Proactor

An Architectural 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[1]
St. Louis, MO 63130, (314) 935-7538

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

1 Intent

The Proactor architecture pattern demultiplexes and dispatches service requests that that are triggered by the completion of asynchronous operations.

2 Example

Consider a networking server that must perform multiple operations concurrently. For example, a high-performance Web server must concurrently process HTTP requests sent from multiple clients [1, 2], as shown in the following figure. When a user wants to download the content at a URL,

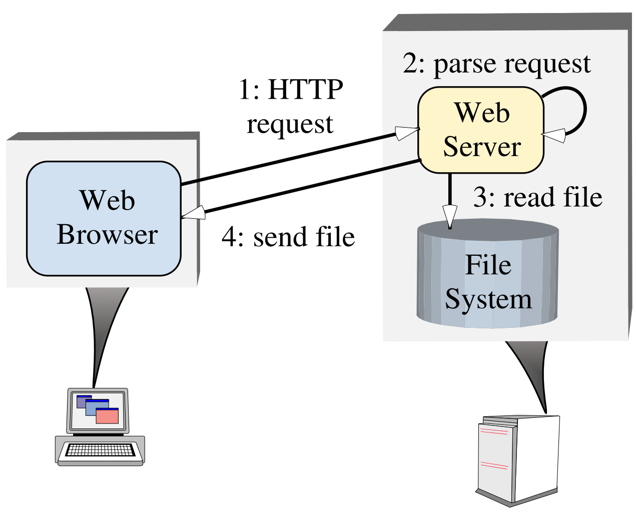


Figure 1: Typical Web Server Communication Software Architecture

the browser establishes a connection to the Web server and sends an HTTP GET request. The server subsequently accepts the client's connection and reads the request. It then parses and validates the request, sends the specified file(s) back to the browser, and closes the connection.

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

One way to implement a concurrent Web server is to use *synchronous multi-threading*. In this model, a new server thread is spawned to process each HTTP GET request [3], *i.e.*, one thread for each client request. Each thread performs connection establishment, HTTP request reading, request parsing, and file transfer operations *synchronously*, *i.e.*, operations block until they complete.

Although synchronous multi-threading is a common concurrency model, it has the following drawbacks:

- *Increased synchronization complexity* – Threading may require complex concurrency control schemes throughout Web server code [4].
- *Increased performance overhead* – Threading may lead to poor performance due to context switching, synchronization, and data movement among CPUs [5];
- *Non-portability* – Threading may not be available on all operating system platforms.
- *Threading policy is tightly coupled to the concurrency policy* – A concurrent server 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.

As a result of these drawbacks, synchronous multi-threading can lead to unnecessarily complex and inefficient concurrent Web servers. Yet Web servers must handle multiple requests concurrently to ensure adequate quality of service for their clients.

2.1 Context

[Frank, after a lot of thought, I think it's too confusing not to use the term "event-driven *server*" here. Although the Proactor pattern can be used for "client" applications, it's still playing the role of a "server" in this context. I recommend we also make this minor change this in the Reactor pattern's Context and Problem statements!]

An event-driven server that receives and processes events from one or more clients concurrently.

3 Problem

Event-driven servers in a distributed system must handle one or more clients that send them service requests. Within a server, each client request is typically associated with a specific event that arrives asynchronously. For example, our Web server receives CONNECT and GET events asynchronously from its clients. Before invoking a specific service, an event-driven server must demultiplex and dispatch each asynchronous event to its corresponding service handler. Addressing this problem effectively requires the resolution of the following *forces*:

- *Concurrency* – The server must process multiple asynchronous 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 [6], should incur minimal maintenance costs.

3.1 Solution

Integrate the asynchronous demultiplexing of events with their dispatching to service handlers that process these events. In addition, decouple these general-purpose event demultiplexing and dispatching mechanisms from the application-specific dispatching of events to services within the service handlers.

[Frank, I recommend we also use the term "server" here instead of "application" for the Reactor pattern!]

For each service offered by a server, introduce a separate `Completion Handler` to process events containing the results of operations that are initiated asynchronously. All `Completion Handlers` implement the same interface. A `Proactive Initiator` invokes an `Asynchronous Operation`, passing along a `Completion Handler` and a `Proactor` to perform the demultiplexing and dispatching when the `Asynchronous Operation` completes. An `Asynchronous Operation Processor` then performs the `Asynchronous Operation`. When the operation completes, the `Asynchronous Operation Processor` notifies the `Proactor`, which calls back to the `Completion Handler` associated with the `Asynchronous Operation`. The `Completion Handler` then dispatches the notification event to a hook method that implements the requested service.

4 Structure

The key participants in the Proactor pattern include the following:

Handles

- Identify resources that are managed by an operating system. These resources commonly include network connections, open files, timers, and synchronization objects.

  The Web server creates a separate socket `Handle` for each client connection. The types of notifications that can occur on these `Handles` include asynchronous CONNECTION completion, asynchronous READ completion, and asynchronous WRITE completion.

Asynchronous Operation

- These operations are used to execute requests, such as writing and writing data through a socket `Handle`, *asynchronously* on behalf of a server. When a server invokes an `Asynchronous Operation`, the operation is performed *without* borrowing the server's thread of control.[2] Thus, from the server's perspective, the operations execute asynchronously.

  The types of `Asynchronous Operations` that can be used in a Web server include the initiation of asynchronous `connects`, asynchronous `reads`, and asynchronous `writes`.

Completion Handler

- Specifies an interface consisting of one or more hook methods [8] that abstractly represent the `Asynchronous Operation` completion handling for service-specific notifications. Hook methods must be implemented by application-specific services.

Concrete Completion Handler

- Implements the hook method(s) that process certain types of completion notification events from a `Proactor` in an application-specific manner. There are two `Concrete Completion Handlers` in the Web server: `HTTP Handler` and `HTTP Acceptor`. The `HTTP Handler` is responsible for receiving, processing, and replying to client requests. The `HTTP Acceptor` creates and connects `HTTP Handlers` that process the subsequent client request.

Asynchronous Operation Processor

- `Asynchronous Operations` are run to completion by an `Asynchronous Operation Processor`, which is typically implemented by the operating system kernel. When an asynchronously initiated operation completes, the `Asynchronous Operation Processor` delegates the subsequent completion dispatch handling to the appropriate `Proactor`.

[2] In contrast, the event dispatching model in the Reactor pattern [7] steals the server's thread of control to perform the operation synchronously.

Proactor

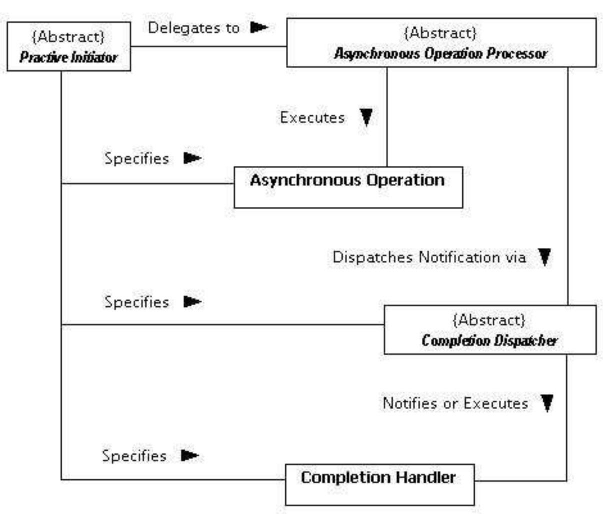
- The `Proactor` is a completion dispatcher that calls back to the designated `Concrete Completion Handler` in a server after the corresponding `Asynchronous Operation` has been executed. The `Proactor` decouples higher-level `Completion Handler` dispatching strategies from the lower-level operation execution mechanisms used by the `Asynchronous Operation Processor`.

Proactive Initiator

- A `Proactive Initiator` is any entity in an server that initiates an `Asynchronous Operation`. The `Proactive Initiator` registers a `Completion Handler` and a `Proactor` with a `Asynchronous Operation Processor`.

  For instance, the `Proactive Initiator` in the Web server is its thread(s) of control. These thread(s) initiate asynchronous `accept`, `read`, and `write` operations to process client HTTP GET requests.

The structure of the participants in the Proactor pattern is illustrated in the following UML diagram.



5 Dynamics

[Frank, can you please trim this down similar to the Reactor pattern intro?!] There are several well-defined steps that occur for all `Asynchronous Operations`, *e.g.*, at a high level of abstraction, servers initiate operations asynchronously and are notified when the operations complete. Figure 2 shows the following interactions that must occur between the pattern participants:

1. Proactive Initiators initiates operation: To perform an operation asynchronously, a server initiates the operation on the `Asynchronous Operation Processor`.

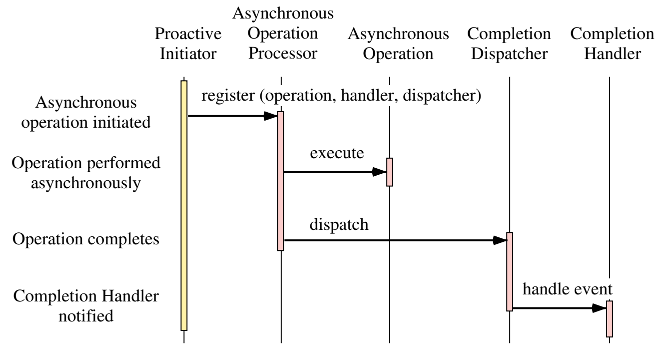


Figure 2: Interaction Diagram for the Proactor Pattern

The server must specify which `Proactor` should perform the callback on which `Completion Handler` when the `Asynchronous Operation` completes.

For instance, a Web server might instruct the operating system to transmit a file across the network using a particular socket `Handle`. To request such an operation, the Web server must specify which file and which connected socket `Handle` to use, as well as which `Completion Handler`, such as an `HTTP Handler`, the designated `Proactor` should callback when the asynchronous file transfer operation completes.

2. Asynchronous Operation Processor performs operation: When the server invokes operations on the `Asynchronous Operation Processor` they run asynchronously with respect to other server operations.

3. The Asynchronous Operation Processor notifies the Proactor: When operations complete, the `Asynchronous Operation Processor` retrieves the `Completion Handler` and `Proactor` that were specified by the server when the operation was initiated. The `Asynchronous Operation Processor` then passes the `Proactor` both 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 connected socket `Handle`.

4. Proactor notifies the server: The `Proactor` calls the completion hook on the `Completion Handler`, passing it any completion data specified by the server.

For instance, if an asynchronous `read` invoked by our Web server completes successfully, its `HTTP Handler` will be passed a pointer to the newly arrived data.

6 Implementation

This section discusses the steps involved to implement the Proactor pattern.

6.1 Implement the Asynchronous Operation Processor

The first step in implementing the Proactor pattern is to develop its `Asynchronous Operation Processor`. An `Asynchronous Operation Processor` is responsible for executing operations asynchronously on behalf of servers. As a result, its two primary responsibilities are exporting `Asynchronous Operation` APIs and implementing a mechanism for processing operations asynchronously.

6.1.1 Define the Asynchronous Operation APIs

The `Asynchronous Operation Processor` provides a facade [9] that allows servers to invoke `Asynchronous Operations`. There are several forces to be considered when designing a API for the `Asynchronous Operation Processor` facade:

Portability: The APIs should not tie a server or its `Proactive Initiators` to a particular platform.

Flexibility: Often, asynchronous APIs can be shared for many types of operations. For instance, asynchronous I/O operations often can be used to perform I/O on multiple I/O devices, 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 callback implementation is to have the calling objects, i.e., the clients, export an interface known by the caller, *i.e.*, the server. Therefore, `Proactive Initiators` must inform the `Asynchronous Operation Processor` which `Completion Handler` to call back when an operation completes.

Proactor: A server can use multiple `Proactors`. Therefore, the `Proactive Initiator` also must indicate which `Proactor` should perform the callback.

[Frank, here's a good place for a =¿! For example, consider the following class that performs asynchronous `read` and `write` operations.

```
class Asynch_Stream
{
  // = TITLE
  //    A Factory for initiating reads and writes asynchronously.
public:
  // 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 <Proactor>.
  Asynch_Stream (Completion_Handler *handler,
                 HANDLE handle,
                 Proactor *);

  // 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);

  class Read_Result {
  // Frank, this is a place holder!
  };

  class Write_Result
  {
  // Frank, this is a place holder!
  };
};
```

The `Asynch_Stream` class is a general-purpose factory for initiating `read` and `write` operations asynchronously. After it's constructed, multiple asynchronous `reads` and `writes` can be initiated by using this class.

[Frank, this is just a placeholder – we need to get help from Irfan to explain how this works.]

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`. Likewise, an `Asynch_Stream::Write_Result` will be passed back to the `handler` when the asynchronous `write` completes via the `handle_write` callback on the `Completion_Handler`.

6.1.2 Implement the Asynchronous Operation Processing Mechanism

The `Asynchronous Operation Processor` contains a mechanism that executes operations asynchronously. In other words, when a server thread invokes an `Asynchronous Operation`, the operation must be performed *without* borrowing the server's thread of control. Many operating systems provide mechanisms for `Asynchronous Operations`, *e.g.*, 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 operating system 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 servers. To implement a threaded `Asynchronous Operation Engine`, there are three primary steps:

1. Operation invocation: The operation will be performed in a different thread of control from the invoking server thread. Therefore, some type of thread synchronization must occur. One approach is to spawn a thread for each operation. A more efficient approach, however, is for the

Asynchronous Operation Processor to control a pool of dedicated threads. This approach requires the server thread to queue the operation request before continuing with other server computations.

2. Operation execution: Each operation will be performed in a dedicated thread. Thus, operations can "block" without directly impeding progress of the server. 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 server must be notified. In particular, the dedicated thread must delegate application-specific notifications to the Proactor. This requires additional synchronization between threads.

6.2 Implement the Proactor

The Proactor calls back to the Completion Handler that is associated with the server objects when it receives operation completions from the Asynchronous Operation Processor. A Proactor is often implemented as a Singleton [9], which is useful for centralizing event demultiplexing and completion dispatching into a single location within an asynchronous application.

For example, the following is an interface for the Proactor:

```
class Proactor
{
public:
  // Entry point into the proactive event loop.
  int handle_events (Time_Value *wait_time = 0);

private:
};
```

The handle_events method is the entry point into a Proactor's event loop. This method will block until it dispatches a single notification to a Completion Handler.

Below, we describe three issues involved with implementing the Proactor: demultiplexing completion notifications to the appropriate Concrete Completion Handler, implementing a completion handler dispatching mechanism, and defining the concurrency strategy used to perform the callbacks.

6.2.1 Determine How to Demultiplex Completion Notifications to Completion Handlers

[Frank, this is a placeholder for a more in-depth discussion of this point. I'll fill this in later with the help from Irfan!]

When Asynchronous Operations complete, the associated Proactor may need more information than simply the notification itself to properly handle the event. The Asynchronous Completion Token (ACT) pattern [10] allows a Proactor to efficiently demultiplex the completion of Asynchronous Operations with their associated Concrete Completion Handlers.

6.3 Implement the Completion Handler Dispatching Mechanism

The Proactor must implement a mechanism through which Concrete 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 Proactor. The Proactor calls back on a method in this interface when the operation completes and passes it information about the completed operation. For example, a Completion Handler for our Web server would pass back the number of bytes read from the network connection.

Function pointer: The Proactor invokes the Completion Handler via a callback function pointer. This approach reduces the dependency between the Proactor and the Completion Handler to a function prototype rather than an interface. The primary benefit is that the Completion Handler is not forced to export a specific class interface.

Rendezvous: The Proactive Initiator can establish an event object or a condition variable, which serves as a rendezvous between the Proactor and the Completion Handler. This approach 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.

6.3.1 Define the Proactor Concurrency Strategy

A Proactor will be notified by the Asynchronous Operation Processor when operations complete. At this point, the Proactor can utilize one of the following concurrency strategies to perform the server callback:

Dynamic-thread dispatching: A Proactor can dynamically allocated a thread for each Completion Handler. 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 Proactor implementations due to the overhead of creating and destroying thread resources.

Post-reactive dispatching: An event or condition variable established by the Proactive Initiator can be signaled by the Proactor. Although polling and spawning a child thread that blocks on an event 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: In this model, the thread of control blocked in a synchronous operation is borrowed by the Asynchronous Operation Processor to execute a Completion Handler via the Proactor. This "cycle stealing" strategy can increase performance by decreasing the incidence of idle threads. This is particularly useful when a single-threaded server wants to perform proactive I/O while still occasionally executing synchronous operations.

One way to implement call-through dispatching in Windows NT is via the Win32 functions ReadFileEx and WaitForSingleObjectEx. A thread of control can initiate an asynchronous read operation via ReadFileEx passing a Completion Handler as a parameter. After the read operation is initiated, the same thread might call the WaitForSingleObjectEx function to wait synchronously for an unrelated event to be signaled. When WaitForSingleObjectEx is called, the thread informs the operating system that it is entering into a special state known as an "alertable wait state." Therefore, when the read operation completes, the operating system can borrow the thread blocked in WaitForSingleObjectEx to dispatch the Completion Handler registered when ReadFileEx was called. In this case, the Windows NT operating system is both the Asynchronous Operation Processor and the Proactor.

Thread pool dispatching: A pool of threads owned by the Proactor 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 Proactor techniques described above, consider the possible combinations of operating system 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: Proactor Concurrency Strategies

If your operating system only supports synchronous I/O, then refer to the Reactor pattern [7]. However, most modern operating systems support alternative forms of asynchronous I/O, such as POSIX asynchronous I/O and WinNT overlapped I/O.

For single-threaded servers (combination A and B from Table 1), the correct concurrency strategy depends on the type of operations that will be performed. If a server will only be performing proactive asynchronous operations then a thread pool of size one can be used. If a server will be performing reactive synchronous operations as well as proactive asynchronous operations, then Post-reactive dispatching should be used. Lastly, if the single-threaded server needs to occasionally perform long-blocking synchronous operations, Call-through dispatching should be used.

For multi-threaded servers (combinations C and D) any of the dispatching techniques can be appropriate. Often, systematic empirical measurements are the only way to make the best selection. For instance, multi-threaded solutions running on single-processor systems can decrease performance for compute-bound servers by increasing context switching overhead. In contrast, a single-processor, multi-threaded solution can sometimes increase performance for I/O-bound servers by allowing the operating system and threading package to overlap computation and communication.

6.3.2 Define the Proactor Preemptive Policy

The type of the Proactor determines if a Completion Handler can be preempted while executing. When attached to Dynamic-thread and Thread pool dispatchers, Completion Handlers are naturally preemptive. However, when tied to a Post-reactive Proactor, 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 because this will significantly decrease the overall responsiveness of the server. 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.

6.4 Implement Completion Handlers

Completion Handlers specify an interface consisting of one or more hook methods that abstractly represent the Asynchronous Operation completion handling for service-specific notifications. Concrete Completion Handlers are defined by server programmers to provide service-specific functionality. The implementation of both abstract and concrete Completion Handlers contains the following substeps:

6.4.1 Define the Completion Handler Interface

For example, the following C++ abstract base class illustrates a multi-method interface used by a Proactor for networking events:

```
class Completion_Handler
{
public:
  // Hook methods that are called back by
  // the Proactor to handle particular types
  // of notifications.

  // Sets proactor to <p>.
```

```
  Completion_Handler (Proactor *p);

  // Virtual destruction.
  virtual ~Completion_Handler (void);

  // This method will be called when an
  // asynchronous read completes on a stream.
  virtual void handle_read_stream
    (const Asynch_Stream::Read_Result &result) = 0;
  // This method will be called when an
  // asynchronous write completes on a stream.
  virtual void handle_write_stream
    (const Asynch_Stream::Write_Result &result) = 0;
  // This method will be called when an
  // asynchronous read completes on a file.
  virtual void handle_accept
    (const Asynch_Accept::Result &result) = 0;
  // This method will be called when an
  // asynchronous transmit file completes.
  virtual void handle_time_out
    (const Time_Value &tv,
     const void *act) = 0;
  // Hook method that returns the underlying
  // I/O Handle.
  virtual Handle get_handle (void) const = 0;
};
```

[Frank, I suggest that when we convert this into Frame format that we copy/paste/edit the corresponding discussion from the Reactor pattern and make it correspond to the Proactor discussion!]

6.4.2 Determine Policies for Handling State in Concrete Completion Handlers

A `Completion Handler` may need to maintain state information concerning a specific request. For instance, the operating system 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 [10].

7 Example Resolved

The following discussion illustrates how to apply the Proactor pattern to develop a Web server that uses proactive event dispatching to handle multiple clients concurrently within one or more threads. As described in the *Structure* section, the `HTTP Handler` is responsible for receiving, processing, and replying to client requests. The `HTTP Acceptor` creates and connects `HTTP Handlers` that process the subsequent client request.

7.1 Interaction Diagrams

The following sequence of steps occur when a client connects to the Web Server:

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

The following figure illustrates these steps:

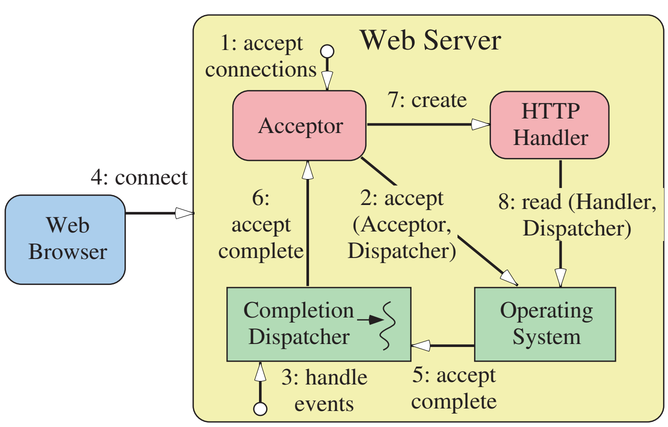


Figure 3: Client connects to a Proactor-based Web Server

The following sequence of steps is used by a proactive Web Server to service an HTTP GET request.

1. The client sends an HTTP GET request;
2. The asynchronous `read` operation completes and the operating system notifies the `Proactor`;
3. The `Proactor` notifies the `HTTP Handler`;
4. The `HTTP Handler` parses the request (steps 2 through 4 will repeat until the entire request has been received); [Frank, the following step has been changed

to reflect the code example below. Can you please make sure we update the diagram accordingly when you convert it to POSA format?!]

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

The following figure illustrates these step:

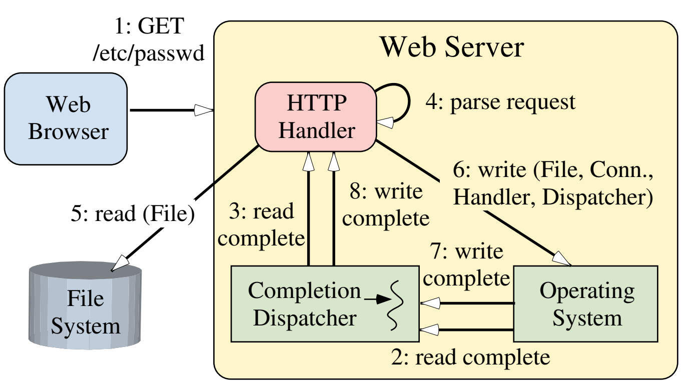


Figure 4: Client Sends requests to a Proactor-based Web Server

7.2 Sample Code

The following C++ interface illustrates how the `HTTP_Handler` can be written using the `Completion_Handler` class defined in the Implementation section.

```
class HTTP_Handler : public Completion_Handler
  // = TITLE
  //    Implements the HTTP protocol
  //    asynchronously.
  //
  // = PATTERN PARTICIPANTS
  //    Proactive Initiator   = HTTP_Handler
  //    Asynch Operation      = Network I/O
  //    Asynch Operation Processor = OS
  //    Completion Dispatcher = Proactor
  //    Completion Handler    = HTPP_Handler
{
public:
   // Constructor.
   HTTP_Handler (Proactor *);

   // Initialization hook called by an
   // Acceptor when a client connection
   // completes asynchronously.
   void open (SOCK_Stream *client);

   // This is called by the Proactor
   // when the asynch read completes.
   void handle_read_stream
     (u_long bytes_transferred);

   void parse_request (void);
   void handle_write_stream
         (u_long bytes_transferred);

private:
   // Set by the constructor.
   Proactor *proactor_;

   // Memory-mapped file_;
   Mem_Map file_;

   // Socket endpoint.
   SOCK_Stream *client_;

   // HTTP Request holder
   HTTP_Request request_;

   // Used for asynchronous socket I/O .
   Asynch_Stream stream_;
};
```

When a client connects to the Web server, the `HTTP_Handler`'s following `open` method is called by an `Acceptor`:

```
void HTTP_Handler::open (SOCK_Stream *client)
{
 // Initialize state for request
 request_.state_ = INCOMPLETE;

 // Store reference to client.
 client_ = client;

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

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

```
\noindent In this method, the server initializes the asynchronous
stream object with the {\tt Completion Handler} and {\tt Proactor} to
use when the {\tt Asynchronous Operation} completes.  The {\tt read}
operation is then started asynchronously and the server returns to its
event loop, which calls the {\tt Proactor::handle\_events} method, as
follows:

{\footnotesize
\ls{0.8}
\begin{verbatim}
while (web_server_not_shutdown)
  proactor->handle_events ();
```

When the `Async read` operation completes, the following `handle_read_stream` is called back by the `Proactor` on the `HTTP_Handler Completion Handler`:

```
void HTTP_Handler::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 ());
}
```

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 and the server returns to its event loop. Once a GET request has been received, the following `parse_request` method memory-maps the requested file and writes the file data asynchronously to the client.

```
void HTTP_Handler::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:
     // ...
  }
}
```

The `Proactor` calls back on `HTTP_Handler::handle_write_stream` when the write operation completes:

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

When all the data has been received the `Completion Handler` frees up all resources that were allocated dynamically.

8 Known Uses

The following are some well-known uses of the Proctor 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 `Proactor`.

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 [11]. These operating system 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 `Proactor`, the Windows NT API is not truly asynchronous.

ACE Proactor: The Adaptive Communications Environment (ACE) [5] 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 and POSIX platforms.

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.

9 Consequences

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 server portability: The Proactor pattern improves server portability by allowing its interface to be reused independently of the underlying operating system 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. For instance, on real-time POSIX platforms, the asynchronous I/O functions are provided by the `aio` family of APIs [11]. Likewise, on Windows NT, I/O completion ports and overlapped I/O are used to implement asynchronous I/O [12].

The Completion Dispatcher encapsulates the concurrency mechanism: A benefit of decoupling the `Proactor` from the `Asynchronous Operation Processor` is that servers can configure `Completion Dispatchers` with various concurrency strategies without affecting other participants. As discussed in Section 6, the `Proactor` can be configured to use several concurrency strategies including single-threaded and thread pool solutions.

Threading policy is decoupled from the concurrency policy: The Asynchronous Operation Processor completes potentially long-running operations on behalf of Proactive Initiators. Therefore, servers are not forced to spawn threads to increase concurrency. This allows a server 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 server performance significantly if the operating system context switches to an idle thread.[3] For instance, threads may poll the operating system 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 need not activate an HTTP Handler if no GET request is pending.

Simplification of server synchronization: As long as Completion Handlers do not spawn additional threads of control, server 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], and hence no additional threads need be spawned.

The Proactor pattern has the following **liabilities**:

Hard to debug: Servers written with the Proactor pattern can be hard to debug because 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 because server 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 servers, debugging is straightforward when the thread of control is within the user-defined action routines. Once the thread of control returns to the generated Deterministic Finite Automata (DFA) skeleton, however, it is hard to follow the program logic.

Scheduling and controlling outstanding operations: The Proactive Initiators may not be able to control the order in which Asynchronous Operations are executed by an Asynchronous Operation Processor. Therefore, an Asynchronous Operation Processor should be designed to support prioritization and cancellation of Asynchronous Operations.

[3] Some older operating system exhibit this behavior, though most modern operating systems do not.

10 See Also

The Proactor pattern is related to the Observer pattern [9], 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 [7]. 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 server 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.

The Chain of Responsibility (COR) pattern [9] 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.

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* 2^{nd} *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. Hu, I. Pyarali, and D. C. Schmidt, "The Object-Oriented Design and Performance of JAWS: A High-performance Web Server Optimized for High-speed Networks," *Parallel and Distributed Computing Practices Journal, special issue on Distributed Object-Oriented Systems*, to appear.

[4] J. Hu, S. Mungee, and D. C. Schmidt, "Principles for Developing and Measuring High-performance Web Servers over ATM," in *Proceeedings of INFOCOM '98*, March/April 1998.

[5] 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.

[6] 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.

[7] 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.

[8] W. Pree, *Design Patterns for Object-Oriented Software Development*. Reading, MA: Addison-Wesley, 1994.

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

[10] 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.

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

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

[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.