an `Asynchronous Read File` object and an Asynchronous Write Stream object. The helper class acts as the Completion Handler for both the operations.

When transmit file API is called, an asynchronous read operation is issued on the file and the control returns to the caller. After the asynchronous read completes and when the caller executes the event loop of the Proactor, the call back method handle_read_file of the helper class gets called. In this call back method, an asynchronous write is initiated to write all the data read from the file on to the socket. When the asynchronous write completes, handle_write_stream method gets called on the helper class. It is possible to have partial writes on the socket. The state of the writes are taken care of by the helper class. Finally when the write fully completes, the helper class initiates an asynchronous read to get the next block of data from the file. The helper class keeps the state of the transmission such as current offset of the file, ACT etc.

Special ACT strings are used to send the header and trailer before and after the file transmission respectively. The ACTs are very useful for the call back methods to differentiate between the completions of header or trailer transmission and the file data transmission. The call back method handle_write_stream, when the header transmission completes, initiates the file read to start transmitting the file. It initiates transmitting the trailer, when the file transmission completes. And when the trailer transmission also completes, it calls the dispatches completion of the transmit file operation.

3.5 Asynchronous Result Classes

We will explain here how the Asynchronous Result classes in the original WIN32 solution were ported to POSIX platforms.

Since the *aio* system calls takes a pointer to the *aiocb* structure, we derive the Asynchronous Result classes from the *aiocb* structure. The rest of the inheritance hierarchy is the same as in WIN32 implementation. Refer to the UML diagram 10 for the Asynchronous Result classes.

3.6 Completion Handler

This component of the framework does not have any platform specific implementations. Hence this is kept common for all the implementations of the framework.

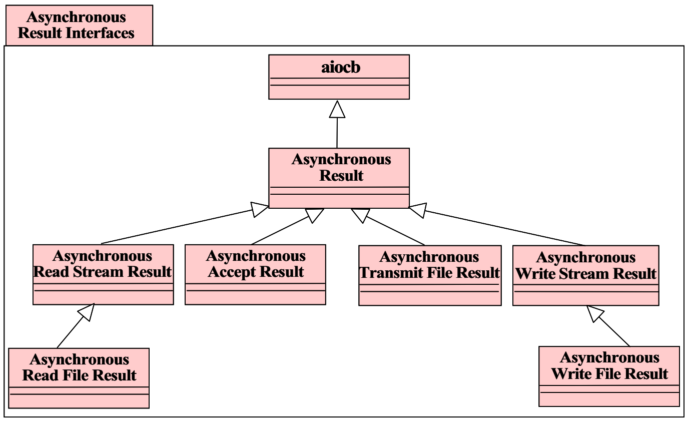

Figure 10: Porting Asynchronous Result Classes to POSIX

3.7 Completion Dispatcher

The Proactor class implements both the completion strategies discussed in 3.3.

To wait for the completions and the to dispatch them, the event loop executes the following steps.

- AIOCB Strategy: In this implementation, the aio_suspend call is used to wait on the array of aiocb objects that are used to issue the asynchronous operations. aio_suspend call returns when there is atleast one completion.

  aio_error is used to find out the correct aiocb object in the array for which the completion had occured. This call also gets the error status of the asynchronous operation.

  we cast the aiocb pointer to the the derived class Asynchronous Result object, since the dispatching functionalities are available only in the Asynchronous Result class.

- SIG Strategy: In this strategy, sigwaitinfo is called with the signal set that has been already masked for all the threads. The sigtimedwait call is used, instead of sigwaitinfo when the timed event loop is called. These wait calls complete on arrival of a signal that is present in the signal set.

  The Asynchronous Result object associated with the signal delivery is retrieved from the signal information obtained in the call.

  aio_error is called on this object to get the error status of the asynchronous operation.

aio_return is called on that aiocb object to retrive the return status of the asynchronous operation.

To dispatch the completion, The complete method is invoked on the Asynchronous Result object, which calls the correct call back method in the completion handler.

When the call back completes, the Asynchronous Result object is deleted by the Proactor.

3.8 Posting Completions to Completion Queue

WIN32 provides the API PostQueuedCompletionStatus to post a completion (a pointer to the OVERLAPPED structure) to a Completion Port.

The API *post_completion* of the Proactor class takes a pointer to the Asynchronous Result object. On POSIX, we have implemented post_completion for the two different completion strategies as follows.

SIG Strategy: The system call sigqueue is used to queue up a reserved real-time signal to the current process. The Asynchronous Result object is assigned as the signal information in the sigqueue call.

The Proactor's event loop which waits for the reserved real-time signal receives the Result block and calls complete method on it, do dispatch the completion.

AIOCB Strategy: We make use of a notify pipe to send the Asynchronous Result objects to the Proactor's completion queue. The Proactor reads the pipe at the other end for the Asynchronous Result objects. Reading for the Asynchronous Result base class pointer helps because a pointer to any type that derives from the Asynchronous Result can be posted through the notify pipe. For example, Asynchronous Accept Result object is posted in the Asynchronous Accept operation implementation. Also, applications can derive their own Asynchronous Result classes and use them for posting to the Proactor to fake completions.

Reading from the pipe has to be synchronized with the event loop of the Proactor, so that the completions are dispatched only when the even loop is running. To achieve this, the Proactor issues an asynchronous read on the notify pipe using the Asynchronous Read Stream operation class object. A helper class called Notify Pipe Manager manages the pipe and acts as the Completion Handler for the asynchronous read operation issued on the pipe.

The Asynchronous Result pointers are read asynchronously from the notify pipe and dispatched during Proactor's event loop.

When a read from the pipe completes, Notify Pipe Manager's handle_read_stream gets called. This method receives the Asynchronous Result pointer and calls complete on it, which dispatches the completion.

After the dispatch, a new asynchronous read is issued on the pipe to handle the completions in the future.

The Figure 11 explains this implementation. The abstract class Handler provides the default implementations for the completion call back hook methods such as handle_read_stream etc.

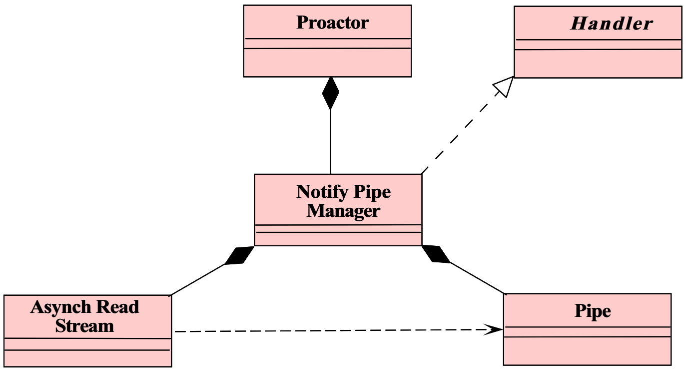

Figure 11: Posting Completions to the Proactor

3.9 A Simple Integrated Proactor Framework

We now integrate the POSIX implementations discussed above with the existing WIN32 implementation. A straight forward integration makes use of the following tricks.

Conditional compilation: We use #if defined statements all over the code to switch between the WIN32 implementation and the POSIX implementations. Class definitions, where we change the inheritance hierarchy depending on the platform, are guarded by pre-compiler conditional directives. For example, the Asynchronous Result classes derive from OVERLAPPED structure on WIN32, but nthey derive from the aiocb structure on POSIX platforms.

The definition of each function which has platform specific implementations also has a similar kind of condition compiler directives to switch between the implementations based on the platforms.

Switch statements: With in the POSIX implementation, we need to switch between, the two different completion strategies that we are using. We need to use the run time switches based on a variable to switch between the two different implementation code. If not run time switches, we need to use the conditional compiler directives for this too.

3.9.1 Drawbacks with the Simple Integrated Proactor Framework

We have achieved the portability goal. We did not make any change to the existing APIs in the framework. We have also kept the APIs simple. But the integrated framework we have discussed so far, is not flexible, extensible and scalable, because of the following reasons.

- Unstructured code: Since we have merged the POSIX specific implementation code on the WIN32 specific implementation, the source code is full of #ifdef precompiler directives which are there to make sure the correct code is compiled on a platform. This really makes the implementation unreadable unstructured.
- Scalability still remains an issue. For example porting the implementation to a new platform will involve defining new #ifdef pre-compiler directive. The implementation code becomes more and more ugly and complex. Even within the POSIX implementation, we have two different implementations and some platforms may need a different implementation.

3.10 Extensible Proactor Framework Design

In this section, we will explain how we enhanced our portable design to be highly extensible and easy to maintain. This design has totally eliminated the precompiler directives from the source code. The code for each implementation is totally decoupled from the other implementations, but all the implementations are bridged by a common simple Interface. The interface has not changed much from the original WIN32-only solution. Therefore the existing applications do not have to change drastically. We have applied the following concepts in order to achieve our goals.

- `Applying Bridge Pattern:` We have applied the Bridge pattern to decouple the concrete implementations from the API. This eliminates all the conditional precompiler directives which switched the code between the WIN32 and the POSIX platform implementations. We have applied bridging to Asynchronous Operation, Asynchronous Result and the Proactor classes. Since the APIs have not changed, the new design is still compatible with the old applications.
- `Inheritance instead of switching:` We have provided separate inheritance hierarchies for the two different POSIX implementations. So all the switch statements for switching between the implementations have been removed. The common code among the two POSIX implementations have been abstracted out to base class. Refer to figures 12, 16 and 20 how the Asynchronous Operation, Asynchronous Result and Proactor classes have been redesigned.
- Factory Methods: The `Proactor` class defines the factory methods [7] to create the correct implementation objects for the `Asynchronous Operation` and `Asynchronous Result` classes.

  Once the right `Proactor` implementation is decided for the application, the switching between the different concrete implementations is decided automatically by the Proactor class through the factory methods.

In the next section, we explain how we re-designed each of the participant in the Proactor pattern to achieve our goals.

3.11 Design of the Asynchronous Operation Classes

The UML diagram in Figure 12 shows how the `Asynchronous Operation` classes have been redesigned in order to fulfill our goals.

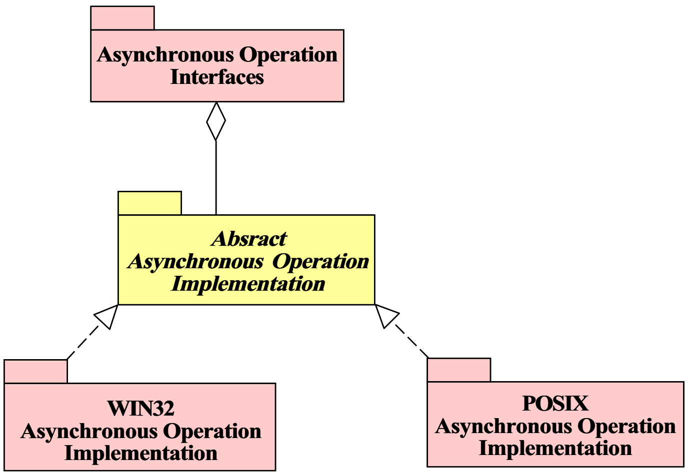

Figure 12: Bridged Asynchronous Operation Classes

The `Asynchronous Operation Interfaces` package, contains the interface classes. The implementation classes are bridged with the interface classes through the `Abstract Asynchronous Operation Implementation` package. The packages `WIN32 Asynchronous Operation Implementation` and `POSIX Asynchronous Operation Implementation` provide the implementation classes for the WIN32 and POSIX platforms respectively.

The UML diagram of the `Asynchronous Operation Interfaces` package looks exactly similar to the Figure 5. But, the interface classes are free from precompiler directives now. They forward their methods to the implementation classes which are bridged together by `Abstract Asynchronous Operation Implementation` classes.

The UML diagram of the `WIN32 Asynchronous Operation Implementation` package is shown in the Figure 13.

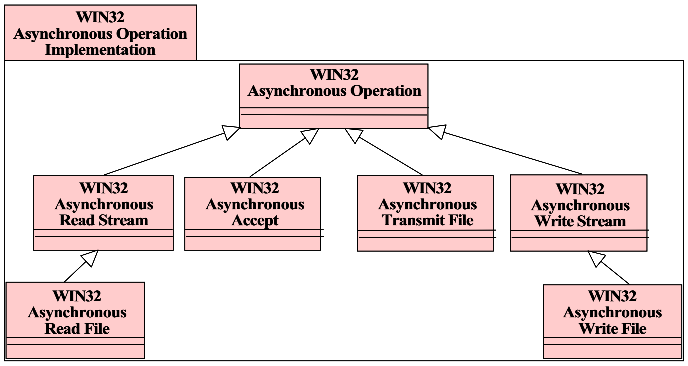

Figure 13: WIN32 Asynchronous Operation Implementation

The UML diagram of the `POSIX Asynchronous Operation Implementation` package is shown in the Figure 14. The implementations of the `Asynchronous`

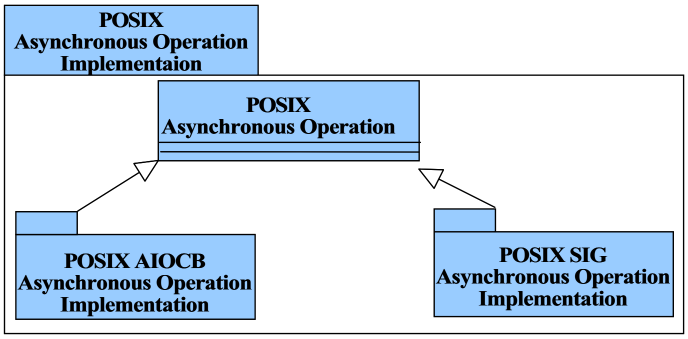

Figure 14: POSIX Asynchronous Operation Implementation

`Operation` classes for the two different completion strategies 3.3 are separated in to two packages as shown in the Figure 14.

The package `POSIX AIOCB Asynchronous Operation Implementation` defines the Asynchronous Operation classes for the AIOCB completion strategy. This package is shown in Figure 15.

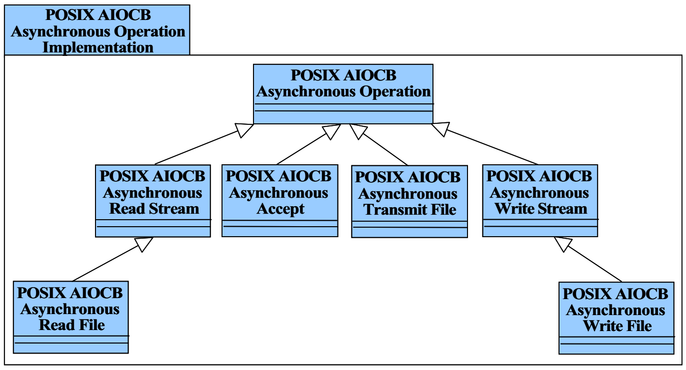

Figure 15: POSIX AIOCB Asynchronous Operation Implementation

The package `POSIX SIG Asynchronous Operation Implementation` defines the Asynchronous Operation classes for the SIG completion strategy. The UML inheritance hierarchy for this strategy looks similar to the POSIX AIOCB Asynchronous Implementation package shown in 15.

3.12 Design of the Asynchronous Result Classes

The architecture of the new Asynchronous Result classes has been shown in the Figure 16. The

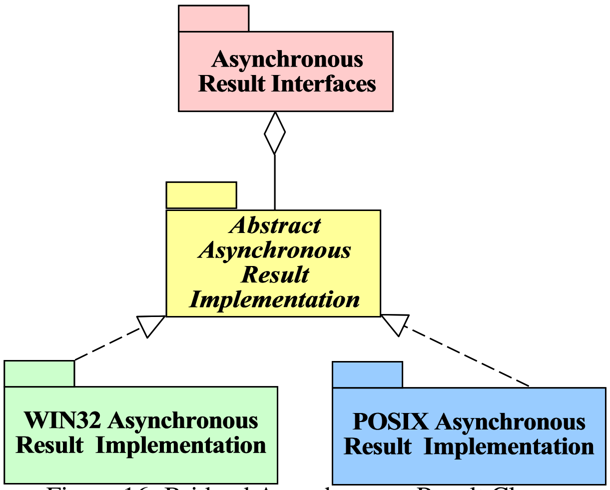

Figure 16: Bridged Asynchronous Result Classes

`Asynchronous Result Interfaces` package, defines the interfaces to the various Asynchronous Result class implementations. The implementations are bridged with the interfaces through the abstract classes in the `Abstract Asynchronous Result Implementation` package. The packages `WIN32`

`Asynchronous Result Implementation` and `POSIX Asynchronous Result Implementation` implement the `Asynchronous Result Interfaces` for the WIN32 and the POSIX4 platforms respectively.

The UML architecture of the `Asynchronous Result Interfaces` has been shown in Figure 17. Note that the `Asynchronous Result` base class does not have to derive from platform specific `OVERLAPPED` or `aiocb` structure. Therefore, there is no need for precompiler directives to switch between the implementation. All the interfaces have the reference to the implementation objects, where they forward all their methods.

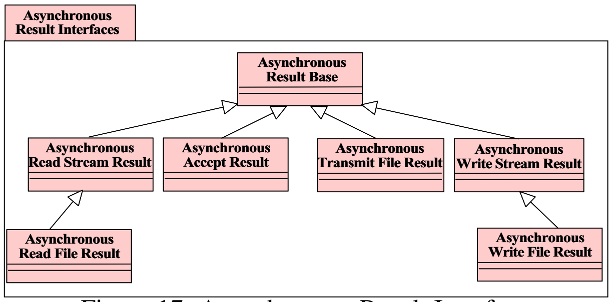

Figure 17: Asynchronous Result Interfaces

The UML diagram of the WIN32 Asynchronous Result classes are shown in Figure 18.

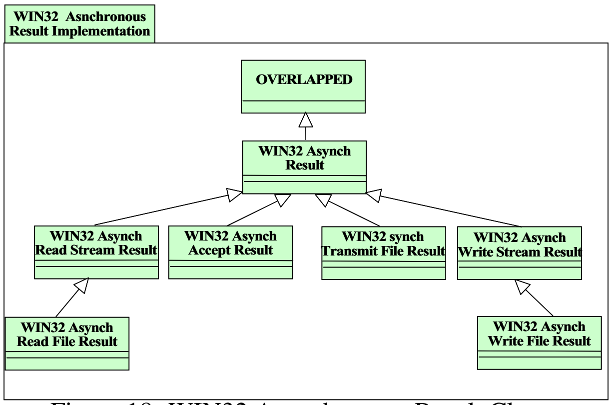

Figure 18: WIN32 Asynchronous Result Classes

The UML diagram of the POSIX Asynchronous Result classes are shown in Figure 19.

Note that the WIN32 and the POSIX implementation classes do not have to have any precompiler directives now.

3.13 Design of the Proactor class

The UML diagram in Figure 20 shows how the Proactor class has been redesigned.

The `Proactor Interface` class acts as the interface to the different Proactor class implementations. The

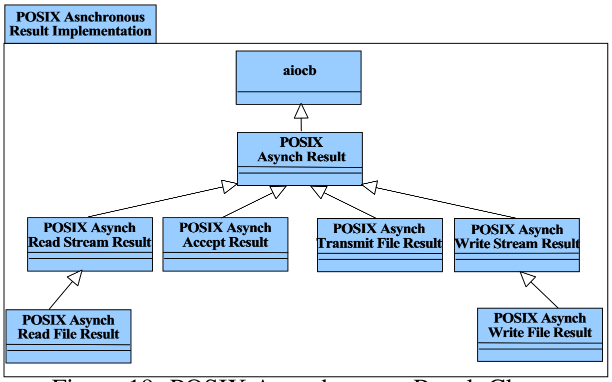

Figure 19: POSIX Asynchronous Result Classes

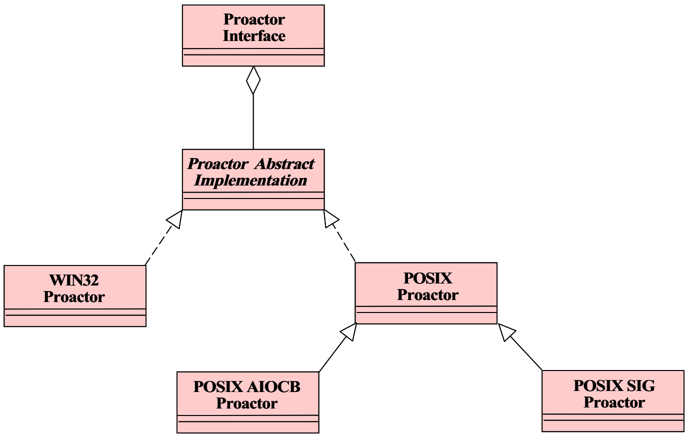

Figure 20: Bridged Proactor Classes

Proactor Abstract Implementation bridges the implementations with the Proactor interface. The Proactor Interface class simply forwards all the methods to the implementation classes.

The two different completion notification/dispatching mechanisms discussed in 3.3, are implemented by the POSIX AIOCB Proactor class and the POSIX SIG Proactor class. The POSIX AIOCB Proactor implements the AIOCB strategy and the The POSIX SIG Proactor class implements the SIG Strategy. The common code between these two implementations have been abstracted out in the POSIX Proactor class.

3.14 Design Analysis

Let us now analyze this design from the view of the goals we had initially.

- Backward Compatibility: The main interfaces of the framework have not been changed, except that the following minor changes have been done to the APIs of the framework to fit with the new design.
  - Proactor Constructor: The constructor of the Proctor class in the interface level, takes Abstract Implementation objects, instead of platform specific constructor parmeters. When no implementation is given, it creates implementations based on the predefined constants, with default options. To override this, applications can create the implementation and then create the interface Proactor object with that implementation.
  - post_completion: This API has been taken off from the main Proactor interface and have been moved to the Proactor Abstract Implementation class. This is necessary since the Abstract Asynchronous Result classes which are posted as completions have different base classes, in POSIX and WIN32 platforms.
  - complete: This method is defined in the Asynchronous Result Implementation classes. This method is used by the Proactor Implementation classes to dispatch the completions to the correct call back methods.

    Applications sometime exploit this feature to fake completions to their handlers. Now, since the Asynchronous Result Interface classes are different from the Asynchronous Result Implementation classes, application should make use of the factory methods defined in the Proactor class, in order to obtain the correct Asynchronous Result Implementations and call complete method on them.
- Separation of Interface and Implementation: We have separated the interfaces from the implementations. We have also provided separation between the various implementations.
- Scalability: The framework now has a very clean architecture to scale.

  For example, to port the framework to a new Asynchronous Operation Processor implementation or to a new Operating System involves reusing the existing bridging hierarchy and define a new implementation hierarchy for the new implementation. Factory methods and call back methods have to be defined appropriately for the new implementation/

  In the simple design discussed in 3.9, this would have involved switch statements and #ifdef precompiler directives all over the code to incorporate the new implementation.
- Flexibility: We have made use of platform specific features in the framework so that applications can use them to their advantage on those platforms with out loosing portability. Default values have been provided to such features.

  For example, specifying priority value for the asynchronous operation is provided to the APIs of the Asynchronous Operation so that applications on POSIX systems can use them successfully. But it is provided with a default value so that applications on WIN32 platforms need not be concerned with it.

4 Conclusions

The WIN32 platform specific Proactor framework was extended to work on the POSIX implementations of the Asynchronous Operation Processor. We have made our design in such way that the framework is not only portable but also extensible to include more features, scalable to more implementations of Asynchronous Operations Processor and highly efficient.

References

[1] I. Pyarali, T. Harrison, D. C. Schmidt, and T. D. Jordan, "Proactor – An Architectural Pattern for Demultiplexing and Dispatching Handlers for Asynchronous Events," in *The* 4^{th} *Pattern*

Languages of Programming Conference (Washington University technical report #WUCS-97-34), September 1997.

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

[3] J. C. Mogul, "The Case for Persistent-connection HTTP," in *Proceedings of ACM SIGCOMM '95 Conference in Computer Communication Review*, (Boston, MA), pp. 299–314, ACM Press, August 1995.

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

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

[6] 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), pp. 1–10, August 1994.

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