Proactor

An Object Behavioral Pattern for Demultiplexing
and Dispatching Handlers for Asynchronous Events

Irfan Pyarali, Tim Harrison, and Douglas C. Schmidt
{irfan, harrison, schmidt}@cs.wustl.edu
Dept. of Computer Science
Washington University[2]
St. Louis, MO 63130, (314) 935-7538

Thomas D. Jordan
ace@programmer.net[1]
SoftElegance
Hartford, WI 53027
(414) 673-9813

This paper appeared at the 4^{th} annual Pattern Languages of Programming conference held in Allerton Park, Illinois, September, 1997.

Abstract

Modern operating systems provide multiple mechanisms for developing concurrent applications. Synchronous multi-threading is a popular mechanism for developing applications that perform multiple operations simultaneously. However, threads often have high performance overhead and require deep knowledge of synchronization patterns and principles. Therefore, an increasing number of operating systems support asynchronous mechanisms that provide the benefits of concurrency while alleviating much of the overhead and complexity of multi-threading.

The Proactor pattern presented in this paper 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.

1 Intent

The Proactor pattern 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] Alternative point of contact is thomas_jordan@deluxedata.com.

[2] This research is supported in part by a grant from Siemens MED.

2 Motivation

This section provides the context and motivation for using the Proactor pattern.

2.1 Context and Forces

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 [1, 2]. Figure 1 shows a typical interaction between Web browsers and a Web server. When a user instructs a browser to open a URL, the browser

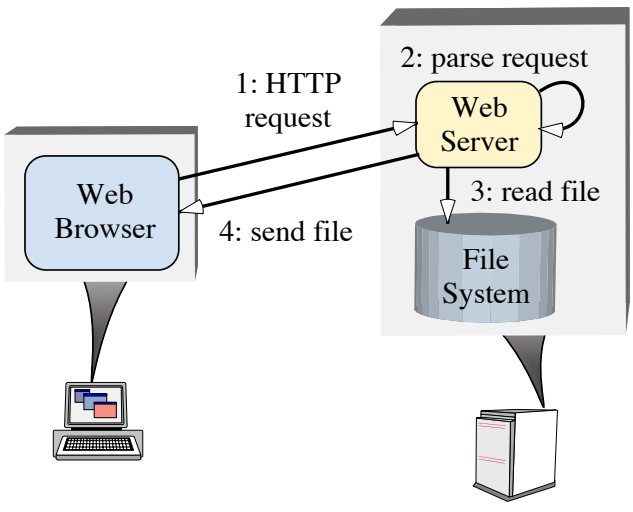

Figure 1: Typical Web Server Communication Software Architecture

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, including multiple synchronous threads, reactive synchronous event dispatching, and proactive asynchronous event dispatching. Below, we examine the drawbacks of conventional approaches and explain how the Proactor pattern provides a powerful technique that supports an efficient and flexible asynchronous event dispatching strategy for high-performance concurrent applications.

2.2 Common Traps and Pitfalls of Conventional Concurrency Models

Synchronous multi-threading and reactive programming are common ways of implementing concurrency. This section describes the shortcomings of these programming models.

2.2.1 Concurrency Through Multiple Synchronous Threads

Perhaps the most intuitive way to implement a concurrent Web server is to use *synchronous multi-threading*. In this model, multiple server threads process HTTP `GET` requests from multiple clients simultaneously. Each thread performs connection establishment, HTTP request reading, request parsing, and file transfer operations synchronously. As a result, each operation blocks until it completes.

The primary advantage of synchronous threading is the simplification of application code. In particular, operations performed by a Web server to service client A's request are mostly independent of the operations required to service client B's request. Thus, it is easy to service different requests in separate threads because the amount of state shared between the threads is low, which minimizes the need for synchronization. Moreover, executing application logic in separate threads allows developers to utilize intuitive sequential commands and blocking operations.

Figure 2 shows how a Web server designed using synchronous threads can process multiple clients concurrently. This figure shows a `Sync Acceptor` object that encapsulates the server-side mechanism for synchronously accepting network connections. The sequence of steps that each thread

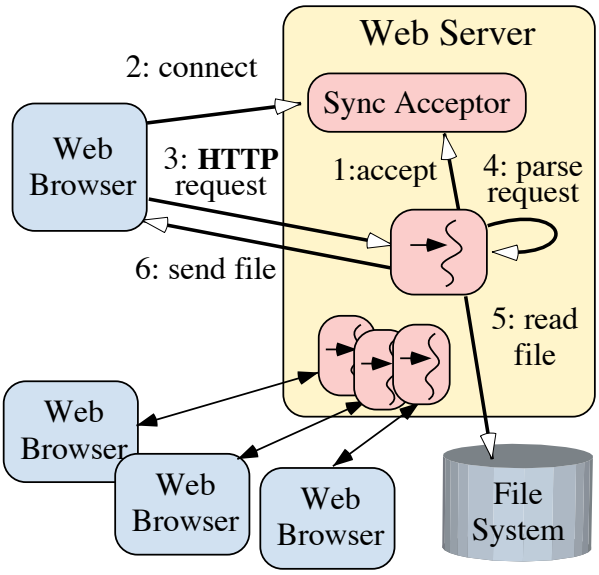

Figure 2: Multi-threaded Web Server Architecture

executes to service an HTTP `GET` request using a Thread Per Connection concurrency model can be summarized as follows:

1. Each thread synchronously blocks in the `accept` socket call waiting for a client connection request;
2. A client connects to the server, and the connection is accepted;
3. The new client's HTTP request is synchronously read from the network connection;
4. The request is parsed;
5. The requested file is synchronously read;
6. The file is synchronously sent to the client.

A C++ code example that applies the synchronous threading model to a Web server appears in Appendix A.1.

As described above, each concurrently connected client is serviced by a dedicated server thread. The thread completes a requested operation synchronously before servicing other HTTP requests. Therefore, to perform synchronous I/O while servicing multiple clients, the Web server must spawn multiple threads. Although this synchronous multi-threaded model is intuitive and maps relatively efficiently onto multi-CPU platforms, it has the following drawbacks:

Threading policy is tightly coupled to the concurrency policy: This architecture requires a dedicated thread for each connected client. A concurrent application may be better optimized by aligning its threading strategy to available resources (such as the number of CPUs via a Thread Pool) rather than to the number of clients being serviced concurrently;

Increased synchronization complexity: Threading can increase the complexity of synchronization mechanisms necessary to serialize access to a server's shared resources (such as cached files and logging of Web page hits);

Increased performance overhead: Threading can perform poorly due to context switching, synchronization, and data movement among CPUs [4];

Non-portability: Threading may not be available on all OS platforms. Moreover, OS platforms differ widely in terms of their support for pre-emptive and non-preemptive threads. Consequently, it is hard to build multi-threaded servers that behave uniformly across OS platforms.

As a result of these drawbacks, multi-threading is often not the most efficient nor the least complex solution to develop concurrent Web servers.

2.2.2 Concurrency Through Reactive Synchronous Event Dispatching

Another common way to implement a synchronous Web server is to use a *reactive event dispatching* model. The Reactor pattern [5] describes how applications can register `Event Handlers` with an `Initiation Dispatcher`. The `Initiation Dispatcher` notifies the `Event Handler` when it is possible to initiate an operation without blocking.

A single-threaded concurrent Web server can use a reactive event dispatching model that waits in an event loop for a `Reactor` to notify it to initiate appropriate operations. An example of a reactive operation in the Web server is the registration of an `Acceptor` [6] with the `Initiation Dispatcher`. When data arrives on the network connection, the dispatcher calls back the `Acceptor`. The `Acceptor` accepts the network connection and creates an `HTTP Handler`. This `HTTP Handler` then registers with the `Reactor` to process the incoming URL request on that connection in the Web server's single thread of control.

Figures 3 and 4 show how a Web server designed using reactive event dispatching handles multiple clients. Figure 3 shows the steps taken when a client connects to the Web server. Figure 4 shows how the Web server processes a client request. The sequence of steps for Figure 3 can be summarized as follows:

1. The Web Server registers an `Acceptor` with the `Initiation Dispatcher` to accept new connections;
2. The
   Web Server invokes event loop of the `Initiation Dispatcher`;
3. A client connects to the Web Server;
4. The `Acceptor` is notified by the `Initiation Dispatcher` of the new connection request and the `Acceptor` accepts the new connection;

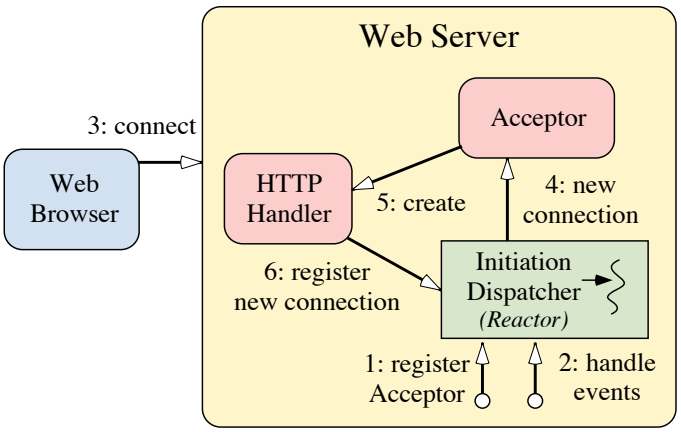

Figure 3: Client Connects to Reactive Web Server

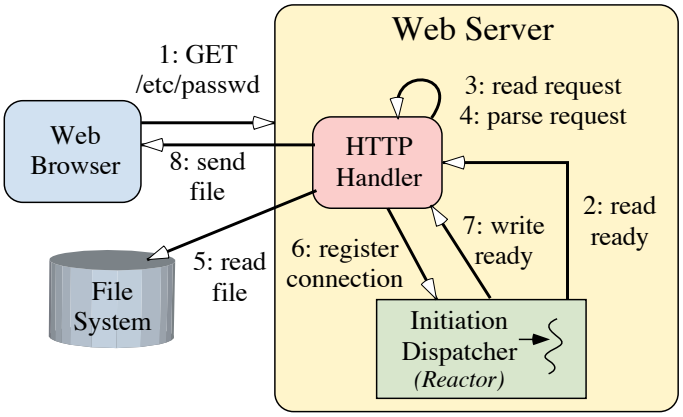

Figure 4: Client Sends HTTP Request to Reactive Web Server

5. The `Acceptor` creates an `HTTP Handler` to service the new client;
6. `HTTP Handler` registers the connection with the `Initiation Dispatcher` for reading client request data (that is, when the connection becomes "ready for reading");
7. The `HTTP Handler` services the request from the new client.

Figure 4 shows the sequence of steps that the reactive Web Server takes to service an HTTP `GET` request. This process is described below:

1. The client sends an HTTP `GET` request;
2. The `Initiation Dispatcher` notifies the `HTTP Handler` when client request data arrives at the server;
3. The request is read in a non-blocking manner such that the read operation returns EWOULDBLOCK if the operation would cause the calling thread to block (steps 2 and 3 repeat until the request has been completely read);
4. The `HTTP Handler` parses the HTTP request;

5. The requested file is synchronously read from the file system;
6. The `HTTP Handler` registers the connection with the `Initiation Dispatcher` for sending file data (that is, when the connection becomes "ready for writing");
7. The `Initiation Dispatcher` notifies the `HTTP Handler` when the TCP connection is ready for writing;
8. The `HTTP Handler` sends the requested file to the client in a non-blocking manner such that the write operation returns EWOULDBLOCK if the operation would cause the calling thread to block (steps 7 and 8 will repeat until the data has been delivered completely).

A C++ code example that applies the reactive event dispatching model to a Web server appears in Appendix A.2.

Since the `Initiation Dispatcher` runs in a single thread, network I/O operations are run under control of the `Reactor` in a non-blocking manner. If forward progress is stalled on the current operation, the operation is handed off to the `Initiation Dispatcher`, which monitors the status of the system operation. When the operation can make forward progress again, the appropriate `Event Handler` is notified.

The main advantages of the reactive model are portability, low overhead due to coarse-grained concurrency control (that is, single-threading requires no synchronization or context switching), and modularity via the decoupling of application logic from the dispatching mechanism. However, this approach has the following drawbacks:

Complex programming: As seen from the list above, programmers must write complicated logic to make sure that the server does not block while servicing a particular client.

Lack of OS support for multi-threading: Most operating systems implement the reactive dispatching model through the `select` system call [7]. However, `select` does not allow more than one thread to wait in the event loop on the same descriptor set. This makes the reactive model unsuitable for high-performance applications since it does not utilize hardware parallelism effectively.

Scheduling of runnable tasks: In synchronous multi-threading architectures that support pre-emptive threads, it is the operating system's responsibility to schedule and time-slice the runnable threads onto the available CPUs. This scheduling support is not available in reactive architectures since there is only one thread in the application. Therefore, developers of the system must by careful to time-share the thread between all the clients connected to the Web server. This can be accomplished by only performing short duration, non-blocking operations.

As a result of these drawbacks, reactive event dispatching is not the most efficient model when hardware parallelism is available. This model also has a relatively high level of programming complexity due to the need to avoid blocking I/O.

2.3 Solution: 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 then 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 dequeueing completion notifications and executing the appropriate callback that contains application-specific Web server code.

Figures 5 and 6 show how a Web server designed using proactive event dispatching handles multiple clients concurrently within one or more threads. Figure 5 shows the sequence of steps taken when a client connects to the Web Server.

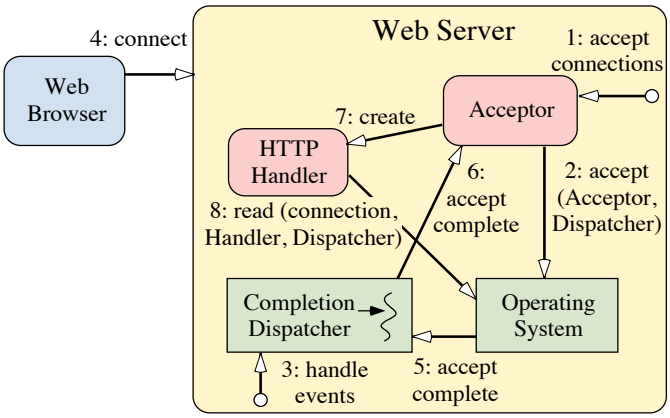

Figure 5: Client connects to a Proactor-based Web Server

1. The Web Server instructs the `Acceptor` to initiate an asynchronous accept;
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 6 shows the sequence of steps that the proactive Web Server takes to service an HTTP `GET` request. These steps

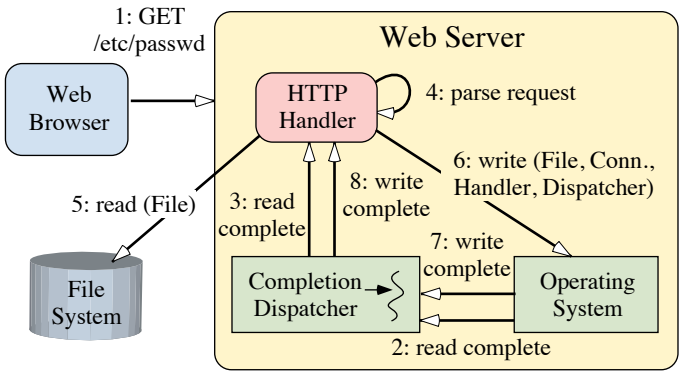

Figure 6: Client Sends requests to a Proactor-based Web Server

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;
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).

A C++ code example that applies the proactive event dispatching model to a Web server appears in Section 8.

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. Section 7 describes how to apply other patterns (such as the Asynchronous Completion Token [8]) to simplify the asynchronous application programming model.

3 Applicability

Use the Proactor pattern 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.

4 Structure and Participants

The structure of the Proactor pattern is illustrated in Figure 7 using OMT notation.

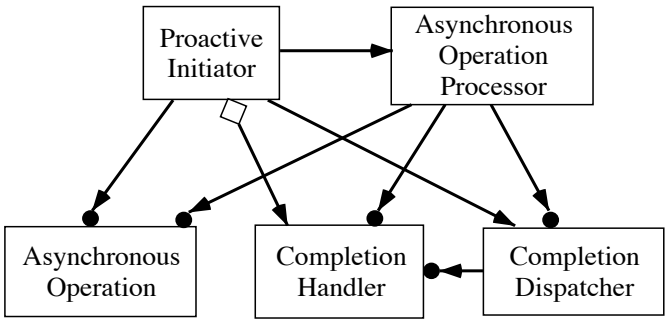

Figure 7: Participants in the Proactor Pattern

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 a `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 `Async_Read`, `Async_Write`, and `Async_Accept`):

- `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.[3] 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`.

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

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.

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 its behalf.

5 Collaborations

There are several well-defined steps that occur for all `Asynchronous Operations`. At a high level of abstraction, applications initiate operations asynchronously and are notified when the operations complete. Figure 8 shows the following interactions that must occur between the

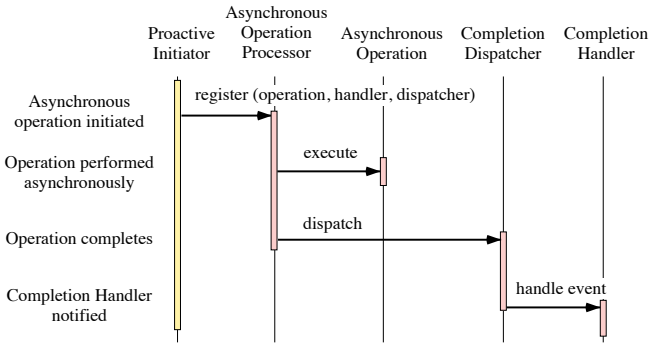

Figure 8: Interaction Diagram for the Proactor Pattern

pattern participants:

1. Proactive Initiators initiates operation: To perform asynchronous operations, the application initiates the operation on the `Asynchronous Operation Processor`. For instance, a Web server might ask the OS to transmit a file over the network using a particular socket connection. To request such an operation, the Web server must specify which file and network connection to use. Moreover, the Web server must specify (1) which `Completion Handler` to notify when the operation completes and (2) which `Completion Dispatcher` should perform the callback once the file is transmitted.

2. Asynchronous Operation Processor performs operation: When the application invokes operations on the `Asynchronous Operation Processor` it runs them asynchronously with respect to other application operations. Modern operating systems (such as Solaris and Windows NT) provide asynchronous I/O subsystems within the kernel.

3. The Asynchronous Operation Processor notifies the Completion Dispatcher: When operations complete, the `Asynchronous Operation Processor` retrieves the `Completion Handler` and `Completion Dispatcher` that were specified when the operation was initiated. The `Asynchronous Operation Processor` then passes the `Completion Dispatcher` the result of the `Asynchronous Operation` and the `Completion Handler` to call back. For instance, if a file was transmitted asynchronously, the `Asynchronous Operation Processor` may report the completion status (such as success or failure), as well as the number of bytes written to the network connection.

4. Completion Dispatcher notifies the application: The `Completion Dispatcher` calls the completion hook on the `Completion Handler`, passing it any completion data specified by the application. For instance, if an asynchronous read completes, the `Completion Handler` will typically be passed a pointer to the newly arrived data.

6 Consequences

This section details the consequences of using the Proactor Pattern.

6.1 Benefits

The Proactor pattern offers the following benefits:

Increased separation of concerns: The Proactor pattern decouples application-independent asynchrony mechanisms from application-specific functionality. The application-independent mechanisms become reusable components that know how to demultiplex the completion events associated with `Asynchronous Operations` and dispatch the appropriate callback methods defined by the `Completion Handlers`. Likewise, the application-specific functionality knows how to perform a particular type of service (such as HTTP processing).

Improved application logic portability: It improves application portability by allowing its interface to be reused independently of the underlying OS calls that perform event demultiplexing. These system calls detect and report the events that may occur simultaneously on multiple event sources. Event sources may include I/O ports, timers, synchronization objects, signals, etc. On real-time POSIX platforms, the asynchronous I/O functions are provided by the `aio` family of APIs [9]. In Windows NT, I/O completion ports and overlapped I/O are used to implement asynchronous I/O [10].

The Completion Dispatcher encapsulates the concurrency mechanism: A benefit of decoupling the `Completion Dispatcher` from the `Asynchronous Operation Processor` is that applications can configure `Completion Dispatchers` with various concurrency strategies without affecting other participants. As discussed in Section 7, the `Completion Dispatcher` can be configured to use several concurrency strategies including single-threaded and Thread Pool solutions.

Threading policy is decoupled from the concurrency policy: Since the `Asynchronous Operation Processor` completes potentially long-running operations on behalf of `Proactive Initiators`, applications are not forced to spawn threads to increase concurrency. This allows an application to vary its concurrency policy independently of its threading policy. For instance, a Web server may only want to have one thread per CPU, but may want to service a higher number of clients simultaneously.

Increased performance: Multi-threaded operating systems perform context switches to cycle through multiple threads of control. While the time to perform a context switch remains fairly constant, the total time to cycle through a large number of threads can degrade application performance significantly if the OS context switches to an idle thread[4]. For instance, threads may poll the OS for completion status, which is inefficient. The Proactor pattern can avoid the cost of context switching by activating only those logical threads of control that have events to process. For instance, a Web server does not need to activate an HTTP Handler if there is no pending `GET` request.

Simplification of application synchronization: As long as `Completion Handlers` do not spawn additional threads of control, application logic can be written with little or no regard to synchronization issues. `Completion Handlers` can be written as if they existed in a conventional single-threaded environment. For instance, a Web server's HTTP `GET` Handler can access the disk through an `Async Read` operation (such as the Windows NT `TransmitFile` function [1]).

6.2 Drawbacks

The Proactor pattern has the following drawbacks:

Hard to debug: Applications written with the Proactor pattern can be hard to debug since the inverted flow of control oscillates between the framework infrastructure and the method callbacks on application-specific handlers. This increases the difficulty of "single-stepping" through the runtime behavior of a framework within a debugger since application developers may not understand or have access to the framework code. This is similar to the problems encountered trying to debug a compiler's lexical analyzer and parser written with `LEX` and `YACC`. In these applications, debugging is straightforward when the thread of control is within the user-defined action routines. Once the thread of control returns to

[4] Some older operating system exhibit this behavior; most modern operating systems don't.

the generated Deterministic Finite Automata (DFA) skeleton, however, it is hard to follow the program logic.

Scheduling and controlling outstanding operations: `Proactive Initiators` may have no control over the order in which `Asynchronous Operations` are executed. Therefore, the `Asynchronous Operation Processor` must be designed carefully to support prioritization and cancellation of `Asynchronous Operations`.

7 Implementation

The Proactor pattern can be implemented in many ways. This section discusses the steps involved in implementing the Proactor pattern.

7.1 Implementing the Asynchronous Operation Processor

The first step in implementing the Proactor pattern is building the `Asynchronous Operation Processor`. The `Asynchronous Operation Processor` is responsible for executing operations asynchronously on behalf of applications. As a result, its two primary responsibilities are exporting `Asynchronous Operation` APIs and implementing an `Asynchronous Operation Engine` to do the work.

7.1.1 Define Asynchronous Operation APIs

The `Asynchronous Operation Processor` must provide an API that allows applications to request `Asynchronous Operations`. There are several forces to be considered when designing these APIs:

Portability: The APIs should not tie an application nor its `Proactve Initiators` to a particular platform.

Flexibility: Often, asynchronous APIs can be shared for many types of operations. For instance, asynchronous I/O operations can often be used to perform I/O on multiple mediums (such as network and files). It may be beneficial to design APIs that support such reuse.

Callbacks: The `Proactive Initiators` must register a callback when the operation is invoked. A common approach to implement callbacks is to have the calling objects (clients) export an interface known by the caller (server). Therefore, `Proactive Initiators` must inform the `Asynchronous Operation Processor` which `Completion Handler` should be called back when an operation completes.

Completion Dispatcher: Since an application may use multiple `Completion Dispatchers`, the `Proactive Initiator` also must indicate which `Completion Dispatcher` should perform the callback.

Given all of these concerns, consider the following API for asynchronous reads and writes. The `Asynch_Stream` class is a factory for initiating asynchronous reads and writes. Once constructed, multiple asynchronous reads and writes can be started using this class. An `Asynch_Stream::Read_Result` will be passed back to the `handler` when the asynchronous read completes via the `handle_read` callback on the `Completion_Handler`. Similarly, an `Asynch_Stream::Write_Result` will be passed back to the `handler` when the asynchronous write completes via the `handle_write` callback on `Completion_Handler`.

```
class Asynch_Stream
  // = TITLE
  //    A Factory for initiating reads
  //    and writes asynchronously.
{
  // Initializes the factory with information
  // which will be used with each asynchronous
  // call.  <handler> is notified when the
  // operation completes. The asynchronous
  // operations are performed on the <handle>
  // and the results of the operations are
  // sent  to the <Completion_Dispatcher>.
  Asynch_Stream (Completion_Handler &handler,
                 HANDLE handle,
                 Completion_Dispatcher *);

  // This starts off an asynchronous read.
  // Upto <bytes_to_read> will be read and
  // stored in the <message_block>.
  int read (Message_Block &message_block,
            u_long bytes_to_read,
            const void *act = 0);

  // This starts off an asynchronous write.
  // Upto <bytes_to_write> will be written
  // from the <message_block>.
  int write (Message_Block &message_block,
             u_long bytes_to_write,
             const void *act = 0);
  ...
};
```

7.1.2 Implement the Asynchronous Operation Engine

The `Asynchronous Operation Processor` must contain a mechanism that performs the operations asynchronously. In other words, when an application thread invokes an `Asynchronous Operation`, the operation must be performed without borrowing the application's thread of control. Fortunately, modern operating systems provide mechanisms for `Asynchronous Operations` (for example, POSIX asynchronous I/O and WinNT overlapped I/O). When this is the case, implementing this part of the pattern simply requires mapping the platform APIs to the `Asynchronous Operation` APIs described above.

If the OS platform does not provide support for `Asynchronous Operations`, there are several implementation techniques that can be used to build an `Asynchronous Operation Engine`. Perhaps the most intuitive solution is to use dedicated threads to perform the `Asynchronous Operations` for applications.

To implement a threaded Asynchronous Operation Engine, there are three primary steps:

1. Operation invocation: Because the operation will be performed in a different thread of control from the invoking application thread, some type of thread synchronization must occur. One approach would be to spawn a thread for each operation. A more common approach is for the Asynchronous Operation Processor to control a pool of dedicated threads. This approach would require that the application thread queue the operation request before continuing with other application computations.

2. Operation execution: Since the operation will be performed in a dedicated thread, it can perform "blocking" operations without directly impeding progress of the application. For instance, when providing a mechanism for asynchronous I/O reads, the dedicated thread can block while reading from socket or file handles.

3. Operation completion: When the operation completes, the application must be notified. In particular, the dedicated thread must delegate application-specific notifications to the Completion Dispatcher. This will require additional synchronization between threads.

7.2 Implementing the Completion Dispatcher

The Completion Dispatcher calls back to the Completion Handler that is associated with the application objects when it receives operation completions from the Asynchronous Operation Processor. There are two issues involved with implementing the Completion Dispatcher: (1) implementing callbacks and (2) defining the concurrency strategy used to perform the callbacks.

7.2.1 Implementing Callbacks

The Completion Dispatcher must implement a mechanism through which Completion Handlers are invoked. This requires Proactive Initiators to specify a callback when initiating operations. The following are common callback alternatives:

Callback class: The Completion Handler exports an interface known by the Completion Dispatcher. The Completion Dispatcher calls back on a method in this interface when the operation completes and passes it information about the completed operation (such as the number of bytes read from the network connection).

Function pointer: The Completion Dispatcher invokes the Completion Handler via a callback function pointer. This approach effectively breaks the knowledge dependency between the Completion Dispatcher and the Completion Handler. This has two benefits:

1. The Completion Handler is not forced to export a specific interface; and
2. There is no need for compile-time dependencies between the Completion Dispatcher and the Completion Handler.

Rendezvous: The Proactive Initiator can establish an event object or a condition variable, which serves as a rendezvous between the Completion Dispatcher and the Completion Handler. This is most common when the Completion Handler is the Proactive Initiator. While the Asynchronous Operation runs to completion, the Completion Handler processes other activity. Periodically, the Completion Handler will check at the rendezvous point for completion status.

7.2.2 Defining Completion Dispatcher Concurrency Strategies

A Completion Dispatcher will be notified by the Asynchronous Operation Processor when operations completes. At this point, the Completion Dispatcher can utilize one of the following concurrency strategies to perform the application callback:

Dynamic-thread dispatching: A thread can be dynamically allocated for each Completion Handler by the Completion Dispatcher. Dynamic-thread dispatching can be implemented with most multi-threaded operating systems. On some platforms, this may be the least efficient technique of those listed for Completion Dispatcher implementations due to the overhead of creating and destroying thread resources.

Post-reactive dispatching: An event object or condition variable established by the Proactive Initiator can be signaled by the Completion Dispatcher. Although polling and spawning a child thread that blocks on the event object are options, the most efficient method for Post-reactive dispatching is to register the event with a Reactor. Post-reactive dispatching can be implemented with aio_suspend in POSIX real-time environments and with WaitForMultipleObjects in Win32 environments.

Call-through dispatching: The thread of control from the Asynchronous Operation Processor can be borrowed by the Completion Dispatcher to execute the Completion Handler. This "cycle stealing" strategy can increase performance by decreasing the incidence of idle threads. In the cases where older operating systems will context switch to idle threads just to switch back out of them, this approach has a great potential of reclaiming "lost" time.

Call-through dispatching can be implemented in Windows NT using the ReadFileEx and WriteFileEx Win32 functions. For example, a thread of control can use these calls to wait on a semaphore to become signaled. When it waits, the thread informs the OS that it is entering into a special state known as an "alertable wait state." At this point, the OS can seize control of the waiting thread of control's stack and associated resources in order to execute the Completion Handler.

Thread Pool dispatching: A pool of threads owned by the `Completion Dispatcher` can be used for `Completion Handler` execution. Each thread of control in the pool has been dynamically allocated to an available CPU. Thread pool dispatching can be implemented with Windows NT's I/O Completion Ports.

When considering the applicability of the `Completion Dispatcher` techniques described above, consider the possible combinations of OS environments and physical hardware shown in Table 1.

Threading model	System Type	
	Single-processor	Multi-processor
`Single-threaded`	A	B
`Multi-threaded`	C	D

Table 1: `Completion Dispatcher` Concurrency Strategies

If your OS only supports synchronous I/O, then refer to the Reactor pattern [5]. However, most modern operating systems support some form of asynchronous I/O.

In combination A and B from Table 1, the Post-reactive approach to asynchronous I/O is probably the best, assuming you are not waiting on any semaphores or mutexes. If you are, a Call-through implementation may be more responsive. In combination C, use a Call-through approach. In combination D, use a Thread Pool approach. In practice, systematic empirical measurements are necessary to select the most appropriate alternative.

7.3 Implementing Completion Handlers

The implementation of `Completion Handlers` raises the following concerns.

7.3.1 State Integrity

A `Completion Handler` may need to maintain state information concerning a specific request. For instance, the OS may notify the Web Server that only part of a file was written to the network communication port. As a result, a `Completion Handler` may need to reissue the request until the file is fully written or the connection becomes invalid. Therefore, it must know the file that was originally specified, how many bytes are left to write, and what was the file pointer position at the start of the previous request.

There is no implicit limitation that prevents `Proactive Initiators` from assigning multiple `Asynchronous Operation` requests to a single `Completion Handler`. As a result, the `Completion Handler` must tie request-specific state information throughout the chain of completion notifications. To do this, `Completion Handlers` can utilize the Asynchronous Completion Token pattern [8].

7.3.2 Resource Management

As with any multi-threaded environment, the Proactor pattern does not alleviate `Completion Handlers` from ensuring that access to shared resources is thread-safe. However, a `Completion Handler` must not hold onto a shared resource across multiple completion notifications. If it does, it risks invoking the dining philosopher's problem [11].

This problem is the deadlock that results when a logical thread of control waits forever for a semaphore to become signaled. This is illustrated by imagining a dinner party attended by a group of philosophers. The diners are seated around a circular table with exactly one chop stick between each philosopher. When a philosopher becomes hungry, he must obtained the chop stick to his left and to his right in order to eat. Once philosophers obtain a chop stick, they will not release it until their hunger is satisfied. If all philosophers pick up the chop stick on their right, a deadlock occurs because the chop stick on the left will never become available.

7.3.3 Preemptive Policy

The `Completion Dispatcher` type determines if a `Completion Handler` can be preemptive while executing. When attached to Dynamic-thread and Thread Pool dispatchers, `Completion Handlers` are naturally preemptive. However, when tied to a Post-reactive `Completion Dispatcher`, `Completion Handlers` are not preemptive with respect to each other. When driven by a Call-through dispatcher, the `Completion Handlers` are not preemptive with respect to the thread-of-control that is in the alertable wait state.

In general, a handler should not perform long-duration synchronous operations unless multiple completion threads are used since this will significantly decrease the overall responsiveness of the application. This risk can be alleviated by increased programming discipline. For instance, all `Completion Handlers` are required to act as `Proactive Initiators` instead of executing synchronous operations.

8 Sample Code

This section shows how to use the Proactor pattern to develop a Web server. The example is based on the Proactor pattern implementation in the ACE framework [4].

When a client connects to the Web server, the `HTTP_Handler`'s `open` method is called. The server then initializes the asynchronous I/O object with the object to callback when the `Asynchronous Operation` completes (which in this case is `this`), the network connection for transferring the data, and the `Completion Dispatcher` to be used once the operation completes (`proactor_`). The read operation is then started asynchronously and the server returns to the event loop.

The `HTTP_Handler::handle_read_stream` is called back by the dispatcher when the `Async read` op-

eration completes. If there is enough data, the client request is then parsed. If the entire client request has not arrived yet, another read operation is initiated asynchronously.

In response to a `GET` request, the server memory-maps the requested file and writes the file data asynchronously to the client. The dispatcher calls back on `HTTP_Handler::handle_write_stream` when the write operation completes, which frees up dynamically allocated resources.

The Appendix contains two other code examples for implementing the Web server using a synchronous threaded model and a synchronous (non-blocking) reactive model.

```
class HTTP_Handler
  : public Proactor::Event_Handler
  // = TITLE
  //    Implements the HTTP protocol
  //    (asynchronous version).
  //
  // = PATTERN PARTICIPANTS
  //    Proactive Initiator   = HTTP_Handler
  //    Asynch Op             = Network I/O
  //    Asynch Op Processor   = OS
  //    Completion Dispatcher = Proactor
  //    Completion Handler    = HTPP_Handler
{
public:
   void open (Socket_Stream *client)
   {
     // Initialize state for request
     request_.state_ = INCOMPLETE;

     // Store reference to client.
     client_ = client;

     // Initialize asynch read stream
     stream_.open (*this,
                   client_->handle (),
                   proactor_);

     // Start read asynchronously.
     stream_.read (request_.buffer (),
                   request_.buffer_size ());
   }

   // This is called by the Proactor
   // when the asynch read completes
   void handle_read_stream
     (u_long bytes_transferred)
   {
     if (request_.enough_data
                 (bytes_transferred))
       parse_request ();
     else
       // Start reading asynchronously.
       stream_.read (request_.buffer (),
                     request_.buffer_size ());
   }

   void parse_request (void)
   {
     // Switch on the HTTP command type.
     switch (request_.command ()) {
     // Client is requesting a file.
     case HTTP_Request::GET:
       // Memory map the requested file.
       file_.map (request_.filename ());

       // Start writing asynchronously.
       stream_.write (file_.buffer (),
                      file_.buffer_size ());
       break;

     // Client is storing a file
     // at the server.
     case HTTP_Request::PUT:
        // ...
     }
   }

   void handle_write_stream
         (u_long bytes_transferred)
   {
     if (file_.enough_data
           (bytes_transferred))
       // Success....
     else
       // Start another asynchronous write
       stream_.write (file_.buffer (),
                      file_.buffer_size ());
   }

private:
   // Set at initialization.
   Proactor *proactor_;

   // Memory-mapped file_;
   Mem_Map file_;

   // Socket endpoint.
   Socket_Stream *client_;

   // HTTP Request holder
   HTTP_Request request_;

   // Used for Asynch I/O
   Asynch_Stream stream_;
};
```

9 Known Uses

The following are some widely documented uses of the Proactor pattern:

I/O Completion Ports in Windows NT: The Windows NT operating system implements the Proactor pattern. Various `Asynchronous Operations` such as accepting new network connections, reading and writing to files and sockets, and transmission of files across a network connection are supported by Windows NT. The operating system is the `Asynchronous Operation Processor`. Results of the operations are queued up at the I/O completion port (which plays the role of the `Completion Dispatcher`).

The UNIX AIO Family of Asynchronous I/O Operations: On some real-time POSIX platforms, the Proactor pattern is implemented by the `aio` family of APIs [9]. These OS features are very similar to the ones described above for Windows NT. One difference is that UNIX signals can be used to implement an truly asynchronous `Completion Dispatcher` (the Windows NT API is not truly asynchronous).

ACE Proactor: The Adaptive Communications Environment (ACE) [4] implements a Proactor component that encapsulates I/O Completion Ports on Windows NT and the

`aio` APIs on POSIX platforms. The ACE Proactor abstraction provides an OO interface to the standard C APIs supported by Windows NT. The source code for this implementation can be acquired from the ACE website at `www.cs.wustl.edu/~schmidt/ACE.html`.

Asynchronous Procedure Calls in Windows NT: Some systems (such as Windows NT) support Asynchronous Procedure Calls (APC)s. An APC is a function that executes asynchronously in the context of a particular thread. When an APC is queued to a thread, the system issues a software interrupt. The next time the thread is scheduled, it will run the APC. APCs made by operating system are called *kernel-mode* APCs. APCs made by an application are called *user-mode* APCs.

10 Related Patterns

Figure 9 illustrates patterns that are related to the Proactor.

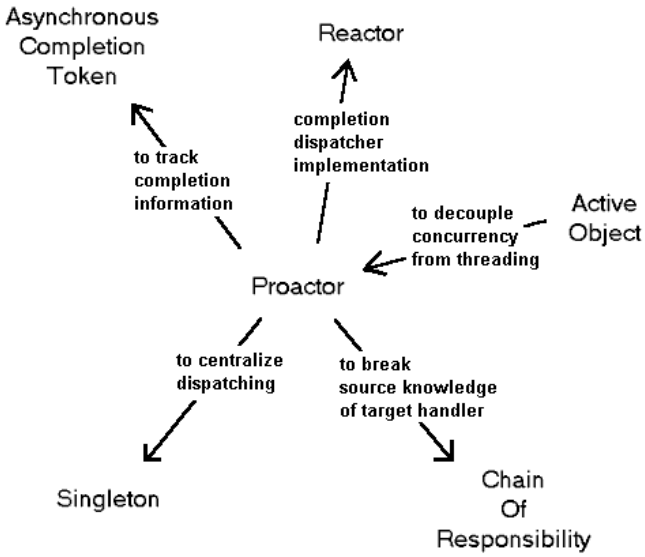

Figure 9: Proactor Pattern's Related Patterns

The Asynchronous Completion Token (ACT) pattern [8] is generally used in conjunction with the Proactor pattern. When `Asynchronous Operations` complete, applications may need more information than simply the notification itself to properly handle the event. The Asynchronous Completion Token pattern allows applications to efficiently associate state with the completion of `Asynchronous Operations`.

The Proactor pattern is related to the Observer pattern [12] (where dependents are updated automatically when a single subject changes). In the Proactor pattern, handlers are informed automatically when events from multiple sources occur. In general, the Proactor pattern is used to asynchronously demultiplex multiple sources of input to their associated event handlers, whereas an Observer is usually associated with only a single source of events.

The Proactor pattern can be considered an *asynchronous* variant of the synchronous Reactor pattern [5]. The Reactor pattern is responsible for demultiplexing and dispatching of multiple event handlers that are triggered when it is possible to *initiate* an operation *synchronously* without blocking. In contrast, the Proactor supports the demultiplexing and dispatching of multiple event handlers that are triggered by the *completion* of *asynchronous* events.

The Active Object pattern [13] decouples method execution from method invocation. The Proactor pattern is similar because `Asynchronous Operation Processors` perform operations on behalf of application `Proactive Initiators`. That is, both patterns can be used to implement `Asynchronous Operations`. The Proactor pattern is often used in place of the Active Object pattern to decouple the systems concurrency policy from the threading model.

A Proactor may be implemented as a Singleton [12]. This is useful for centralizing event demultiplexing and completion dispatching into a single location within an asynchronous application.

The Chain of Responsibility (COR) pattern [12] decouples event handlers from event sources. The Proactor pattern is similar in its segregation of `Proactive Initiators` and `Completion Handlers`. However, in COR, the event source has no prior knowledge of which handler will be executed, if any. In Proactor, `Proactive Initiators` have full disclosure of the target handler. However, the two patterns can be combined by establishing a `Completion Handler` that is the entry pont into a responsibility chain dynamically configured by an external factory.

11 Concluding Remarks

The Proactor pattern embodies a powerful design paradigm that supports efficient and flexible event dispatching strategies for high-performance concurrent applications. The Proactor pattern provides the performance benefits of executing operations concurrently, without constraining the developer to synchronous multi-threaded or reactive programming.

References

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

[2] J. Hu, I. Pyarali, and D. C. Schmidt, "Applying the Proactor Pattern to High-Performance Web Servers," in *Proceedings of the 10th International Conference on Parallel and Distributed Computing and Systems*, IASTED, Oct. 1998.

[3] J. C. Mogul, "The Case for Persistent-connection HTTP," in *Proceedings of ACM SIGCOMM '95 Conference in Computer*

Communication Review, (Boston, MA, USA), pp. 299–314, ACM Press, August 1995.

[4] 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.

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

[7] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley, 1996.

[8] 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.

[9] "Information Technology – Portable Operating System Interface (POSIX) – Part 1: System Application: Program Interface (API) [C Language]," 1995.

[10] *Microsoft Developers Studio, Version 4.2 - Software Development Kit*, 1996.

[11] E. W. Dijkstra, "Hierarchical Ordering of Sequential Processes," *Acta Informatica*, vol. 1, no. 2, pp. 115–138, 1971.

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

[13] R. G. Lavender and D. C. Schmidt, "Active Object: an Object Behavioral Pattern for Concurrent Programming," in *Proceedings of the 2^{nd} Annual Conference on the Pattern Languages of Programs*, (Monticello, Illinois), pp. 1–7, September 1995.

A Alternative Implementations

This Appendix outlines the code used to develop alternatives to the Proactor pattern. Below, we examine both synchronous I/O using multi-threading and reactive I/O using single-threading.

A.1 Multiple Synchronous Threads

The following code shows how to use synchronous I/O with a pool of threads to develop a Web server. When a client connects to the server a thread in the pool accepts the connection and calls the `open` method in class `HTTP_Handler`. The server then synchronously reads the request from the network connection. When the read operation completes, the client request is then parsed. In response to a `GET` request, the server memory-maps the requested file and writes the file data synchronously to the client. Note how blocking I/O allows the Web server to follow the steps outlined in Section 2.2.1.

```
class HTTP_Handler
  // = TITLE
  //    Implements the HTTP protocol
  //    (synchronous threaded version).
  //
  // = DESCRIPTION
  //    This class is called by a
  //    thread in the Thread Pool.
{
public:
  void open (Socket_Stream *client)
  {
    HTTP_Request request;

    // Store reference to client.
    client_ = client;

    // Synchronously read the HTTP request
    // from the network connection and
    // parse it.
    client_->recv (request);

    parse_request (request);
  }

  void parse_request (HTTP_Request &request)
  {
    // Switch on the HTTP command type.
    switch (request.command ())
    {
      // Client is requesting a file.
      case HTTP_Request::GET:
        // Memory map the requested file.
        Mem_Map input_file;
        input_file.map (request.filename());

        // Synchronously send the file
        // to the client.  Block until the
        // file is transferred.
        client_->send (input_file.data (),
                       input_file.size ());
        break;

        // Client is storing a file at
        // the server.
        case HTTP_Request::PUT:
          // ...
     }
  }

private:
  // Socket endpoint.
  Socket_Stream *client_;

  // ...
};
```

A.2 Single-threaded Reactive Event Dispatching

The following code shows the use of the Reactor pattern to develop a Web server. When a client connects to the server, the `HTTP_Handler::open` method is called. The server registers the I/O handle and the object to callback (which in this case is `this`) when the network handle is "ready for reading." The server returns to the event loop.

When the request data arrives at the server, the `reactor_` calls back the `HTTP_Handler::handle_input` method. The client data is read in a non-blocking manner. If there is

enough data, the client request is parsed. If the entire client request has not yet arrived, the application returns to the reactor event loop.

In response to a GET request, the server memory maps the requested file and registers with the reactor to be notified when the network connection becomes "ready for writing." The reactor_ then calls back on HTTP_Handler::handle_output method when writing data to the connection would not blocking the calling thread. When all the data has been sent to the client, the network connection is closed.

```
class HTTP_Handler :
  public Reactor::Event_Handler
  // = TITLE
  //    Implements the HTTP protocol
  //    (synchronous reactive version).
  //
  // = DESCRIPTION
  //    The Event_Handler base class
  //    defines the hooks for
  //    handle_input()/handle_output().
  //
  // = PATTERN PARTICIPANTS
  //    Reactor       = Reactor
  //    Event Handler = HTTP_Handler
{
public:
  void open (Socket_Stream *client)
  {
    // Initialize state for request
    request_.state_ = INCOMPLETE;

    // Store reference to client.
    client_ = client;

    // Register with the reactor for reading.
    reactor_->register_handler
      (client_->handle (),
       this,
       Reactor::READ_MASK);
  }

  // This is called by the Reactor when
  // we can read from the client handle.
  void handle_input (void)
  {
    int result = 0;

    // Non-blocking read from the network
    // connection.
    do
      result = request_.recv (client_->handle ());
    while (result != SOCKET_ERROR
           && request_.state_ == INCOMPLETE);

    // No more progress possible,
    // blocking will occur
    if (request_.state_ == INCOMPLETE
        && errno == EWOULDBLOCK)
      reactor_->register_handler
        (client_->handle (),
         this,
         Reactor::READ_MASK);
    else
      // We now have the entire request
      parse_request ();
  }

  void parse_request (void)
  {
    // Switch on the HTTP command type.
    switch (request_.command ()) {
    // Client is requesting a file.
    case HTTP_Request::GET:
      // Memory map the requested file.
      file_.map (request_.filename ());

      // Transfer the file using Reactive I/O.
      handle_output ();
      break;

      // Client is storing a file at
      // the server.
      case HTTP_Request::PUT:
      // ...
    }
  }

  void handle_output (void)
  {
    // Asynchronously send the file
    // to the client.
    if (client_->send (file_.data (),
                       file_.size ())
                       == SOCKET_ERROR
        && errno == EWOULDBLOCK)
      // Register with reactor...
    else
      // Close down and release resources.
      handle_close ();
  }

private:
  // Set at initialization.
  Reactor *reactor_;

  // Memory-mapped file_;
  Mem_Map file_;

  // Socket endpoint.
  Socket_Stream *client_;

  // HTTP Request holder.
  HTTP_Request request_;
};
```