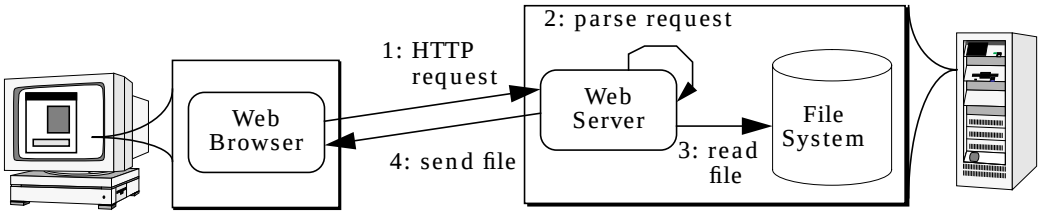
# Proactor

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

**Example** Consider a networking application that must perform multiple operations simultaneously. For example, a high-performance Web server must concurrently process HTTP requests sent from multiple remote client Web browsers [HPS99]. When a user wants to download content from a URL, the browser establishes a connection to the Web server and sends an HTTP GET request. The Web server subsequently receives the browser's `CONNECT` indication event, accepts the connection, and reads the request. It then parses and validates the request, sends the specified file(s) back to the Web, and closes the connection.




One way to implement a Web server is to use a reactive event demultiplexing model in accordance with the Reactor pattern (75). When a Web browser connects to a Web server, a new event handler is created and registered with a reactor, which coordinates the synchronous demultiplexing and dispatching of indication events. After the browser's HTTP `GET` request arrives, the reactor demultiplexes the associated indication event and calls back to the event handler. The handler then reads, parses, and processes the request and transfers the file back to the browser. Although this model is straightforward to program, however, it does not scale up to support many simultaneous users and/or long-duration user requests because it serializes all HTTP processing at the event demultiplexing layer.

A more scalable 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 browser's HTTP `GET` request [HS98], for example, one thread for each request. Each thread performs connection establishment, HTTP request reading, request parsing, and file transfer operations *synchronously*, that is, server processing operations block until they complete. Although this is a common concurrency model, however, the same efficiency, programming complexity, and portability drawbacks with synchronous multi-threading arise as discussed in the Reactor pattern (75).

As a result of the drawbacks outlined above, using reactive event demultiplexing or 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 users.

**Context** An event-driven application that processes multiple service requests simultaneously.

**Problem** Event-driven applications, particularly servers, in a distributed system must handle multiple service requests asynchronously, as well as invoke requests themselves that execute asynchronously.

➥ For example, our Web server can be programmed to receive completion events when TCP `CONNECT` and HTTP `GET` requests arrive asynchronously. Likewise, the Web server can invoke `read` and `write` operations asynchronously to transmit a requested file back to a Web browser. When these asynchronous operations complete, the operating system delivers `READ` and `WRITE` completion events, respectively, to the Web server. ❑

Before it invokes a specific service to process a completion event, an application must demultiplex and dispatch each event to its corresponding service handler. Addressing this problem effectively requires the resolution of the following four *forces*:

- To maximize performance, an application must be able to process multiple completion events simultaneously without having long-duration operations postpone the processing of other operations for an excessive amount of time.
- An application must minimize latency, maximize throughput, and avoid utilizing the CPU(s) unnecessarily.

- The design of an application should simplify its use of suitable concurrency strategies.
- Integrating new or improved services should incur minimal modifications to the generic event demultiplexing and dispatching mechanisms used by an application.

**Solution** Invoke operations asynchronously and integrate the demultiplexing of asynchronously delivered completion events resulting from these operations with their dispatching to service handlers that process these events. In addition, decouple these general-purpose event demultiplexing and dispatching mechanisms from application-specific processing of completion events within the service handlers.

For every service offered by an application, introduce an *asynchronous operation* to perform the associated service requests asynchronously, together with a *completion handler* to process completion events containing the results of each asynchronous operation. An asynchronous operation is invoked by a *client* and performed by an *asynchronous operation processor*. When the operation finishes executing a completion event is generated by the asynchronous operation processor. In addition, the asynchronous operation processor notifies a *proactor*, which dispatches to the application-specific *concrete completion handler* associated with the asynchronous operation. This completion handler then processes the results of the asynchronous operation.

**Structure** The key participants in the Proactor pattern include the following:

*Handles* identify operating system resources that can generate and queue completion events, such as network connections, open files, timers, synchronization objects, and I/O completion ports.

| ***Class*** Handle | ***Collaborator*** |
|---|---|
| ***Responsibility*** • Identifies an operating system resource • Generates and queues completion events | |

➥ The Web server creates a separate socket handle for each client connection. Completion events occur on these handles when asynchronous connect, read, and write operations finish executing.❑

*Asynchronous operations* are used to execute service requests, such as writing and writing data through a socket handle, asynchronously. After an asynchronous operation is invoked, the operation is executed *without* borrowing the caller's thread of control. Thus, from the caller's perspective, the operations execute asynchronously.

➥ The types of asynchronous operations that can be used in a Web server include connect, read, and write operations. ❑

| ***Class*** Asynchronous Operation | ***Collaborator*** |
|---|---|
| ***Responsibility*** • Defines a service that can be executed asynchronously | |

A *completion handler* specifies an interface consisting of one or more hook methods [Pree95] that abstractly represent the application- or service-specific processing for completion events generated when asynchronous operations finish execution.

*Concrete completion handlers* derive from the abstract completion handler. Each implements the methods for a specific service that the application offers. In addition, concrete event handlers implement the inherited hook method that is responsible for processing completion events from external sources, such as data sent to the service from remote clients, or completion events an application generates internally, such as timeouts. When these completion events arrive, the proactor dispatches the hook method of the appropriate concrete completion handler.

➥ There are two concrete completion handlers in the Web server, HTTP handler and HTTP acceptor, that perform completion processing on these asynchronous operations. The HTTP handler is responsible for receiving, processing, and replying to HTTP `GET` requests.

The HTTP acceptor creates and connects HTTP handlers that process subsequent requests from remote clients asynchronously. ❑

| ***Class*** Completion Handler | ***Collaborator*** |
|---|---|
| ***Responsibility*** • Defines an interface for processing results of asynchronous operations | • Handle |

| ***Class*** Concrete Completion Handler | ***Collaborator*** |
|---|---|
| ***Responsibility*** • Processes results of asynchronous operations in a specific manner | |

Asynchronous operations are run to completion by an *asynchronous operation processor*, which is typically implemented by the operating system kernel. When an asynchronous operation finishes executing, the asynchronous operation processor delegates the subsequent completion dispatching to the appropriate proactor.

The *proactor* is a completion dispatcher that calls back to the designated concrete completion handler in an application after the corresponding asynchronous operation has finished executing. To enhance reuse and to separate concerns, the proactor decouples its higher-level completion handler dispatching strategies from the lower-level operation execution mechanisms provided by the asynchronous operation processor.

| ***Class*** Asynchronous Operation Processor | ***Collaborator*** • Asynchronous Operation • Proactor |
|---|---|
| ***Responsibility*** • Executes asynchronous operations • Delegates to a proactor when asynchronous operations complete | |

| ***Class*** Proactor | ***Collaborator*** • Completion Handlers |
|---|---|
| ***Responsibility*** • Dispatches Completion Handlers | |

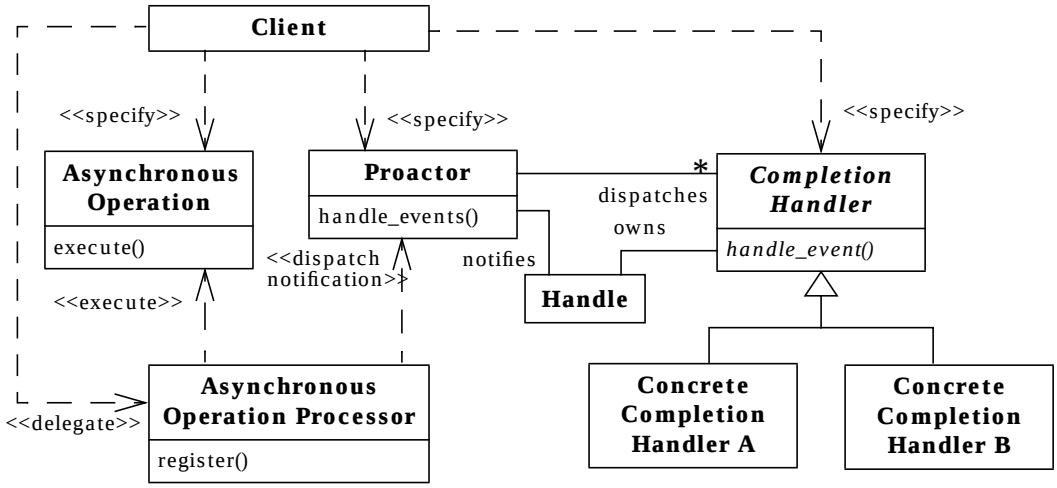
A *client* is any entity in an application that initiates an asynchronous operation. Note that the term 'client' refers to any programming entity that initiates an asynchronous operation, which may or may not correspond to the notion of a 'remote client' in a two-tier client/server

architecture. In particular, a server can play the role of a client when it initiates an asynchronous operation. When invoking an asynchronous operation, the client also registers a completion handler and a proactor with an asynchronous operation processor.

➥ In the Web server, the 'client' for asynchronous operations is its thread(s) of control. These thread(s) initiate asynchronous accept and read/write operations on HTTP acceptors and HTTP handlers, respectively, to process HTTP GET requests from Web browsers, which play the role of 'remote clients.' ❑

| ***Class*** Client | ***Collaborator*** |
|---|---|
| ***Responsibility*** | • Asynchronous Operation Processor |
| • Invokes asynchronous operations | • Asynchronous Operation |
| • Configures the asynchronous operation processor for executing specific asynchronous operations | • Proactor |
| | • Completion Handler |

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



**Dynamics** The following collaborations occur in the Proactor pattern:

- To execute a service asynchronously, an application playing the role of a client initiates an asynchronous operation by passing it to the asynchronous operation processor. In addition, the client must specify to the asynchronous operation processor which proactor should dispatch the callback on a particular completion handler when the asynchronous operation completes.

  ➥ For instance, the HTTP handler in a Web server may instruct the operating system to read an new HTTP `GET` request via an asynchronous operation on a particular socket handle. To request the operation, the Web server must specify which connected socket handle to use, as well as which completion handler the designated proactor should dispatch when the asynchronous read operation completes. ❑

- After a client invokes an operation on the asynchronous operation processor, the operation and client can run concurrently with respect to other asynchronous operations invoked by the application.

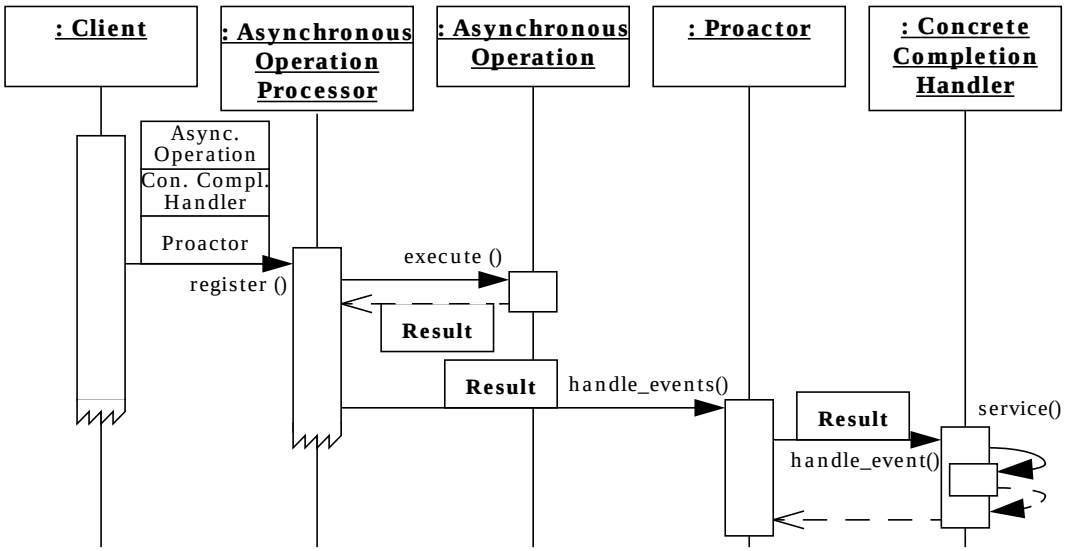
- When an operation finishes executing, the asynchronous operation processor retrieves the completion handler and proactor that were specified by the client when the operation was initiated. The asynchronous operation processor then passes the proactor both the result of the asynchronous operation and the completion handler it should dispatch.

  ➥ For example, if an HTTP handler reads an HTTP `GET` request asynchronously, the asynchronous operation processor may report to the appropriate proactor the completion status, such as success or failure, along with the number of bytes read. ❑

- The proactor dispatches the hook method on the completion handler, passing it any completion data specified by the server.

  ➥ For instance, a Web server might instruct the operating system to transmit a large file asynchronously across a network one chunk at a time. After the operating system completes each asynchronous write operation successfully, the proactor will pass the hook method on the associated completion handler the number of bytes

transferred so that it can invoke the next asynchronous write operation at the appropriate offset in the file. . ❑




**Implementation** The participants in the Proactor pattern can be decomposed into two layers: *demultiplexing/dispatching infrastructure* and *application components*. The demultiplexing/dispatching infrastructure layer performs generic, application-independent strategies for executing asynchronous operations, as well as demultiplexing and dispatching completion events from these asynchronous operations to their associated completion handlers. Components in the application layer define asynchronous operations and concrete completion handlers that perform application-specific service processing. The implementation steps in this section start at the bottom with the demultiplexing/dispatching infrastructure layer and work upwards to the application layer.

1 *Implement the asynchronous operation processor*. An asynchronous operation processor is responsible for executing operations asynchronously on behalf of clients. Thus, its two primary responsibilities are defining the asynchronous operation APIs and implementing a mechanism to execute operations asynchronously.

1.1 *Define the asynchronous operation APIs*. The asynchronous operation processor provides a facade [GHJV95] that allows clients to invoke

asynchronous operations. Several forces must be considered when designing an API for the asynchronous operation processor facade:

- *Portability*: The APIs should not tightly couple applications to a particular platform.
- *Flexibility*: Asynchronous APIs can be shared by many types of operations. For instance, asynchronous I/O operations often can be used to read and write data on multiple I/O devices, such as network and files. It may be beneficial to design APIs that support such reuse.
- *Callbacks*: A client registers a callback when it invokes an asynchronous operation so its proactor can dispatch the appropriate completion handler when the operation finishes executing. A common callback implementation is to have the client export an interface known by the callee. In this case, clients must inform the asynchronous operation processor which completion handler to call back when an operation finishes executing and a completion event is generated.
- *Proactor*: Applications can use multiple proactors simultaneously. Therefore, a client must indicate which proactor should dispatch completion handler callbacks for each asynchronous operation.

➥ Consider the following class used in our Web server example.

```
class Asynch_Stream {
     // A Factory for initiating reads and writes
     // asynchronously.
public:
     // Initializes the factory with information that will
     // be used with each asynchronous call. <handler> is
     // notified when the operation completes. The async.
     // operations are performed on the <handle>. 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 Asynch_Result {
                // Bytes transferred by asynchronous operation.
                u_long bytes_transferred ();

                // Error value if operation failed.
                u_long error ();

                // Asynchronous Completion Token (ACT)
                // associated with the operation.
                const void *act ();
            };

            class Read_Result : public Asynch_Result {
                // Read-specific information.
            };
            class Write_Result : public Asynch_Result {
                // Write-specific information
            };
        };
```

The `Asynch_Stream` class is a general-purpose factory that can be used to initiate multiple `read()` and `write()` operations asynchronously. When an asynchronous `read()` operation completes, the associated proactor will create an `Asynch_Stream::Read_Result` and pass it to the completion handler. Likewise, when the asynchronous `write()` operation completes, the associated proactor will create and pass an `Asynch_Stream::Write_Result` to the completion handler. These result classes contain information that allows the client application to unambiguously identify which asynchronous operation completed. In addition, the result classes indicate the success or failure of each operation, as well as conveying operation-specific information, such as how many bytes were transferred successfully.

1.2 *Implement the Asynchronous Operation Processing Mechanism*. The asynchronous operation processor contains a mechanism that executes operations asynchronously. Thus, when a client invokes an asynchronous operation, the operation must be performed *without* borrowing the client's thread of control. Many operating systems provide asynchronous operation execution mechanisms, such as 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 support asynchronous operations, however, several implementation techniques can be used to build an asynchronous operation processor. Perhaps the most intuitive solution is to use dedicated threads to perform asynchronous operations. There are three steps involved with implementing a threaded asynchronous operation processor:

- *Operation invocation*: The operation will be performed in a different thread of control than the invoking client 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 maintain a pool of dedicated threads. This approach requires the client thread to queue the operation request before continuing with other client computations.
- *Operation execution*: Each operation will be performed in a dedicated thread. Thus, operations can 'block' without impeding progress of the client directly. For instance, when providing a mechanism for asynchronous I/O reads, the dedicated thread can block while reading from socket or file handles.
- *Operation completion*: When the operation completes, the client must be notified. For example, a dedicated asynchronous operation processor thread can be used to delegate application-specific completion events to the appropriate proactor. This requires additional synchronization between threads.

2 *Implement the Proactor.* When the proactor receives operation completions from the asynchronous operation processor, it calls back to the completion handler that is associated with original client invocation. A Proactor is often implemented as a Singleton [GHJV95], which is useful for centralizing the demultiplexing and dispatching of asynchronous operation completion events into a single location within an application.

➥ For example, the following is an interface for the Proactor used in our Web server example:

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

    // Singleton access point.
    static Proactor *instance (void);
private:
};
```
❑

The `handle_events()` method is the entry point into a Proactor's event loop and is typically called by one or more server threads.[1] This method blocks the caller until it dispatches a single callback to a completion handler. The `instance()` method is the singleton access point.

In the following discussion, we describe four issues involved with implementing the proactor: demultiplexing completion notifications to the appropriate concrete completion handler, implementing a completion handler dispatching mechanism, defining the concurrency strategy used to perform the callbacks, and finally defining a preemptive policy.

2.1 *Determine how to demultiplex completion notifications to completion handlers*. When an asynchronous operation completes, the associated proactor may need more information than simply the completion event itself to dispatch the appropriate completion handler efficiently. A common solution is to use the Asynchronous Completion Token pattern (127).

In the Proactor pattern, when an asynchronous operation is invoked, the associated proactor uses the Asynchronous Completion Token pattern to associate state with the operation. This state typically involves a reference to the completion handler and an asynchronous completion token (ACT) provided by the client. An object containing this state is then passed as an ACT to the asynchronous operation processor, which plays the role of the *service* in the Asynchronous Completion Token pattern.

When the asynchronous operation completes, the asynchronous operation processor returns the ACT unchanged to the designated

1. Multiple threads can be organized into a thread pool and call `handle_events()` on the same Proactor simultaneously. This design is well-suited for I/O bound applications [HPS99]

proactor, along with information about the completed operation. The proactor uses this ACT to find the completion handler and to recover the client's ACT. It then calls back the completion handler, passing it information about the completed operation and the ACT that was provided by the client during its original asynchronous invocation.

The proactor's ACT contains a reference to the completion handler. Therefore, no additional lookups are necessary for the proactor to dispatch the completion to the completion handler. Thus, the ACT pattern enables the proactor to dispatch to the appropriate completion handler in constant time, regardless of the number of operations that have been invoked asynchronously.

2.2 *Implement the completion handler dispatching mechanism*. The proactor must implement a mechanism through which concrete completion handlers are invoked. This requires clients 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.
- *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 client can establish a synchronization object, such as 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 client. While the asynchronous operation runs to completion, the completion handler processes other activity. Periodically, the completion handler will check the rendezvous point for completion status.

2.3 *Define the proactor concurrency strategy*. When operations complete, the asynchronous operation processor will notify the appropriate proactor. At this point, the proactor can utilize one of the following concurrency strategies to perform its dispatching:

- *Dynamic-thread dispatching*. A proactor can dynamically allocate a thread for each completion handler. Dynamic-thread dispatching can be implemented with most multi-threaded operating systems. On some platforms, however, this may be inefficient due to the overhead of creating and destroying threads.
- *Event demultiplexer dispatching*. A synchronization object, such as a condition variable, established by the client can be signaled by a proactor when a completion event is pending. Although polling and spawning a child thread that blocks on a condition variable are potential approaches, it may be more efficient to use *event demultiplexer dispatching*. This technique requires the client to register completion events with some type of event demultiplexer. For instance, event demultiplexer dispatching can be implemented with `aio_suspend()` in POSIX real-time environments, with `WaitForMultipleObjects()` in Win32 environments, or even with a Reactor (75).
- *Call-through dispatchin*g. 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 application 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.

The applicability of the four proactor concurrency techniques described above depends largely on the possible combinations of operating system environments and physical hardware.

- If your operating system only supports synchronous I/O and does not support threads, then refer to the Reactor (75) pattern. However, most modern operating systems support alternative forms of asynchronous I/O, such as POSIX asynchronous I/O and Windows NT overlapped I/O, as well as threads.
- For single-threaded applications on either single-processor or multi-processor hardware, the correct concurrency strategy depends on the type of operations that will be performed. If an application will only perform proactive asynchronous operations then a single thread can be used. If an application will be performing both reactive synchronous operations and proactive asynchronous operations, then event demultiplexer dispatching should be used. Lastly, if the single-threaded application needs to perform long-blocking synchronous operations occasionally, call-through dispatching should be used.
- For multi-threaded applications on either single-processor or multi-processor hardware, any of the dispatching techniques can be appropriate. Often, systematic empirical measurements are the best way to make a selection. For instance, multi-threaded solutions running on single-processor systems can decrease performance for compute-bound applications by increasing context switching overhead. In contrast, a single-processor, multi-threaded solution can sometimes increase performance for I/O-bound applications by allowing the operating system and threading package to overlap computation and communication.

2.4 *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. When tied to a proactor

implemented using event demultiplexer dispatching, however, 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 completion handler should not perform long-duration synchronous operations unless multiple completion threads are used because this will significantly decrease the overall responsiveness of the application. This risk can be alleviated by following particular programming conventions. For instance, all completion handlers might be required to act as clients instead of executing synchronous operations.

3 *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. Application programmers define concrete completion handlers that process service-specific completion events generated when asynchronous operations finish executing. The implementation of both abstract and concrete completion handlers contains the following two substeps.

3.1 *Define the completion handler interface*. The concrete implementation of the completion handler interface depends on the type of completion handler dispatching mechanism selected for the proactor: callback class, function pointer, or rendevouz. In case of implementing a callback class there are two approaches for designing the completion handler interface.

- *A single-method interface*. The UML class diagram in the *Structure* section illustrates an implementation of the `Completion_Handler` base class interface that contains a single method with a signature `handle_event()`. This method is used by the proactor to dispatch events. In this case, the type of the event that has occurred is passed as a parameter to the method. The second parameter is the base class for all asynchronous results, which can be further downcast to the correct type, depending on the completed event.

  ➥ The following C++ abstract base class illustrates the single-method interface:

```
class Completion_Handler {
public:
     // Hook method that is called back by the
     // Proactor to handle events of a particular type. The
     // <Async_Result> parameter contains both the
     // information about the completed asynchronous
     // operation as well as the corresponding ACT.
     virtual int handle_event
          (Event_Type et,
           const Asynch_Stream::Asynch_Result &result) = 0;

     // Hook method that returns the underlying I/O Handle.
     virtual Handle get_handle (void) const = 0;
};                                                          ❑
```

The advantage of the single-method interface is that it is possible to add new types of events without changing the method interface. However, this approach encourages the use of switch statements in the subclass's `handle_event()` method implementation, which limits its internal extensibility.

- *A multi-method interface*. Another way to implement the `Event_Handler` interface is to define separate virtual hook methods for each general type of event, such as `handle_read_stream()`, `handle_write_stream()`, or `handle_accept()` in our Web server example.

  ➥ 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 accept completes.
        virtual void handle_accept
             (const Asynch_Stream::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;
  };                                                          ❑
```

The benefit of a multi-method interface is that it is easy to selectively override methods in the base class and avoid further demultiplexing via `switch` or `if` statements in the hook method implementation. However, this design requires the framework developer to anticipate the set of completion handler hook methods in advance. For instance, the various `handle_*` hook methods in the `Completion_Handler` interface above are tailored for general networking events. However, this interface is not broad enough to encompass all the types of events, such as synchronization events, handled via the Win32 `WaitForMultipleObjects()` mechanism [Sch95b].

Both approaches described above are examples of patterns such as Hook Method [Pree95] and Template Method [GHJV95]. The intent of these patterns is to provide well-defined hooks that can be specialized by applications and called back by lower-level dispatching code.

3.2 *Determine policies for handling state in concrete completion handlers.* A completion handler may need to maintain state information associated with a specific request. For instance, an operating system may notify the Web server that only part of a file was written to a socket due to the occurrence of transport-level flow control. As a result, a completion handler may need to issue the remainder of the request until the file is fully transferred or the connection becomes invalid. Therefore, it must know the file that was originally specified, how many bytes are left to write, and what the file pointer position was at the start of the previous request.

There is no implicit limitation that prevents clients from assigning multiple asynchronous operation requests to a single completion handler. As a result, the completion handler must shepherd request-

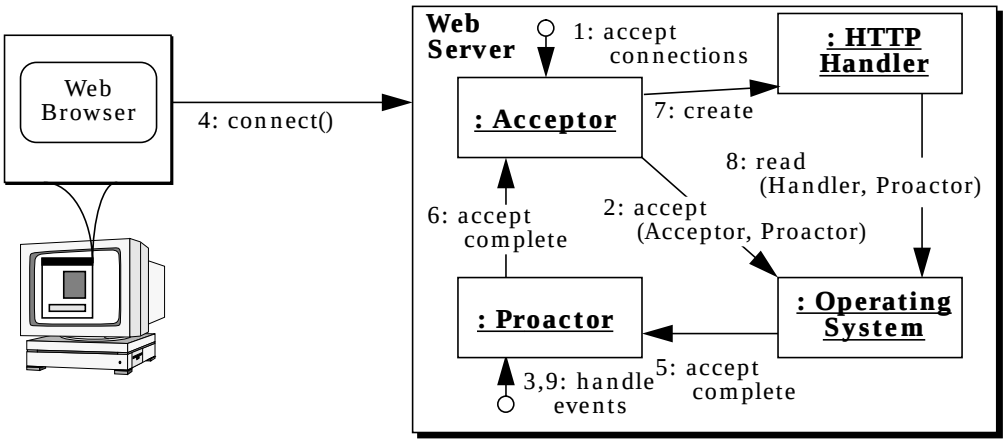
specific state information throughout the chain of completion notifications. To do this, completion handlers can utilize the Asynchronous Completion Token pattern (127).

**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 service requests concurrently within one or more threads. As described in the *Structure* section, the HTTP handler is responsible for receiving, processing, and replying to requests from Web browsers. HTTP acceptors create and connect HTTP handlers that process subsequent requests, which are delivered via completion events.

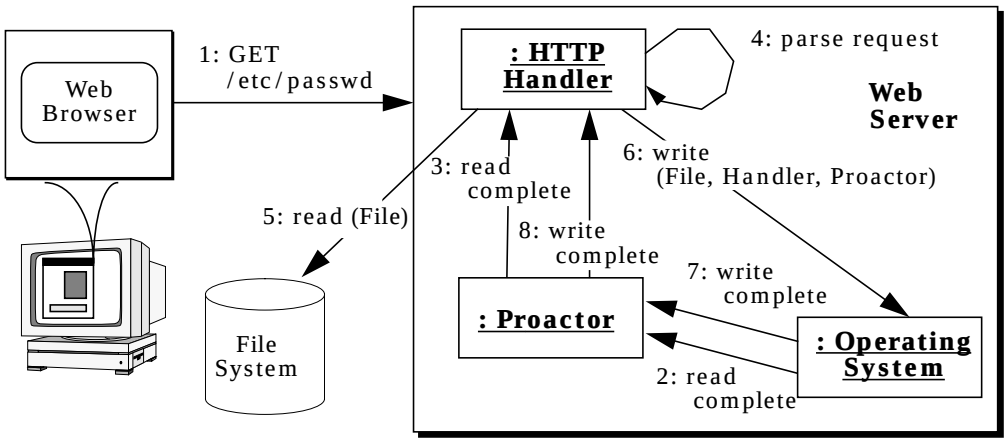
The following sequence of steps occur during Web server connection processing:

- The Web server instructs (1) the acceptor, acting in the role of a client, to initiate an asynchronous accept operation.
- The acceptor invokes an asynchronous accept operation (2) on the operating system and passes itself as a completion handler and a reference to the proactor that will be used to dispatch completion events back to the acceptor after the asynchronous accept operation is finished.
- The Web server invokes the event loop of the proactor (3).
- The Web browser connects to the Web server (4).
- After the asynchronous accept operation completes, the operating system generates a completion event and notifies the proactor (5), which dispatches the completion event to the acceptor (6).
- The acceptor creates an HTTP handler (7) to process the completion event.
- The HTTP handler parses the completion event and then initiates an asynchronous read operation (8) to obtain the `GET` request data from the Web browser. The HTTP handler passes itself as the completion handler and also passes a reference to the proactor that will be used to notify the HTTP handler when the asynchronous read operation finishes executing.
- Control of the Web server then returns to the event loop of the Proactor (9).




Once the connection is established and the HTTP handler created, the following scenario is used by a proactive Web server to service an HTTP GET request:

- The Web browser sends (1) an HTTP GET request.
- The asynchronous read operation initiated in the previous scenario completes and the operating system generates a completion event and notifies the proactor (2). The proactor then dispatches to the HTTP handler (3).
- The HTTP handler parses the request (4). Steps (2) through (4) will repeat until the entire request has been received asynchronously.
- The HTTP handler memory-maps the requested file (5).
- HTTP Handle initiates an asynchronous write operation (6) to transfer the file data via the 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.
- When the asynchronous write operation completes, the operating system generates another completion event and notifies the proactor (7), which dispatches to the completion handler (8). Steps (6) through (8) continue asynchronously until the entire file has been delivered.




The following C++ code illustrates how the HTTP handler for our Web server example can be written using the `Connection_Handler` class defined in the *Implementation* section.

```
class HTTP_Handler : public Completion_Handler  {
    // Implements the HTTP protocol asynchronously.
public:
    // Constructor.
    HTTP_Handler (Proactor *);

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

    // This is called by the Proactor
    // when the asynch read completes.
    void handle_read_stream
        (const Asynch_Stream::Read_Result &result);

    // This is called by the Proactor
    // when the asynch write completes.
    void handle_write_stream
        (const Asynch_Stream::Write_Result &result);

private:
    // Parse the incoming request
    void parse_request (void);

    // Set by the constructor.
    Proactor *proactor_;
```

```
        // Memory-mapped file_;
        Mem_Map file_;

        // Socket endpoint.
        SOCK_Stream *sock_;

        // HTTP Request holder
        HTTP_Request request_;

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

When a Web browser connects to the Web server, the HTTP_Handler's following open() method is called by an acceptor:

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

        // Store reference to the socket.
        sock_ = sock;

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

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

In this method, the HTTP_Handler initializes the asynchronous stream object with the completion handler and proactor to use when the asynchronous operation completes. The read operation is then invoked asynchronously and the Web server returns to its event loop, which calls the Proactor::handle_events() method, as follows:

```
    while (web_server_not_shutdown)
        proactor->handle_events ();
```

When the asynchronous read operation completes, the following handle_read_stream() method is called back by the proactor on the HTTP_Handler completion handler:

```
    void HTTP_Handler::handle_read_stream
        (const Asynch_Stream::Read_Result &result) {
        if (request_.enough_data
                      (result.bytes_transferred ()))
            parse_request ();
```

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

If there is enough data, the Web browser's service request is then parsed. If the entire request has not arrived yet, another read operation is initiated asynchronously and the Web server returns to its event loop. Once a GET request has been received from a Web browser, the `parse_request()` method maps the requested file into memory and writes the file data asynchronously to the Web browser, as follows:.

```
    void HTTP_Handler::parse_request (void) {
        // Switch on the HTTP command type.
        switch (request_.command ()) {
        // Web browser 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;
        // Web browser is storing a file at the Web server.
        case HTTP_Request::PUT:
            // ...
        }
    }
```

When the write operation completes, the proactor calls back on the HTTP handler's `handle_write_stream()` method:

```
    void HTTP_Handler::handle_write_stream
        (const Asynch_Stream::Write_Result &result) {
        if (file_.enough_data (result.bytes_transferred ()))
            // Success, so cleanup resources.
        else
        // Start another asynchronous write
            stream_.write (file_.buffer (),
                           file_.buffer_size ());
    }
```

When all the data has been received, the HTTP handler frees up all resources that were allocated dynamically.

**Known uses**

**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 [POSIX95]. 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) [Sch97] 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.

**Consequences**

The Proactor pattern offers the following **benefits**:

*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 in completion handlers. Likewise, the application-specific functionality knows how to perform a particular type of service, such as HTTP processing.

*Portability*. The Proactor pattern improves application 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 [POSIX95]. Likewise, on Windows NT, I/O completion ports and overlapped I/O are used to implement asynchronous I/O [MDS96].

*Encapsulation of concurrency mechanisms*. A benefit of decoupling the proactor from the asynchronous operation processor is that applications can configure proactors with various concurrency strategies without affecting other application components and services. As discussed in the *Implementation* section, the proactor can be configured to use various concurrency strategies, including single-threaded and thread pools.

*Decoupling of threading from concurrency*. The asynchronous operation processor executes potentially long-running operations on behalf of clients. Therefore, applications need not spawn many 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.

*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 operating system context switches to an idle thread.[2] 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 application synchronization*. As long as completion handlers do not spawn additional threads of control, application logic

2. Some older operating system exhibit this behavior, though most modern operating systems do not.

can be written with little or no concern for 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 asynchronous read operation, such as the Windows NT `TransmitFile()` function [HPS97], and hence no additional threads need be spawned.

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

*Hard to debug*. Applications 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 run-time behavior of a framework within a debugger because 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 user-defined semantic 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*. Clients 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.

**See Also** The Proactor pattern is related to the Observer [GHJV95] and Publisher-Subscriber [POSA1] patterns, where all dependents are informed when a single subject changes. In the Proactor pattern, however, handlers are informed automatically when events from multiple sources occur. In general, the Proactor pattern is used to demultiplex multiple sources of asynchronously delivered completion events to their associated completion handlers, whereas an observer or subscriber is usually associated with only a single source of events.

The Proactor pattern can be considered an *asynchronous* variant of the synchronous Reactor pattern (75). 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 operations*.

The Active Object pattern (239) decouples method execution from method invocation. The Proactor pattern is similar because asynchronous operation processors perform operations on behalf of clients. 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 [GHJV95] pattern decouples event handlers from event sources. The Proactor pattern is similar in its segregation of clients and completion handlers. However, in COR, the event source has no prior knowledge of which handler will be executed, if any. In Proactor, clients have full disclosure of the target handler. However, the two patterns can be combined by establishing a completion handler that is the entry point into a responsibility chain dynamically configured by an external factory.

**Credits** Tim Harrison, Thomas D. Jordan, and Irfan Pyarali are co-authors of the original version of the Proactor pattern.