APPLYING THE PROACTOR PATTERN TO HIGH-PERFORMANCE WEB SERVERS

James Hu
jxh@cs.wustl,edu

Irfan Pyarali
irfan@cs.wustl.edu

Douglas C. Schmidt
schmidt@cs.wustl.edu

Department of Computer Science, Washington University
St. Louis, MO 63130, USA*

This paper is a revised and expanded version of the paper that will appear in the 10th International Conference on Parallel and Distributed Computing and Systems, IASTED, Las Vegas, Nevada, October 28-31, 1998.

ABSTRACT

Modern operating systems provide multiple concurrency mechanisms to develop high-performance Web servers. Synchronous multi-threading is a popular mechanism for developing Web servers that must perform multiple operations simultaneously to meet their performance requirements. In addition, an increasing number of operating systems support asynchronous mechanisms that provide the benefits of concurrency, while alleviating much of the performance overhead of synchronous multi-threading.

This paper provides two contributions to the study of high-performance Web servers. First, it examines how synchronous and asynchronous event dispatching mechanisms impact the design and performance of JAWS, which is our high-performance Web server framework. The results reveal significant performance improvements when a ***proactive*** *concurrency model is used to combine lightweight concurrency with asynchronous event dispatching.*

In general, however, the complexity of the proactive concurrency model makes it harder to program applications that can utilize asynchronous concurrency mechanisms effectively. Therefore, the second contribution of this paper describes how to reduce the software complexity of asynchronous concurrent applications by applying the ***Proactor pattern****. This pattern describes the steps required to structure object-oriented applications that seamlessly combine concurrency with asynchronous event dispatching. The Proactor pattern simplifies concurrent programming and improves performance by allowing concurrent application to have multiple operations running simultaneously without requiring a large number of threads.*

*This work was supported in part by Microsoft, Siemens Med, and Siemens Corporate Research.

1 INTRODUCTION

Computing power and network bandwidth on the Internet has increased dramatically over the past decade. High-speed networks (such as ATM and Gigabit Ethernet) and high-performance I/O subsystems (such as RAID) are becoming ubiquitous. In this context, developing scalable Web servers that can exploit these innovations remains a key challenge for developers. Thus, it is increasingly important to alleviate common Web server bottlenecks, such as inappropriate choice of concurrency and dispatching strategies, excessive filesystem access, and unnecessary data copying.

Our research vehicle for exploring the performance impact of applying various Web server optimization techniques is the *JAWS Adaptive Web Server* (JAWS) [1]. JAWS is both an adaptive Web server and a development framework for Web servers that runs on multiple OS platforms including Win32, most versions of UNIX, and MVS Open Edition.

Our experience [2] building Web servers on multiple OS platforms demonstrates that the effort required to optimize performance can be simplified significantly by leveraging OS-specific features. For example, an optimized file I/O system that automatically caches open files in main memory via `mmap` greatly reduces latency on Solaris. Likewise, support for asynchronous event dispatching on Windows NT can substantially increase server throughput by reducing context switching and synchronization overhead incurred by multi-threading.

Unfortunately, the increase in performance obtained through the use of asynchronous event dispatching on existing operating systems comes at the cost of increased software complexity. Moreover, this complexity is further compounded when asynchrony is coupled with multi-threading. This style of programming, *i.e.*, proactive programming, is relatively unfamiliar to many developers accustomed to the synchronous event dispatching paradigm. This paper describes how the *Proactor pattern* can be applied to improve both the performance and the design of high-performance communication applications, such as Web servers.

A pattern represents a recurring solution to a software development problem within a particular context [3]. Patterns

identify the static and dynamic collaborations and interactions between software components. In general, applying patterns to complex object-oriented concurrent applications can significantly improve software quality, increase software maintainability, and support broad reuse of components and architectural designs [4]. In particular, applying the *Proactor* pattern to JAWS simplifies asynchronous application development by structuring the demultiplexing of completion events and the dispatching of their corresponding completion routines.

The remainder of this paper is organized as follows: Section 2 provides an overview of the JAWS server framework design; Section 3 discusses alternative event dispatching strategies and their performance impacts; Section 4 explores how to leverage the gains of asynchronous event dispatching through application of the Proactor pattern; and Section 5 presents concluding remarks.

2 JAWS FRAMEWORK OVERVIEW

Figure 1 illustrates the major structural components and design patterns that comprise the JAWS framework [1]. JAWS

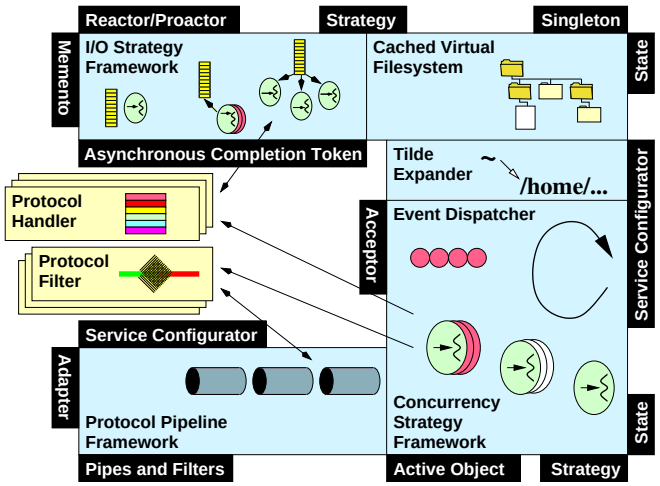

Figure 1: Architectural Overview of the JAWS Framework

is designed to allow the customization of various Web server strategies in response to environmental factors. These factors include *static* factors (*e.g.*, number of available CPUs, support for kernel-level threads, and availability of asynchronous I/O in the OS), as well as *dynamic* factors (*e.g.*, Web traffic patterns and workload characteristics).

JAWS is structured as a *framework of frameworks*. The overall JAWS framework contains the following components and frameworks: an *Event Dispatcher*, *Concurrency Strategy*, *I/O Strategy*, *Protocol Pipeline*, *Protocol Handlers*, and *Cached Virtual Filesystem*. Each framework is structured as a set of collaborating objects implemented using components in ACE [5]. The collaborations among JAWS components and frameworks are guided by a family of patterns, which are listed along the borders in Figure 1. An outline of the key frameworks, components, and patterns in JAWS is presented below; Section 4 then focuses on the Proactor pattern in detail.[1]

Event Dispatcher: This component is responsible for coordinating JAWS' *Concurrency Strategy* with its *I/O Strategy*. The passive establishment of connection events with Web clients follows the *Acceptor* pattern [6]. New incoming HTTP request events are serviced by a concurrency strategy. As events are processed, they are dispatched to the *Protocol Handler*, which is parameterized by an I/O strategy. JAWS ability to dynamically bind to a particular concurrency strategy and I/O strategy from a range of alternatives follows the *Strategy* pattern [3].

Concurrency Strategy: This framework implements concurrency mechanisms (such as single-threaded, thread-per-request, or thread pool) that can be selected adaptively at run-time using the *State* pattern [3] or pre-determined at initialization-time. The *Service Configurator* pattern [7] is used to configure a particular concurrency strategy into a Web server at run-time. When concurrency involves multiple threads, the strategy creates protocol handlers that follow the *Active Object* pattern [8].

I/O Strategy: This framework implements various I/O mechanisms, such as asynchronous, synchronous and reactive I/O. Multiple I/O mechanisms can be used simultaneously. In JAWS, asynchronous I/O is implemented using the *Proactor* pattern [9], while reactive I/O is accomplished through the *Reactor* pattern [10]. These I/O strategies may utilize the *Memento* [3] and *Asynchronous Completion Token* [11] patterns to capture and externalize the state of a request so that it can be restored at a later time.

Protocol Handler: This framework allows system developers to apply the JAWS framework to a variety of Web system applications. A *Protocol Handler* is parameterized by a concurrency strategy and an I/O strategy. These strategies are decoupled from the protocol handler using the *Adapter* pattern [3]. In JAWS, this component implements the parsing and handling of HTTP/1.0 request methods. The abstraction allows for other protocols (such as HTTP/1.1, DICOM, and SFP [12]) to be incorporated easily into JAWS. To add a new protocol, developers simply write a new *Protocol Handler* implementation, which is then configured into the JAWS framework.

Protocol Pipeline: This framework allows filter operations to be incorporated easily with the data being processed by the *Protocol Handler*. This integration is achieved by employing the *Adapter* pattern. Pipelines follow the *Pipes and Filters* pattern [13] for input processing. Pipeline components can be

[1] Due to space limitations it is not possible to describe all the patterns mentioned below in detail. The references provide complete coverage of each pattern, however.

linked dynamically at run-time using the *Service Configurator* pattern.

Cached Virtual Filesystem: This component improves Web server performance by reducing the overhead of filesystem access. Various caching strategies, such as LRU, LFU, Hinted, and Structured, can be selected following the *Strategy* pattern [3]. This allows different caching strategies to be profiled and selected based on their performance. Moreover, optimal strategies to be configured statically or dynamically using the *Service Configurator* pattern. The cache for each Web server is instantiated using the *Singleton* pattern [3].

Tilde Expander: This component is another cache component that uses a perfect hash table [14] that maps abbreviated user login names (*e.g.*, `~schmidt`) to user home directories (*e.g.*, `/home/cs/faculty/schmidt`). When personal Web pages are stored in user home directories, and user directories do not reside in one common root, this component substantially reduces the disk I/O overhead required to access a system user information file, such as `/etc/passwd`. By virtue of the *Service Configurator* pattern, the Tilde Expander can be unlinked and relinked dynamically into the server when a new user is added to the system.

Our previous work on high-performance Web servers has focused on (1) the design of the JAWS framework [1] and (2) detailed measurements on the performance implications of alternative Web server optimization techniques [2]. In our earlier work, we discovered that a concurrent proactive Web server can achieve substantial performance gains [15].

This paper focuses on a previously unexamined point in the high-performance Web server design space: *the application of the Proactor pattern to simplify Web server software development, while maintaining high-performance.* Section 3 motivates the need for concurrent proactive architectures by analyzing empirical benchmarking results of JAWS and outlining the software design challenges involved in developing proactive Web servers. Section 4 then demonstrates how these challenges can be overcome by designing the JAWS Web server using the Proactor pattern.

3 CONCURRENCY ARCHITECTURES

Developing a high-performance Web server like JAWS requires the resolution of the following forces:

• **Concurrency:** The server must perform multiple client requests simultaneously;

• **Efficiency:** The server must minimize latency, maximize throughput, and avoid utilizing the CPU(s) unnecessarily.

• **Adaptability:** Integrating new or improved transport protocols (such as HTTP 1.1 [16]) should incur minimal enhancement and maintenance costs.

• **Programming simplicity:** The design of the server should simplify the use of various concurrency strategies, which may differ in performance on different OS platforms;

The JAWS Web server can be implemented using several concurrency strategies, such as multiple synchronous threads, reactive synchronous event dispatching, and proactive asynchronous event dispatching. Below, we compare and contrast the performance and design impacts of using conventional multi-threaded synchronous event dispatching versus proactive asynchronous event dispatching, using our experience developing and optimizing JAWS as a case-study.

3.1 CONCURRENT SYNCHRONOUS EVENTS

Overview: An intuitive and widely used concurrency architecture for implementing concurrent Web servers is to use *synchronous multi-threading*. In this model, multiple server threads can process HTTP `GET` requests from multiple clients simultaneously. Each thread performs connection establishment, HTTP request reading, request parsing, and file transfer operations synchronously. As a result, each operation blocks until it completes.

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

Figure 2 shows how JAWS can be configured to use synchronous threads to process multiple clients concurrently. This figure shows a `Sync Acceptor` object that encapsulates the server-side mechanism for synchronously accepting network connections.

The sequence of steps that each thread uses to service an HTTP `GET` request using a Thread Pool concurrency model can be summarized as follows:

1. Each thread synchronously blocks in the `Sync Acceptor` waiting for a client connection request;
2. A client connects to the server, and the `Sync Acceptor` selects one of the waiting threads to accept the connection;
3. The new client's HTTP request is synchronously read from the network connection by the selected thread;
4. The request is parsed;
5. The requested file is synchronously read;
6. The file is synchronously sent to the client.

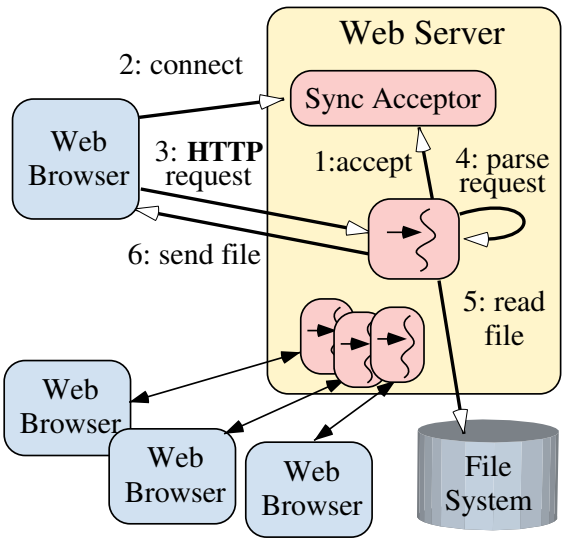

Figure 2: Multi-threaded Web Server Architecture

As described above, each concurrently connected client is serviced by a dedicated server thread. The thread completes a requested operation synchronously before servicing other HTTP requests. Therefore, to perform synchronous I/O while servicing multiple clients, JAWS must spawn multiple threads.

Evaluation: Although the synchronous multi-threaded model is intuitive and maps relatively efficiently onto multi-CPU platforms, it has the following drawbacks:

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

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

• **Increased performance overhead:** Threading can perform poorly due to context switching, synchronization, and data movement among CPUs [5];

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

As a result of these drawbacks, multi-threading may not always be the most efficient nor the least complex solution to develop concurrent Web servers. The solution may not be obvious, since the disadvantages may not result in any actual performance penalty except under certain conditions, such as a particularly high number of long running requests intermixed with rapid requests for smaller files. Therefore, it is important to explore alternative Web server architecture designs, such as the concurrent asynchronous architecture described next.

3.2 CONCURRENT ASYNCHRONOUS EVENTS

Overview: When the OS platform supports asynchronous operations, an efficient and convenient way to implement a high-performance Web server is to use *proactive event dispatching*. Web servers designed using this dispatching model handle the *completion* of asynchronous operations with one or more threads of control.

JAWS implements proactive event dispatching by first issuing an asynchronous operation to the OS and registering a callback (which is the `Completion Handler`) with the *Event Dispatcher*.[2] This *Event Dispatcher* notifies JAWS when the operation completes. The OS then performs the operation and subsequently queues the result in a well-known location. The *Event Dispatcher* is responsible for dequeuing completion notifications and executing the appropriate `Completion Handler`.

Figures 3 and 4 show how the JAWS Web server configured using proactive event dispatching handles multiple clients concurrently within one or more threads. Figure 3 shows the sequence of steps taken when a client connects to JAWS.

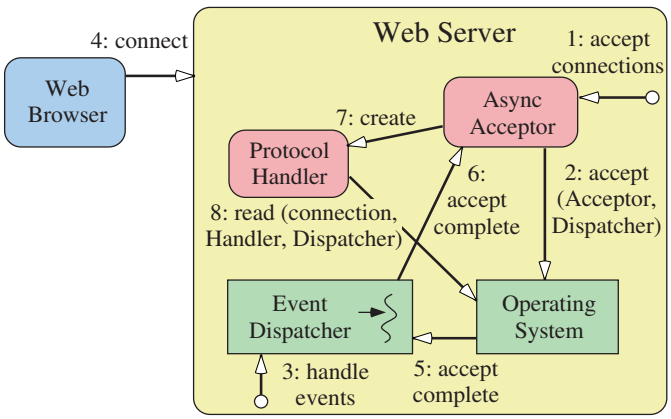

Figure 3: Client connects to the Proactor-based JAWS Web Server

1. JAWS instructs the `Async Acceptor` to initiate an asynchronous accept;
2. The `Async Acceptor` initiates an asynchronous accept with the OS and passes itself as a `Completion`

[2]In our discussion, JAWS framework components presented in Section 2 appear in *italics* and pattern participants presented in Section 4 appear in `typewriter` font.

`Handler` and a reference to the *Event Dispatcher* that will be used to notify the `Async Acceptor` upon completion of the asynchronous accept;

3. JAWS invokes the event loop of the *Event Dispatcher*;
4. The client connects to JAWS;
5. When the asynchronous accept operation completes, the OS notifies the *Event Dispatcher*;
6. The *Event Dispatcher* notifies the `Async Acceptor`;
7. The `Async Acceptor` creates an appropriate *Protocol Handler*;
8. The *Protocol Handler* initiates an asynchronous operation to read the request data from the client and passes itself as a `Completion Handler` and a reference to the *Event Dispatcher* that will be used to notify the *Protocol Handler* upon completion of the asynchronous read.

Figure 4 shows the sequence of steps that the JAWS uses to service an HTTP `GET` request when it is configured using proactive event dispatching. These steps are outlined below:

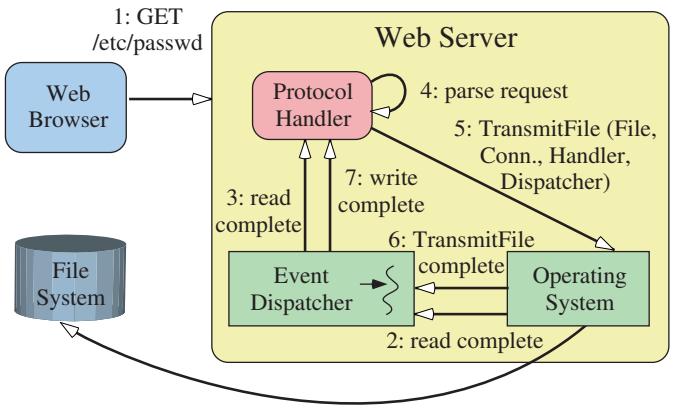

Figure 4: Client Sends requests to a Proactor-based Web Server

1. The client sends an HTTP `GET` request;
2. The read operation completes and the OS notifies the *Event Dispatcher*;
3. The *Event Dispatcher* notifies the *Protocol Handler* (steps 2 and 3 will repeat until the entire request has been received);
4. The *Protocol Handler* parses the request;
5. The *Protocol Handler* initiates an asynchronous `TransmitFile` operation to read the file data and write it out to the client connection. When doing so, it passes itself as a `Completion Handler` and a reference to the *Event Dispatcher* that will be used to notify the *Protocol Handler* upon completion of the asynchronous operation;
6. When the write operation completes, the OS notifies the *Event Dispatcher*;
7. The *Event Dispatcher* then notifies the `Completion Handler`

Evaluation: The primary advantage of using proactive event dispatching is that multiple operations can be initiated and run concurrently *without* requiring the application to have as many threads as there are simultaneous I/O operations. The operations are initiated asynchronously by the application and they run to completion within the I/O subsystem of the OS. Once the asynchronous operation is initiated, the thread that started the operation become available to service additional requests.

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

The primary drawback with the proactive event dispatching model is that the application structure and behavior can be considerably more complicated than with the conventional synchronous multi-threaded programming paradigm. In general, asynchronous applications are hard to develop since programmer's must explicitly retrieve OS notifications when asynchronous events complete. However, completion notifications need not appear in the same order that the asynchronous events were requested. Moreover, combining concurrency with asynchronous events is even harder since the thread that issues an asynchronous request may ultimately handle the completion of an event started by a different thread. The JAWS framework alleviates many of the complexities of concurrent asynchronous event dispatching by applying the Proactor pattern described in Section 4.

3.3 Benchmarking Results

To gain an understanding of how different concurrency and event dispatching mechanisms impact the performance of Web servers subject to heavy load conditions, JAWS was designed to use both synchronous and proactive I/O and event dispatching. Each one was benchmarked and the performance of each was compared.

3.3.1 Hardware Testbed

Our hardware testbed is shown in Figure 5.

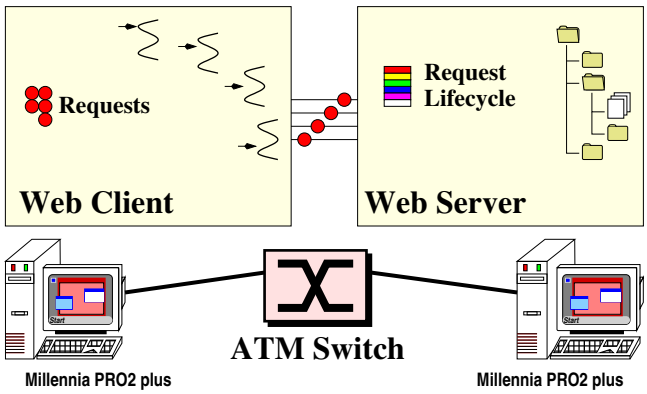

Figure 5: Benchmarking Testbed Overview

The testbed consists of two Micron Millennia PRO2 plus workstations. Each PRO2 has 128 MB of RAM and is equipped with 2 Pentium Pro processors. The client machine has a clock speed of 200 MHz, while the server machine runs 180 MHz. In addition, each PRO2 has an ENI-155P-MF-S ATM card made by Efficient Networks, Inc. and is driven by Orca 3.01 driver software. The two workstations were connected via an ATM network running through a FORE Systems ASX-200BX, with a maximum bandwidth of 622 Mbps. However, due to limitations of LAN emulation mode, the peak bandwidth of our testbed is approximately 120 Mbps.

3.3.2 Software Request Generator

We used the WebSTONE [17] v2.0 benchmarking software to collect client- and server-side metrics. These metrics included *average server throughput*, and *average client latency*. WebSTONE is a standard benchmarking utility, capable of generating load requests that simulate typical Web server file access patterns. Our experiments used WebSTONE to generate loads and gather statistics for particular file sizes to determine the impacts of different concurrency and event dispatching strategies.

The file access pattern used in the tests is shown in Table 1. This table represents actual load conditions on popular servers, based on a study of file access patterns conducted by SPEC [18].

Document Size	Frequency
500 bytes	35%
5 Kbytes	50%
50 Kbytes	14%
5 Mbytes	1%

Table 1: File Access Patterns

3.3.3 Experimental Results

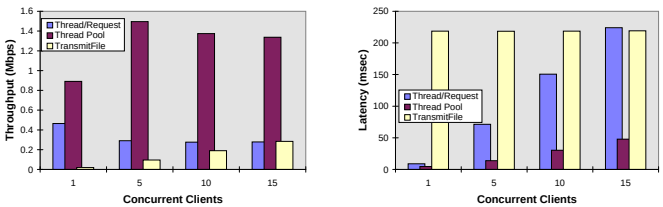

Figure 6: Experiment Results from 500 Byte File

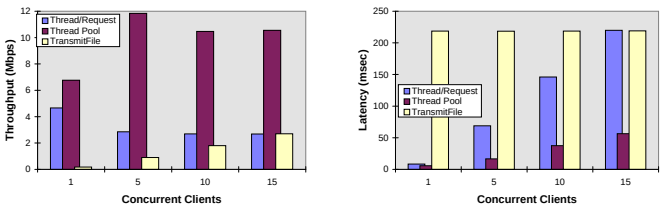

Figure 7: Experiment Results from 5K File

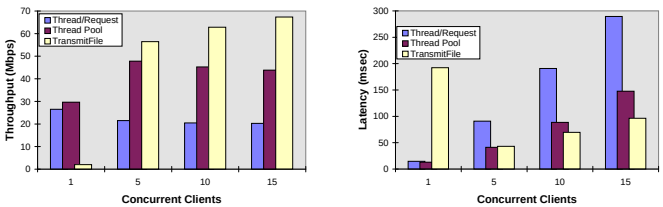

Figure 8: Experiment Results from 50K File

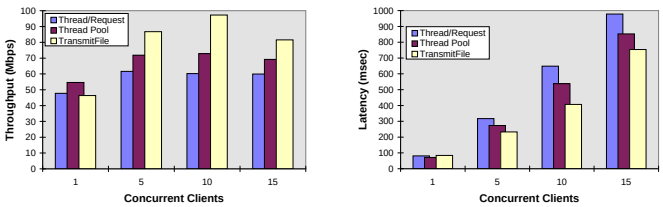

Figure 9: Experiment Results from 500K File

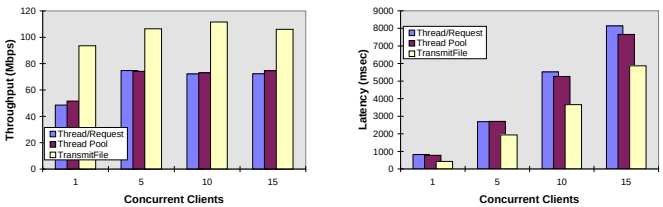

Figure 10: Experiment Results from 5M File

The results presented below compare the performance of several different adaptations of the JAWS Web server. We discuss the effect of different event dispatching and I/O models on throughput and latency. Throughput is defined as the average number of bits received per second by the client. A high-resolution timer for throughput measurement was started before the client benchmarking software sent the HTTP request. The high-resolution timer stops just after the connection is closed at the client end. The number of bits received

includes the HTML headers sent by the server.

Latency is defined as the average amount of delay in milliseconds seen by the client from the time it sends the request to the time it completely receives the file. It measures how long an end user must wait after sending an HTTP `GET` request to a Web server, and before the content begins to arrive at the client. The timer for latency measurement is started just before the client benchmarking software sends the HTTP request and stops just after the client receives the first response from the server.

The five graphs shown for each of throughput and latency represent different file sizes used in each experiment, 500 bytes through 5 Mbytes by factors of 10. These files sizes represent the spectrum of files sizes benchmarked in our experiments, to discover what impact file size has on performance.

Throughput Comparisons: Figures 6-10 demonstrate the variance of throughput as the size of the requested file and the server hit rate are systematically increased. As expected, the throughput for each connection generally degrades as the connections per second increases. This stems from the growing number of simultaneous connections being maintained, which decreases the throughput per connection.

As shown in Figure 8, the throughput of Thread-per-Request can degrade rapidly for smaller files as the connection load increases. In contrast, the throughput of the synchronous Thread Pool implementation degrade more gracefully. The reason for this difference is that Thread-per-Request incurs higher thread creation overhead since a new thread is spawned for each `GET` request. In contrast, thread creation overhead in the Thread Pool strategy is amortized by pre-spawning threads when the server begins execution.

The results in figures 6-10 illustrate that `TransmitFile` performs extremely poorly for small files (*i.e.,* $<$ 50 Kbytes). Our experiments indicate that the performance of `TransmitFile` depends directly upon the number of simultaneous requests. We believe that during heavy server loads (*i.e.*, high hit rates), `TransmitFile` is forced to wait while the kernel services incoming requests. This creates a high number of simultaneous connections, degrading server performance.

As the size of the file grows, however, `TransmitFile` rapidly outperforms the synchronous dispatching models. For instance, at heavy loads with the 5 Mbyte file (shown in Figure 10), it outperforms the next closest model by nearly 40%. `TransmitFile` is optimized to take advantage of Windows NT kernel features, thereby reducing the number of data copies and context switches.

Latency Comparisons: Figures 6-10 demonstrate the variance of latency performance as the size of the requested file and the server hit rate increase. As expected, as the connections per second increases, the latency generally increases, as well. This reflects the additional load placed on the server, which reduces its ability to service new client requests.

As before, `TransmitFile` performs extremely poorly for small files. However, as the file size grows, its latency rapidly improves relative to synchronous dispatching during light loads.

3.4 SUMMARY OF PERFORMANCE RESULTS

As illustrated in the benchmarking results presented above, there is significant variance in throughput and latency depending on the concurrency and event dispatching mechanisms. For small files, the synchronous Thread Pool strategy provides better overall performance. Under moderate loads, the synchronous event dispatching model provides slightly better latency than the asynchronous model. Under heavy loads and with large file transfers, however, the asynchronous model using `TransmitFile` provides better quality of service. Thus, under Windows NT, an optimal Web server should adapt itself to either event dispatching and file I/O model, depending on the server's workload and distribution of file requests.

Despite the potential for substantial performance improvements, it is considerably harder to develop a Web server that manages concurrency using asynchronous event dispatching, compared with traditional synchronous approaches. This is due to the additional details associated with asynchronous programming (*e.g.* explicitly retrieving OS notifications that may appear in non-FIFO order), and the added complexity of combining the approach with multi-threaded concurrency. Moreover, proactive event dispatching can be difficult to debug since asynchronous operations are often nondeterministic. Our experience with designing and developing a proactive Web server indicates that the *Proactor* pattern provides an elegant solution to managing these complexities.

4 THE PROACTOR PATTERN

In general, patterns help manage complexity by providing insight into known solutions to problems in a particular software domain. In the case of concurrent proactive architectures, the complexity of the additional details of asynchronous programming are compounded by the complexities associated with multi-threaded programming. Fortunately, patterns identified in software solutions to other proactive architectures have yielded the *Proactor* pattern, which is described below.[3]

4.1 INTENT

The Proactor pattern supports the demultiplexing and dispatching of multiple event handlers, which are triggered by the *completion* of asynchronous events. This pattern simplifies asynchronous application development by integrating the

[3]For brevity, portions of the complete description have been elided. Detailed coverage of implementation and sample code are available in [9].

demultiplexing of completion events and the dispatching of their corresponding event handlers.

4.2 APPLICABILITY

Use the Proactor pattern when one or more of the following conditions hold:

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

4.3 STRUCTURE AND PARTICIPANTS

The structure of the Proactor pattern is illustrated in Figure 11 using OMT notation.

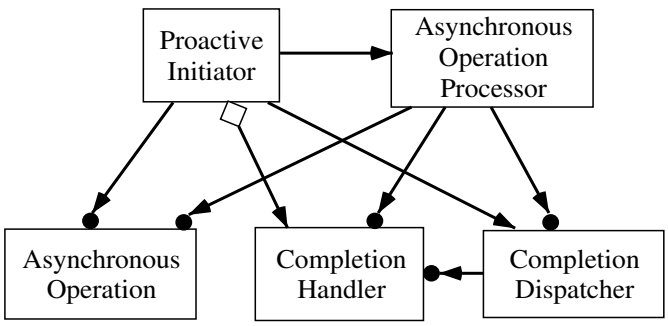

Figure 11: Participants in the Proactor Pattern

The key participants in the Proactor pattern include the following:

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

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

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

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

Asynchronous Operation (the methods `Async_Read`, `Async_Write`, and `Async_Accept`):

- `Asynchronous Operations` are used to execute requests (such as I/O and timer operations) on behalf of applications. When applications invoke `Asynchronous Operations`, the operations are performed *without* borrowing the application's thread of control.[4] Therefore, from the application's perspective, the operations are performed *asynchronously*. When `Asynchronous Operations` complete, the `Asynchronous Operation Processor` delegates application notifications to a `Completion Dispatcher`.

Asynchronous Operation Processor (the `Operating System`):

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

Completion Dispatcher (the `Notification Queue`):

- The `Completion Dispatcher` is responsible for calling back to the application's `Completion Handlers` when `Asynchronous Operations` complete. When the `Asynchronous Operation Processor` completes an asynchronously initiated operation, the `Completion Dispatcher` performs an application callback on its behalf.

4.4 COLLABORATIONS

There are several well-defined steps that occur for all `Asynchronous Operations`. At a high level of abstraction, applications initiate operations asynchronously and are notified when the operations complete. Figure 12 shows the following interactions that must occur between the pattern participants:

1. Proactive Initiators initiates operation: To perform asynchronous operations, the application initiates the operation on the `Asynchronous Operation Processor`. For instance, a Web server might ask the OS to transmit a file over the network using a particular socket connection. To request such an operation, the Web server must specify which

[4] In contrast, the reactive event dispatching model [10] steals the application's thread of control to perform the operation synchronously.

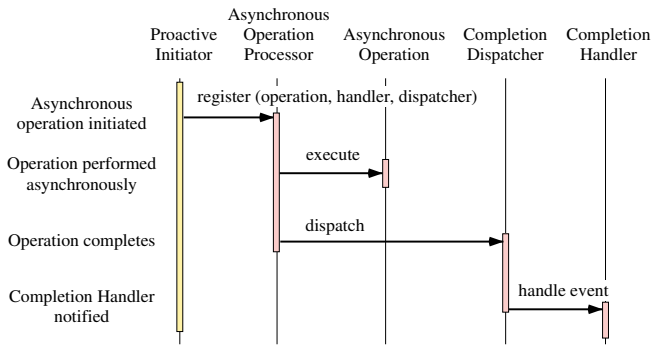

Figure 12: Interaction Diagram for the Proactor Pattern

file and network connection to use. Moreover, the Web server must specify (1) which `Completion Handler` to notify when the operation completes and (2) which `Completion Dispatcher` should perform the callback once the file is transmitted.

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

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

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

4.5 CONSEQUENCES

This section details the consequences of using the Proactor Pattern.

4.5.1 BENEFITS

The Proactor pattern offers the following benefits:

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

Improved application logic portability: It improves application portability by allowing its interface to be reused independently of the underlying OS calls that perform event demultiplexing. These system calls detect and report the events that may occur simultaneously on multiple event sources. Event sources may include I/O ports, timers, synchronization objects, signals, etc. On real-time POSIX platforms, the asynchronous I/O functions are provided by the `aio` family of APIs [19]. In Windows NT, I/O completion ports and overlapped I/O are used to implement asynchronous I/O [20].

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

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

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

Simplification of application synchronization: As long as `Completion Handlers` do not spawn additional threads

of control, application logic can be written with little or no regard to synchronization issues. `Completion Handlers` can be written as if they existed in a conventional single-threaded environment. For instance, a Web server's HTTP `GET` Handler can access the disk through an `Async Read` operation (such as the Windows NT `TransmitFile` function [15]).

4.5.2 DRAWBACKS

The Proactor pattern has the following drawbacks:

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

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

4.6 KNOWN USES

The following are some widely documented 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 `Completion Dispatcher`).

ACE Proactor: The Adaptive Communications Environment (ACE) [5] implements a Proactor component that encapsulates I/O Completion Ports on Windows NT. The ACE Proactor abstraction provides an OO interface to the standard C APIs supported by Windows NT. The source code for this implementation can be acquired from the ACE website at `www.cs.wustl.edu/~schmidt/ACE.html`.

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 [19]. These OS features are very similar to the ones described above for Windows NT. One difference is that UNIX signals can be used to implement an truly asynchronous `Completion Dispatcher` (the Windows NT API is not truly asynchronous).

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.

5 CONCLUDING REMARKS

Over the past several years, computer and network performance has improved substantially. However, the development of high-performance Web servers has remained expensive and error-prone. The JAWS framework described in this paper aims to support Web server developers by simplifying the application of various server designs and optimization strategies.

This paper illustrates that a Web server based on traditional synchronous event dispatching performs adequately under light server loads. However, when the Web server is subject to heavy loads, a design based on a concurrent proactive architecture provides significantly better performance. However, programming this model adds complexity to the software design and increases the effort of developing high-performance Web servers.

Much of the development effort stems from the repeated rediscovery and reinvention of fundamental design patterns. Design patterns describe recurring solutions found in existing software systems. Applying design patterns to concurrent software systems can reduce software development time, improve code maintainability, and increase code reuse over traditional software engineering techniques.

The Proactor pattern described in this paper embodies a powerful technique that supports both efficient and flexible event dispatching strategies for high-performance concurrent applications. In general, applying this pattern enables developers to leverage the performance benefits of executing operations concurrently, without constraining them to synchronous multi-threaded or reactive programming. In our experience, applying the Proactor pattern to the JAWS Web server framework has made it considerably easier to design, develop, test, and maintain.

REFERENCES

[1] D. C. Schmidt and J. Hu, "Developing Flexible and High-performance Web Servers with Frameworks and Patterns," *ACM Computing Surveys*, vol. 30, 1998.

[2] 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.

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

[4] D. C. Schmidt, "Experience Using Design Patterns to Develop Reuseable Object-Oriented Communication Software," *Communications of the ACM (Special Issue on Object-Oriented Experiences)*, vol. 38, October 1995.

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

[7] P. Jain and D. C. Schmidt, "Service Configurator: A Pattern for Dynamic Configuration of Services," in *Proceedings of the 3^{rd} Conference on Object-Oriented Technologies and Systems*, USENIX, June 1997.

[8] R. G. Lavender and D. C. Schmidt, "Active Object: an Object Behavioral Pattern for Concurrent Programming," in *Pattern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), Reading, MA: Addison-Wesley, 1996.

[9] T. Harrison, I. Pyarali, D. C. Schmidt, and T. Jordan, "Proactor – An Object Behavioral Pattern for Dispatching Asynchronous Event Handlers," in *The 4^{th} Pattern Languages of Programming Conference (Washington University technical report #WUCS-97-34)*, September 1997.

[10] 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.

[11] 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.

[12] Object Management Group, *Control and Management of Audio/Video Streams: OMG RFP Submission*, 1.2 ed., Mar. 1997.

[13] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture - A System of Patterns*. Wiley and Sons, 1996.

[14] D. C. Schmidt, "GPERF: A Perfect Hash Function Generator," in *Proceedings of the 2^{nd} C++ Conference*, (San Francisco, California), pp. 87–102, USENIX, April 1990.

[15] 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.

[16] 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.

[17] Gene Trent and Mark Sake, "WebSTONE: The First Generation in HTTP Server Benchmarking." Silicon Graphics, Inc. whitepaper, February 1995. ww.sgi.com/.

[18] A. Carlton, "An Explanation of the SPECweb96 Benchmark." Standard Performance Evaluation Corporation whitepaper, 1996. www.specbench.org/.

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

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