JAWS: A Framework for High-performance Web Servers

James C. Hu
jxh@cs.wustl.edu

Douglas C. Schmidt
schmidt@cs.wustl.edu

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

Abstract

Developers of communication software face many challenges. Communication software contains both inherent complexities, such as fault detection and recovery, and accidental complexities, such as the continuous re-rediscovery and re-invention of key concepts and components. Meeting these challenges requires a thorough understanding of object-oriented application frameworks and patterns. This paper illustrates how we have applied frameworks and patterns for communication software to develop a high-performance Web server called JAWS.

JAWS is an object-oriented framework that supports the configuration of various Web server strategies, such as a Thread Pool concurrency model with asynchronous I/O and LRU caching vs.. a Thread-per-Request concurrency model with synchronous I/O and LFU caching. Because JAWS is a framework, these strategies can be customized systematically and measured both independently and collaboratively to determine the best strategy profiles. Using these profiles, JAWS can statically and dynamically adapt its behavior to deploy the most effective strategies for a given software/hardware platform and workload. JAWS' adaptive software features make it a powerful application framework for constructing high-performance Web servers.

1 Introduction

During the past several years, the volume of traffic on the World Wide Web (Web) has grown dramatically. Traffic increases are due largely to the proliferation of inexpensive and ubiquitous Web browsers such as NCSA Mosaic, Netscape Navigator, and Internet Explorer. Likewise, Web protocols and browsers are increasingly applied to specialized computationally expensive tasks, such as image processing servers used by Siemens [1] and Kodak [2] and database search engines such as AltaVista and Lexis-Nexis.

To keep pace with increasing demand, it is essential to develop high-performance Web servers. However, developers face a remarkably rich set of design strategies when configuring and optimizing Web servers. For instance, developers must select from among a wide range of concurrency models (such as Thread-per-Request *vs.* Thread Pool), dispatching models (such as synchronous vs. asynchronous dispatching), file caching models (such as LRU vs. LFU), and protocol processing models (such as HTTP/1.0 vs. HTTP/1.1). No single configuration is optimal for all hardware/software platform and workloads [1, 3].

The existence of all these alternative strategies ensures that Web servers can be tailored to their users' needs. However, navigating through many design and optimization strategies is tedious and error-prone. Without guidance on which design and optimization strategies to apply, developers face the herculean task of engineering Web servers from the ground up, resulting in *ad hoc* solutions. Such systems are often hard to maintain, customize, and tune since much of the engineering effort is spent just trying to get the system operational.

1.1 Definitions

We will frequently refer to the terms *OO class library*, *framework*, *pattern*, and *component*. These terms refer to tools that are used to build reusable software systems. An OO class libarary is a collection of software object implemetations that provide reusable functionality as the user calls into the object methods. A framework is a reusable, "semi-complete" application that can be specialized to produce custom applications [4]. A pattern represents a recurring solution to a software development problem within a particular context [5]. A component refers to a "reifiable" object. Both OO class libraries and frameworks are collections of components that are reified by instantiation and specialization. A pattern component is reified through codification.

*The work described here was made possible by funds from Siemens AG, Eastman Kodak, and NSF Research Grant NCR-9628218.

1.2 Overview

This paper illustrates how to produce flexible and efficient Web servers using OO *application frameworks* and *design patterns*. Patterns and frameworks can be applied synergistically to improve the efficiency and flexibility of Web servers. Patterns capture abstract designs and software architectures of high-performance and adaptive Web servers in a systematic and comprehensible format. Frameworks capture the concrete designs, algorithms, and implementations of Web servers in a particular programming language, such as C++ or Java. In contrast, *OO class libraries* provide the raw materials necessary to build applications, but no guidance as to how to put the pieces together.

This paper focuses on the patterns and framework components used to develop a high-performance Web server called JAWS [1, 3]. JAWS is both a Web server and a framework from which other types of servers can be built. The JAWS framework itself was developed using the ACE framework [6, 7]. The ACE framework reifies key patterns [5] in the domain of communication software. The framework and patterns in JAWS and ACE are representative of solutions that have been applied successfully to communication systems ranging from telecommunication system management [8] to enterprise medical imaging [2] and real-time avionics [9].

This paper is organized as follows: Section 2 presents an overview of patterns and frameworks and motivates the need for the type of communication software framework provided by JAWS; Section 3 illustrates how patterns and components can be applied to develop high-performance Web servers; Section 4 compares the performance of JAWS over high-speed ATM networks with other high-performance Web servers; and Section 5 presents concluding remarks.

2 Applying Patterns and Frameworks to Web Servers

There is increasing demand for high-performance Web servers that can provide services and content to the growing number of Internet and intranet users. Developers of Web servers strive to build fast, scalable, and configurable systems. This task can be difficult, however, if care is not taken to avoid common traps and pitfalls, which include tedious and error-prone low-level programming details, lack of portability, and the wide range of design alternatives. This section presents a road-map to these hazards. We then describe how patterns and frameworks can be applied to avoid these hazards by allowing developers to leverage both design and code reuse.

2.1 Common Pitfalls of Developing Web Server Software

Developers of Web servers confront recurring challenges that are largely independent of their specific application requirements. For instance, like other communication software, Web servers must perform various tasks related to the following: connection establishment, service initialization, event demultiplexing, event handler dispatching, interprocess communication, memory management and file caching, static and dynamic component configuration, concurrency, synchronization, and persistence. In most Web servers, these tasks are implemented in an *ad hoc* manner using low-level native OS application programming interfaces (APIs), such as the Win32 or POSIX, which are written in C.

Unfortunately, native OS APIs are not an effective way to develop Web servers or other types of communication middleware and applications [10]. The following are common pitfalls associated with the use of native OS APIs:

Excessive low-level details: Building Web servers with native OS APIs requires developers to have intimate knowledge of low-level OS details. Developers must carefully track which error codes are returned by each system call and handle these OS-specific problems in their server. Such details divert attention from the broader, more strategic issues, such as semantics and program structure. For example, UNIX developers who use the `wait` system call must distinguish the between return errors due to no child processes being present and errors from signal interrupts. In the latter case, the `wait` must be reissued.

Continuous rediscovery and reinvention of incompatible higher-level programming abstractions: A common remedy for the excessive level of detail with OS APIs is to define higher-level programming abstractions. For instance, many Web servers create a file cache to avoid accessing the filesystem for each client request. However, these types of abstractions are often rediscovered and reinvented independently by each developer or project. This *ad hoc* process hampers productivity and creates incompatible components that are not readily reusable within and across projects in large software organizations.

High potential for errors: Programming to low-level OS APIs is tedious and error-prone due to their lack of typesafety. For example, most Web servers are programmed with the Socket API [11]. However, endpoints of communication in the Socket API are represented as untyped handles. This increases the potential for subtle programming mistakes and run-time errors.

Lack of portability: Low-level OS APIs are notoriously non-portable, even across releases of the same OS. For instance, implementations of the Socket API on Win32 platforms (WinSock) are subtly different than on UNIX platforms. Moreover, even WinSock implementations on different versions of Windows NT possess incompatible timing-related bugs that cause sporadic failures when performing non-blocking connections.

Steep learning curve: Due to the excessive level of detail, the effort required to master OS-level APIs can be very high. For instance, it is hard to learn how to program with POSIX asynchronous I/O [12] correctly. It is even harder to learn how to write a *portable* application using asynchronous I/O mechanisms since they differ widely across OS platforms.

Inability to handle increasing complexity: OS APIs define basic interfaces to mechanisms like process and thread management, interprocess communication, file systems, and memory management. However, these basic interfaces do not scale up gracefully as applications grow in size and complexity. For instance, a typical UNIX process allows a backlog of only $\sim$7 pending connections [13]. This number is inadequate for heavily accessed Web servers that must handle hundreds of simultaneous clients.

2.2 Overcoming Web Server Pitfalls with Patterns and Frameworks

Software reuse is a a widely touted method of reducing development effort. Reuse leverages the domain knowledge and prior effort of experienced developers. When applied effectively, reuse can avoid re-creating and re-validating common solutions to recurring application requirements and software design challenges.

Java's `java.lang.net` and RogueWave `Net.h++` are two common examples of applying reusable OO class libraries to communication software. Although class libraries effectively support component reuse-in-the-small, their scope is overly constrained. In particular, class libraries do not capture the canonical control flow and collaboration among families of related software components. Thus, developers who apply class library-based reuse often re-invent and re-implement the overall software architecture for each new application.

A more powerful way to overcome the pitfalls described above is to identify the *patterns* that underlie proven Web servers and to reify these patterns in *object-oriented application frameworks*. Patterns and frameworks help alleviate the continual re-discovery and re-invention of key Web server concepts and components by capturing solutions to common software development problems [5].

The benefits of patterns for Web servers: Patterns document the structure and participants in common Web server micro-architectures. For instance, the Reactor [14] and Active Object [15] patterns are widely used as Web server dispatching and concurrency strategies, respectively. These patterns are generalizations of object-structures that have proven useful to build flexible and efficient Web servers.

Traditionally, these types of patterns have either been locked in the heads of the expert developers or buried deep within the source code. Allowing this valuable information to reside only in these locations is risky and expensive, however. For instance, the insights of experienced Web server designers will be lost over time if they are not documented. Likewise, substantial effort may be necessary to reverse engineer patterns from existing source code. Therefore, capturing and documenting Web server patterns explicitly is essential to preserve design information for developers who enhance and maintain existing software. Moreover, knowledge of domain-specific patterns helps guide the design decisions of developers who are building new servers in other domains.

The benefits of frameworks for Web servers: Knowledge of patterns helps to reduce development effort and maintenance costs. However, reuse of patterns alone is not sufficient to create flexible and efficient Web server software. While patterns enable reuse of abstract design and architecture knowledge, abstractions documented as patterns do not directly yield reusable code [16]. Therefore, it is essential to augment the study of patterns with the creation and use of application frameworks. Frameworks help developers avoid costly re-invention of standard Web server components by implementing common design patterns and factoring out common implementation roles.

2.3 Relationship Between Frameworks, Patterns, and Other Reuse Techniques

Frameworks provide reusable software components for applications by integrating sets of abstract classes and defining standard ways that instances of these classes collaborate [4]. In general, the components are not self-contained, since they usually depend upon functionality provided by other components within the framework. However, the collection of these components forms a partial implementation, *i.e.*, an application skeleton. This skeleton can be customized by inheriting and instantiating from reusable components in the framework.

The scope of reuse in a Web server framework can be significantly larger than using traditional function libraries or OO class libraries of components. In particular, the JAWS framework described in Section 3 is tailored for a wide range of Web server tasks. These tasks include service initialization, error handling, flow control, event processing, file caching, concurrency control, and prototype pipelining. It is important to recognize that these tasks are also reusable for many other types of communication software.

In general, frameworks and patterns enhance reuse techniques based on class libraries of components in the following ways.

Frameworks define "semi-complete" applications that embody domain-specific object structures and functionality: Class libraries provide a relatively small granularity of reuse. For instance, the classes in Figure 1 are typically low-level, relatively independent, and general-purpose components like Strings, complex numbers, arrays, and bit sets.

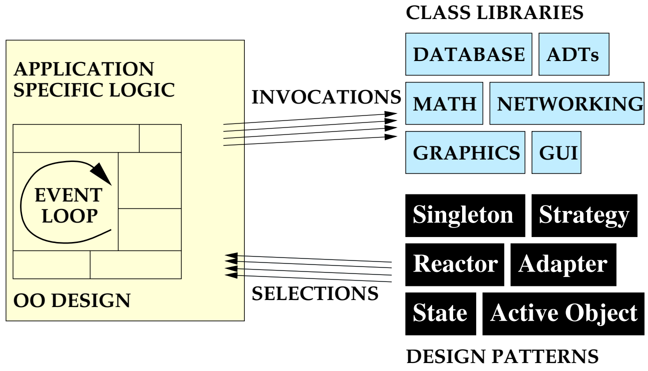

Figure 1: Class Library Component Architecture

In contrast, components in a framework collaborate to provide a customizable architectural skeleton for a family of related applications. Complete applications can be composed by inheriting from and/or instantiating framework components. As shown in Figure 2, frameworks reduce the amount of application-specific code since much of the domain-specific processing is factored into generic framework components.

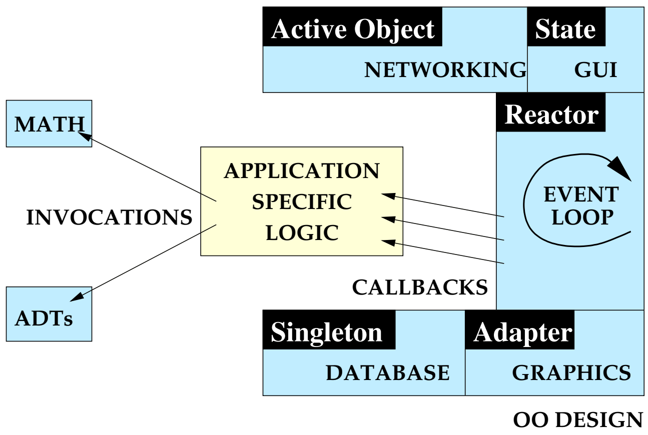

Figure 2: Application Framework Component Architecture

Frameworks are active and exhibit "inversion of control" at run-time: Class library components generally behave *passively*. In particular, class library components often perform their processing by borrowing the thread(s) of control from application objects that are "self-directed." Because application objects are self-directed, application developers are largely responsible for deciding how to combine the components and classes to form a complete system. For instance, the code to manage an event loop and determine the flow of control among reusable and application-specific components is generally rewritten for each new application.

The typical structure and dynamics of applications built with class libraries and components is illustrated in Figure 1. This figure also illustrates how design patterns can help guide the design, implementation, and use of class library components. Note, that the existence of class libraries, while providing tools to solve particular tasks (*e.g.*, establishing a network connection) do not offer explicit guidance to system design. In particular, software developers are solely responsible for identifying and applying patterns in designing their applications.

In contrast to class libraries, components in a framework are more *active*. In particular, they manage the canonical flow of control within an application via event dispatching patterns like Reactor [14] and Observer [5]. The callback-driven runtime architecture of a framework is shown in Figure 2.

Figure 2 illustrates a key characteristic of a framework: its "inversion of control" at run-time. This design enables the canonical application processing steps to be customized by event handler objects that are invoked via the framework's reactive dispatching mechanism [14]. When events occur, the framework's dispatcher reacts by invoking hook methods on pre-registered handler objects, which perform application-specific processing on the events.

Inversion of control allows the framework, rather than each application, to determine which set of application-specific methods to invoke in response to external events (such as HTTP connections and data arriving on sockets). As a result, the framework reifies an integrated set of patterns, which are pre-applied into collaborating components. This design reduces the burden for software developers.

In practice, frameworks, class libraries, and components are complementary technologies. Frameworks often utilize class libraries and components internally to simplify the development of the framework. For instance, portions of the JAWS framework use the string and vector containers provided by the C++ Standard Template Library [17] to manage connection maps and other search structures. In addition, application-specific callbacks invoked by framework event handlers frequently use class library components to perform basic tasks such as string processing, file management, and numerical analysis.

To illustrate how OO patterns and frameworks have been applied successfully to develop flexible and efficient communication software, the remainder of this paper examines the structure, use, and performance of the JAWS framework.

3 The JAWS Adaptive Web Server

The benefits of applying frameworks and patterns to communication software are best illustrated by example. This section describes the structure and functionality of the JAWS. JAWS is a high-performance and adaptive Web server that implements the HTTP protocol. It is also a platform-independent application framework from which other types of communication servers can be built.

3.1 Overview of the JAWS Framework

Figure 3 illustrates the major structural components and design patterns that comprise the JAWS Adaptive Web Server (JAWS) framework. JAWS is designed to allow various Web

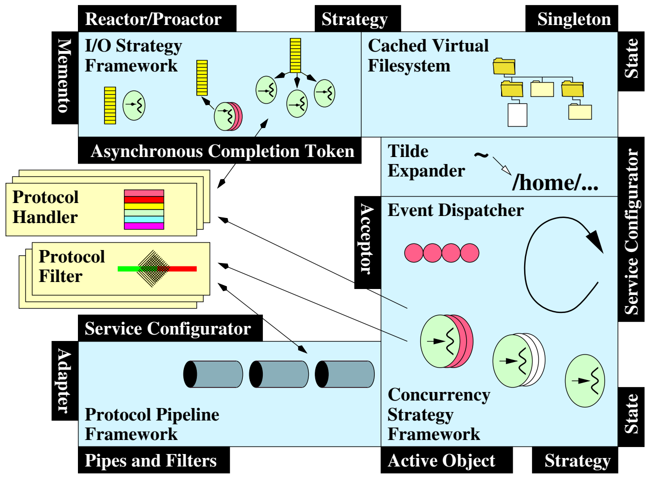

Figure 3: Architectural Overview of the JAWS Framework

server strategies to be customized in response to environmental factors. These factors include *static* factors, such as support for kernel-level threading and/or asynchronous I/O in the OS, and the number of available CPUs, as well as *dynamic* factors, such as 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 [18]. The collaborations among JAWS components and frameworks are guided by a family of patterns, which are listed along the borders in Figure 3. An outline of the key frameworks, components, and patterns in JAWS is presented below. A more detailed description of how these patterns have been applied to JAWS' design will be shown in Section 3.2.

Event Dispatcher: This component is responsible for coordinating JAWS' *Concurrency Strategy* with its *I/O Strategy*. The passive establishment of connections with Web clients follows the *Acceptor* pattern [19]. New incoming requests 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. The ability to dynamically bind to a particular concurrency strategy and I/O strategy from a range of alternatives follows the *Strategy* pattern [5].

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 [5] or pre-determined at initialization-time. The *Service Configurator* pattern [20] 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 [15].

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. Asynchronous I/O is implemented via the *Proactor* [21] and *Asynchronous Completion Token* [22] patterns. Reactive I/O is accomplished through the *Reactor* pattern [14]. Reactive I/O utilizes the *Memento* pattern [5] 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 remain opaque to the protocol handler by following the *Adapter* [5] pattern. 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 and DICOM) 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 [23] for input processing. Pipeline components can be linked dynamically at run-time using the *Service Configurator* pattern.

Cached Virtual Filesystem: The component improves Web server performance by reducing the overhead of filesystem accesses. Various caching strategies, such as LRU, LFU, Hinted, and Structured, can be selected following the *Strategy* pattern [5]. This allows different caching strategies to be profiled for effectiveness and enables optimal strategies to be configured statically or dynamically. The cache for each Web server is instantiated using the *Singleton* pattern [5].

Tilde Expander: This component is another cache component that uses a perfect hash table [24] 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, for example.

3.2 Overview of the Design Patterns in JAWS

The JAWS architecture diagram in Figure 3 illustrates *how* JAWS is structured, but not *why* it is structured this particular way. To understand why JAWS contains frameworks and components like the *Concurrency Strategy*, *I/O Strategy*, *Protocol Handler*, and *Event Dispatcher* requires a deeper understanding of the design patterns underlying the domain of communication software, in general, and Web servers, in particular. Figure 4 illustrates the *strategic* and *tactical* patterns related to JAWS. These patterns are summarized below.

3.2.1 Strategic Patterns

The following patterns are *strategic* to the overall software architecture of Web servers. Their use widely impacts the level of interactions of a large number of components in the system. These patterns are also widely used to guide the architecture of many other types of communication software.

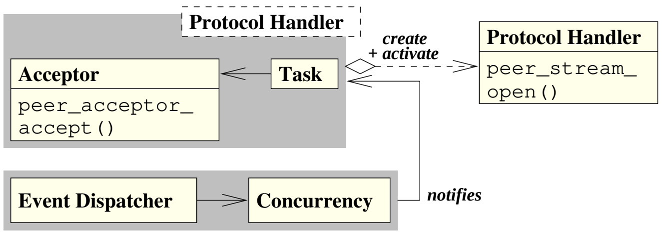

Figure 5: Structure of the Acceptor Pattern in JAWS

The Acceptor pattern: This pattern decouples passive connection establishment from the service performed once the connection is established [19]. JAWS uses the Acceptor pattern to adaptively change its concurrency and I/O strategies independently from its connection management strategy. Figure 5 illustrates the structure of the Acceptor pattern in the context of JAWS. The *Acceptor* is a factory [5]. It creates, accepts, and activates a new *Protocol Handler* whenever the *Event Dispatcher* notifies it that a connection has arrived from a client.

The Reactor pattern: This pattern decouples the synchronous event demultiplexing and event handler notification dispatching logic of server applications from the service(s) performed in response to events [14]. JAWS uses the Reactor pattern to process multiple synchronous events from multiple sources of events, *without* polling all event sources or blocking indefinitely on any single source of events. Figure 6 illustrates the structure of the Reactor pattern in the context of JAWS.

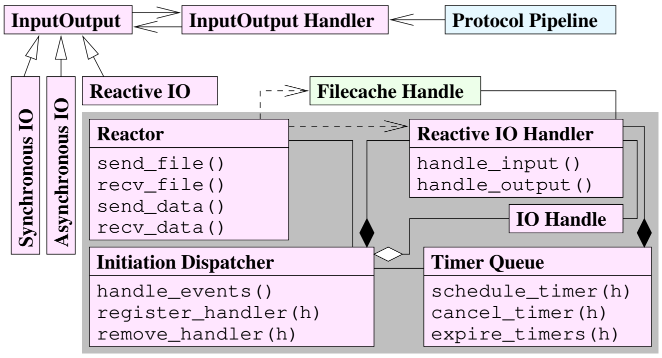

Figure 6: Structure of the Reactor Pattern in JAWS

JAWS *Reactive IO Handler* objects register themselves with the *Initiation Dispatcher* for events, *i.e.*, input and output through connections established by HTTP requests. The *Initiation Dispatcher* invokes the `handle_input` notification hook method of these *Reactive IO Handler* objects when their associated events occur. The Reactor pattern is used by the Single-Threaded Web server concurrency model presented in Section 3.3.2.

The Proactor pattern pattern: This pattern decouples the asynchronous event demultiplexing and event handler completion dispatching logic of server applications from the service(s) performed in response to events [21]. JAWS uses the Proactor pattern to perform server-specific processing, such as parsing the headers of a request, while asynchronously processing other I/O events. Figure 7 illustrates the structure of the Proactor pattern in the context of JAWS. JAWS *Proactive IO Handler* objects register themselves with the *Completion*

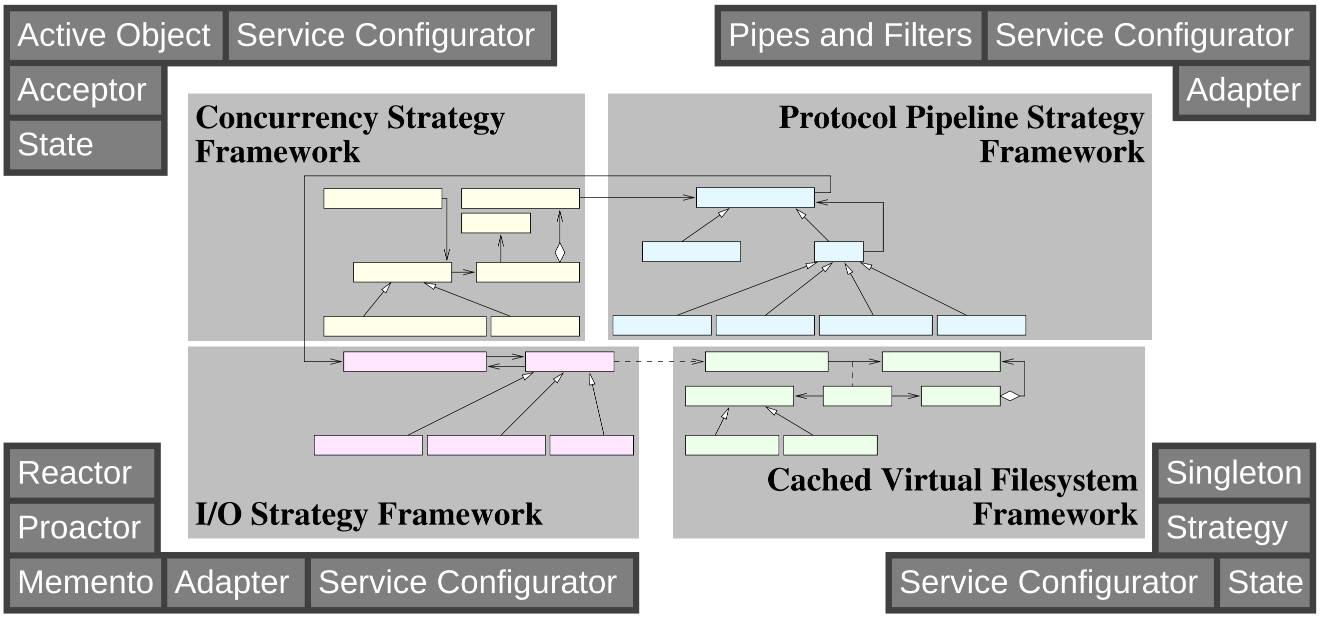

Figure 4: Design Patterns Used in the JAWS Framework

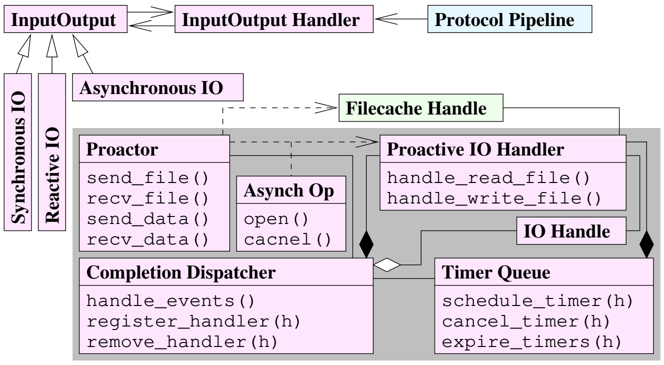

Figure 7: Structure of the Proactor Pattern in JAWS

Dispatcher for events (*i.e.*, receipt and delivery of files through connections established by HTTP requests).

The primary difference between the Reactor and Proactor patterns is that the *Proactive IO Handler* defines *completion* hooks, whereas the *Reactive IO Handler* defines *initiation* hooks. Therefore, when asynchronously invoked operations like `recv_file` or `send_file` complete, the *Completion Dispatcher* invokes the appropriate completion hook method of these *Proactive IO Handler* objects. The Proactor pattern is used by the asynchronous variant of the Thread Pool in Section 3.3.2.

The Active Object pattern: This pattern decouples method invocation from method execution, allowing methods to run concurrently [15]. JAWS uses the Active Object pattern to execute client requests concurrently in separate threads of control. Figure 8 illustrates the structure of the Active Object pattern in the context of JAWS.

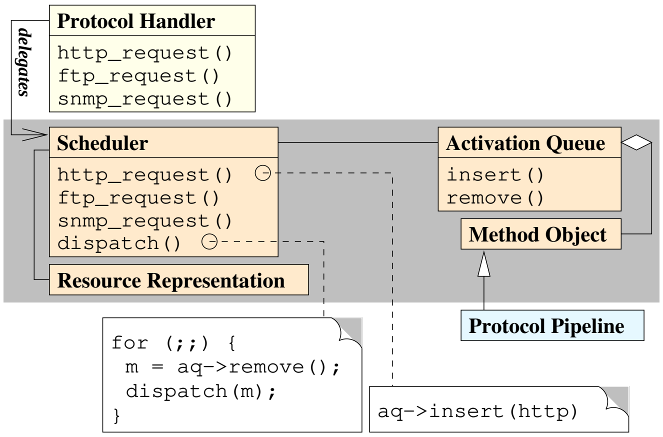

Figure 8: Structure of the Active Object Pattern in JAWS

The *Protocol Handler* issues requests to the *Scheduler*, which transforms the request method (such as an HTTP request) into *Method Objects* that are stored on an *Activation Queue*. The *Scheduler*, which runs in a separate thread from the client, dequeues these *Method Objects* and transforms them back into method calls to perform the specified protocol. The Active Object pattern is used in the Thread-per-Request, Thread Pool, and Thread-per-Session concurrency models described in Section 3.3.2.

The Service Configurator pattern: This pattern decouples the implementation of individual components in a system from the *time* when they are configured into the system. JAWS uses the Service Configurator pattern to dynamically optimize, control, and reconfigure the behavior of Web server strategies at installation-time or during run-time [25]. Figure 9 illustrates the structure of the Service Configurator pattern in the context of the *Protocol Pipeline Filters* and *Caching Strategies*.

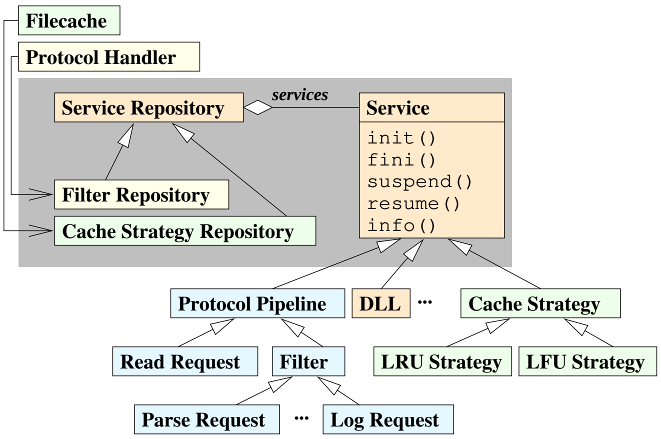

Figure 9: Structure of the Service Configurator Pattern in JAWS

The figure depicts how the *Service Configurator* can dynamically manage *DLLs*, which are dynamically linked libraries. This allows the framework to dynamically configure different implementations of server strategies at run-time. The *Filter Repository* and *Cache Strategy Repository* inherit functionality from the *Service Repository*. Likewise, the strategy implementations (such as *Parse Request* and *LRU Strategy*) borrow interfaces from the *Service* component of the pattern so that they can be managed dynamically by the repository.

3.2.2 Tactical Patterns

Web servers also utilize many *tactical* patterns, which are more ubiquitous and domain-independent than the strategic patterns described above. The following tactical patterns are used in JAWS:

The Strategy pattern: This pattern defines a family of algorithms, encapsulates each one, and make them interchangeable [5]. JAWS uses this pattern extensively to selectively configure different cache replacement strategies without affecting the core software architecture of the Web server.

The Adapter pattern: This pattern transforms a non-conforming interface into one that can be used by a client [5]. JAWS uses this pattern in its I/O Strategy Framework to uniformly encapsulate the operations of synchronous, asynchronous and reactive I/O operations.

The State pattern: This pattern defines a composite object whose behavior depends upon its state [5]. The `Event Dispatcher` component in JAWS uses the State pattern to seamlessly support different concurrency strategies and both synchronous and asynchronous I/O.

The Singleton pattern: This pattern ensures a class only has one instance and provides a global point of access to it [5]. JAWS uses a Singleton to ensure that only one copy of its Cached Virtual Filesystem exists in a Web server process.

In contrast to the strategic patterns described earlier, tactical pattern have a relatively localized impact on a software design. For instance, Singleton is a tactical pattern that is often used to consolidate certain globally accessible resources in a Web server. Although this pattern is domain-independent and thus widely applicable, the problem it addresses does not impact Web server software architecture as pervasively as strategic patterns like the Active Object and Reactor. A thorough understanding of tactical patterns is essential, however, to implement highly flexible software that is resilient to changes in application requirements and platform characteristics.

The remainder of this section discusses the structure of JAWS' frameworks for concurrency, I/O, protocol pipelining, and file caching. For each framework, we describe the key design challenges and outline the range of alternatives solution strategies. Then, we explain how each JAWS framework is structured to support the configuration of alternative strategy profiles.

3.3 Concurrency Strategies

3.3.1 Design Challenges

Concurrency strategies impact the design and performance of a Web system significantly. Empirical studies [3] of existing Web servers, including Roxen, Apache, PHTTPD, Zeus, Netscape and the Java Web server, indicate that a large portion of non-I/O related Web server overhead is due to the Web server's concurrency strategy. Key overheads include synchronization, thread/process creation, and context switching. Therefore, choosing an efficient concurrency strategy is crucial to achieve high-performance.

3.3.2 Alternative Solution Strategies

Selecting the right concurrency strategy is nontrivial. The factors influencing the decision are both static and dynamic. *Static* factors can be determined *a priori*. These factors include hardware configuration (*e.g.*, number of processors, amount of memory, and speed of network connection), OS platform (*e.g.*, availability of threads and asynchronous I/O), and Web server use case (*e.g.*, database interfacing, image server, or HTML server). *Dynamic* factors are those detectable and measurable conditions that occur during the execution of the system. These factors include machine load, number of simultaneous requests, dynamic memory use, and server workload.

Existing Web servers use a wide range of concurrency strategies, reflecting the numerous factors involved. These strategies include single-threaded concurrency (*e.g.*, Roxen), process-based concurrency (*e.g.*, Apache and Zeus), and multi-threaded concurrency (*e.g.*, PHTTPD, and JAWS). Each strategy offers positive and negative benefits, which must be analyzed and evaluated in the context of both static and dynamic factors. These tradeoffs are summarized below.

Thread-per-Request: This model handles each request from a client in a separate thread of control. Thus, as each request arrives, a new thread is created to process the request. This design allows each thread to use well understood synchronous I/O mechanisms to read and write the requested file. Figure 10 illustrates this model in the context of the JAWS framework. Here, there is an *Acceptor* thread that iterates on waiting for a connection, creating a *Protocol Handler*, and spawning a new thread so that the handler can continue processing the connection.

The advantage of Thread-per-Request is its simplicity and its ability to exploit parallelism on multi-processor platforms. Its chief drawback is lack of scalability, *i.e.*, the number of running threads may grow without bound, exhausting available memory and CPU resources. Therefore, Thread-per-Request is adequate for lightly loaded servers with low latency. However, it may be unsuitable for servers that are accessed frequently to perform time consuming tasks.

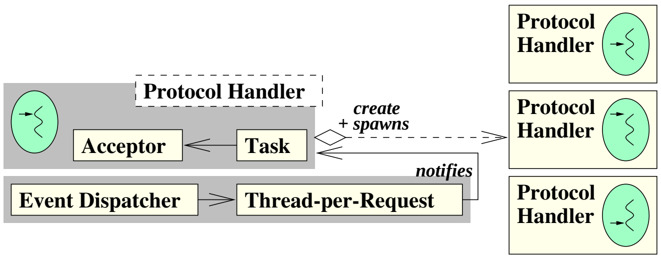

Figure 10: Thread-per-Request Strategy in JAWS

Thread-per-Session: A *session* is a series of requests that one client makes to a server. In Thread-per-Session, all of these requests are submitted through a connection between each client and a separate thread in the Web server process. Therefore, this model amortizes both thread creation and connection establishment costs across multiple requests.

Thread-per-Session is less resource intensive than Thread-per-Request since it does not spawn a separate thread for each request. However, it is still vulnerable to unbounded resource consumption as the number of clients grows. Also, the use of Thread-per-Session requires both the client and server to support the concept of re-using an established connection with multiple requests. For example, if both the Web client and Web server are HTTP/1.1 compliant, Thread-per-Session can be used between them. However, if either the client or server only supports HTTP/1.0, then Thread-per-Session degrades to Thread-per-Request [26, 27].

Thread Pool: In this model, a group of threads are prespawned during Web server initialization. Each thread from the pool retrieves a task from a job queue. While the thread is processing the job, it is removed from the pool. Upon finishing the task, the thread is returned to the pool. As illustrated in Figure 11, the job being retrieved is completion of the *Acceptor*. When it completes, the thread creates a *Protocol Handler* and lends its thread of control so that the handler can process the connection.

Thread Pool has lower overhead than Thread-per-Request since the cost of thread creation is amortized through prespawning. Moreover, the number of resources the pool can consume is bounded since the pool size is fixed. However, if the pool is too small, it can be depleted. This causes new incoming requests to be dropped or wait indefinitely. Furthermore, if the pool is too large, resource consumption may be no better than using Thread-per-Request.

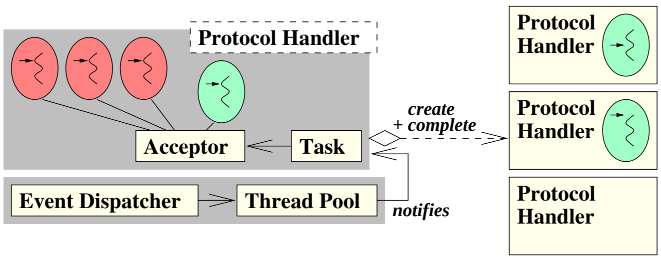

Figure 11: Thread Pool Strategy in JAWS

Single-Threaded: In this model, all connections and requests are handled by the same thread of control. Simple implementations of Single-Threaded servers process requests iteratively. This is usually inadequate for high volume production servers since subsequent requests are blocked until they are reached, creating unacceptable delays.

More sophisticated Single-Threaded implementations attempt to process multiple requests concurrently using asynchronous or reactive I/O (which are described in Section 3.4). Single-threaded concurrency strategies can perform better than multi-threaded solutions on uni-processor machines that support asynchronous I/O [1]. Since JAWS' I/O framework is orthogonal to its concurrency framework, we consider the Single-Threaded concurrency strategy as a special case of the Thread Pool where the pool size is 1.

Experiments in [1] and [3] demonstrate that the choice of concurrency and event dispatching strategies significantly affect the performance of Web servers that experience varying load conditions. In particular, no single server strategy provides optimal performance for all cases. Thus, a server framework should provide at least two degrees of freedom:

1. *Static adaptivity* – The framework should allow the Web server developers to select the concurrency strategy that best meets the static requirements of the system. For example, a multi-processor machine may be more suited to multi-threaded concurrency than a uni-processor machine.
2. *Dynamic adaptivity* – The framework should allow its concurrency strategy to adapt dynamically to current server conditions to achieve optimal performance in the presence of dynamic server load conditions. For instance, in order to accommodate unanticipated load usage, it may be necessary to increase the number of available threads in the thread pool.

3.3.3 JAWS Concurrency Strategy Framework

As discussed above, no single concurrency strategy performs optimally under for all conditions. However, not all platforms can make effective use of all available concurrency strategies. To address these issues, the JAWS Concurrency Strategy Framework supports both static and dynamic adaptivity with respect to its concurrency and event dispatching strategies.

Figure 12 illustrates the OO design of JAWS' Concurrency Strategy framework. The *Event Dispatcher* and *Concurrency* objects interact according to the *State* pattern. As illustrated, `server` can be changed to either *Thread-per-Connection* or *Thread Pool* between successive calls to `server->dispatch()`, allowing different concurrency mechanisms to take effect. The *Thread-per-Connection* strategy is an abstraction over both the Thread-per-Request and Thread-per-Session strategies discussed above. Each concurrency mechanism makes use of a *Task*. Depending on the choice of concurrency, a *Task* may represent a single *Active Object*, or a collection of *Active Objects*. The behavior of the *Concurrency* object follows the *Acceptor* pattern. This architecture enables the server developer to integrate alternate concurrency strategies. With the aid of a strategy profile, the server can dynamically choose different strategies at run-time to achieve optimal performance.

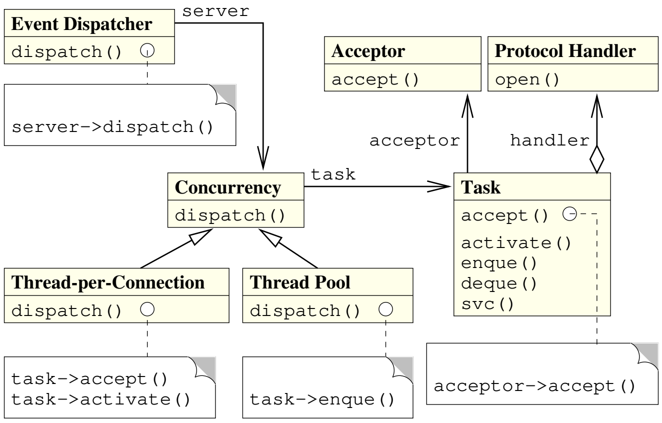

Figure 12: Structure of the Concurrency Strategy Framework

3.4 I/O Strategies

3.4.1 Design Challenges

Another key challenge for Web server developers is to devise efficient data retrieval and delivery strategies, collectively referred to as *I/O*. The issues surrounding efficient I/O can be quite challenging. Often, a system developer must resolve how to arrange multiple I/O operations to utilize concurrency available from the hardware/software platform. For instance, a high-performance Web server should simultaneously transfer files to and from the network, while concurrently parsing newly obtained incoming requests from other clients.

Certain types of I/O operations have different requirements than others. For example, a Web transaction involving monetary fund transfer may need to run *synchronously*, *i.e.*, to conclude before the user can continue. Conversely, Web accesses to static information, such as CGI-based search engine queries, may run *asynchronously* since they can be cancelled at any time. Resolving these various requirements have led to different strategies for performing I/O.

3.4.2 Alternative Solution Strategies

As indicated above, various factors influence which I/O strategy to choose. Web servers can be designed to use several different I/O strategies, including *synchronous*, *reactive*, and *asynchronous* I/O. The relative benefits of using these strategies are discussed below.

The synchronous I/O strategy: Synchronous I/O describes the model of I/O interaction between a Web server process and the kernel. In this model, the kernel does not return the thread of control to the server until the requested I/O operation either completes, completes partially, or fails. [1] shows that synchronous I/O usually performs well for small file transfers on Windows NT over high-speed ATM networks.

Synchronous I/O is well known to UNIX server programmers and is arguably the easiest to use. However, there are disadvantages to this model. First, if combined with a Single-Threaded concurrency strategy, it is not possible to perform multiple synchronous I/O operations simultaneously. Second, when using multiple threads (or processes), it is still possible for an I/O request to block indefinitely. Thus, finite resources (such as socket handles or file descriptors) may become exhausted, making the server unresponsive.

The reactive I/O strategy: Early versions of UNIX provided synchronous I/O exclusively. System V UNIX introduced non-blocking I/O to avoid the blocking problem. However, non-blocking I/O requires the Web server to poll the kernel to discover if any input is available [11]. Reactive I/O alleviates the blocking problems of synchronous I/O without resorting to polling. In this model, a Web server uses an OS event demultiplexing system call (*e.g.* `select` in UNIX or `WaitForMultipleObjects` in Win32) to determine which socket handles can perform I/O. When the call returns, the server can perform I/O on the returned handles, *i.e.*, the server *reacts* to multiple events occurring on separate handles.

Reactive I/O is widely used by event-driven applications (such as X windows) and has been codified as the *Reactor* design pattern [14]. Unless reactive I/O is carefully encapsulated, however, the technique is error-prone due to the complexity of managing multiple I/O handles. Moreover, reactive I/O may not make effective use of multiple CPUs.

The asynchronous I/O strategy: Asynchronous I/O simplifies the de-multiplexing of multiple events in one or more threads of control without blocking the Web server. When a Web server initiates an I/O operation, the kernel runs the operation asynchronously to completion while the server processes other requests. For instance, the `TransmitFile` operation in Windows NT can transfer an entire file from the server to the client asynchronously.

The advantage of asynchronous I/O is that the Web server need not block on I/O requests since they complete asynchronously. This allows the server to scale efficiently for operations with high I/O latency, such as large file transfers. The disadvantage of asynchronous I/O is that it is not available on many OS platforms (particularly UNIX). In addition, writing asynchronous programs can be more complicated than writing synchronous programs [21, 22, 28].

3.4.3 JAWS I/O Strategy Framework

Empirical studies in [1] systematically subjected different server strategies to various load conditions. The results reveal that each I/O strategy behaves differently under different load conditions. Furthermore, no single I/O strategy performed optimally under all load conditions. The JAWS I/O Strategy Framework addresses this issue by allowing the I/O strategy to adapt dynamically to run-time server conditions. Furthermore, if a new OS provides a custom I/O mechanism (such as, asynchronous scatter/gather I/O) that can potentially provide better performance, the JAWS I/O Strategy framework can easily be adapted to use it.

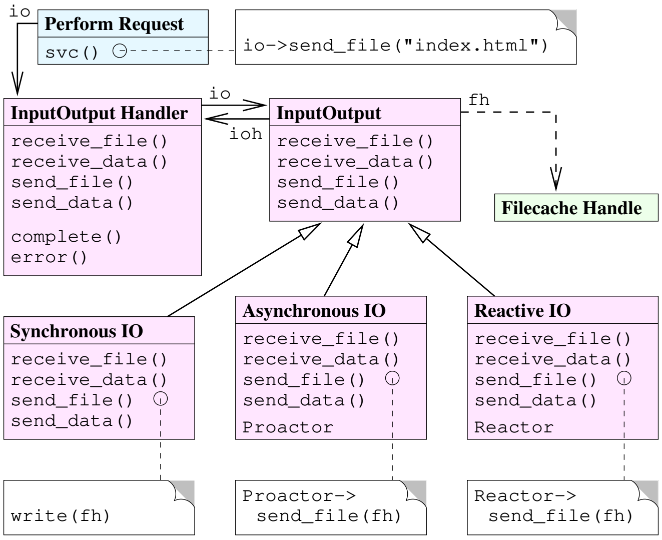

Figure 13: Structure of the I/O Strategy Framework

Figure 13 illustrates the structure of the I/O Strategy Framework provided by JAWS. *Perform Request* is a *Filter* that derives from *Protocol Pipeline*, which is explained in Section 3.5. In this example, *Perform Request* issues an I/O request to its *InputOutput Handler*. The *InputOutput Handler* delegates I/O requests made to it to the *InputOutput* object.

The JAWS framework provides *Synchronous*, *Asynchronous* and *Reactive IO* component implementations derived from *InputOutput*. Each I/O strategy issues requests using the appropriate mechanism. For instance, the *Synchronous IO* component utilizes the traditional blocking `read` and `write` system calls; the *Asynchronous IO* performs requests according to the *Proactor* pattern [21]; and *Reactive IO* utilizes the *Reactor* pattern [14].

An *InputOutput* component is created with respect to the stream associated by the *Acceptor* from the *Task* component described in Section 3.3. File operations are performed with respect to a *Filecache Handle* component, described in Section 3.6. For example, a `send_file` operation sends the file represented by a *Filecache Handle* to the stream returned by the *Acceptor*.

3.5 Protocol Pipeline Strategies

3.5.1 Design Challenges

Early Web servers, like NCSA's original `httpd`, performed very little file processing. They simply retrieved the requested file and transfered its contents to the requester. However, modern Web servers perform data processing other than file retrieval. For instance, the HTTP/1.0 protocol can be used to determine various file characteristics, such as the file type (*e.g.*, text, image, audio or video), the file encoding and compression types, the file size and its date of last modification. This information is returned to requesters via an HTTP header.

With the introduction of CGI, Web servers have been able to perform an even wider variety of tasks. These include search engines, map generation, interfacing with database systems, and secure commercial and financial transactions. However, a limitation of CGI is that the server must spawn a new *process* to extend server functionality. It is typical for each request requiring CGI to spawn its own process to handle it, causing the server to behave as a *Process-per-Request* server, which is a performance inhibitor [3]. The challenge for a high-performance Web server framework is to allow developers to extend server functionality without resorting to CGI processes.

3.5.2 Alternative Solution Strategies

Most Web servers conceptually process or transform a stream of data in several stages. For example, the stages of processing an HTTP/1.0 request can be organized as a sequence of tasks. These tasks involve (1) reading in the request, (2) parsing the request, (3) parsing the request header information, (4) performing the request, and (5) logging the request. This sequence forms a *pipeline* of tasks that manipulate an incoming request, as shown in Figure 14.

The tasks performed to process HTTP/1.0 requests have a fixed structure. Thus, Figure 14 illustrates a *static pipeline* configuration. Static configurations are useful for server extensions that are requested to perform a limited number of processing operations. If these operations are relatively few and known *a priori*, they can be pre-fabricated and used directly during the execution of the server. Examples of static processing operations include: marshalling and demarshaling data, data demultiplexing through custom Web protocol layers, and

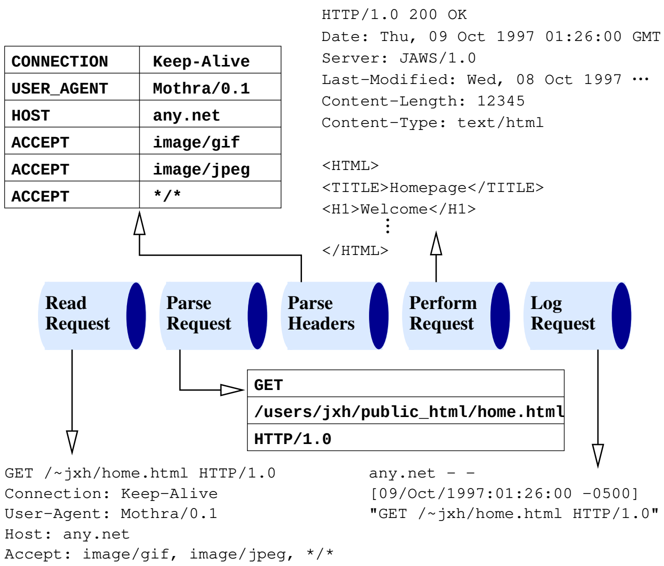

Figure 14: The Pipeline of Tasks to Perform for HTTP Requests

compiling applet code. Incorporating static pipelines into the server makes it possible to extend server functionality without resorting to spawning external processes. The Apache Web server provides this type of extensibility by allowing *modules*, which encapsulate static processing, to be dynamically linked to the Web server.

However, situations may arise that require a pipeline of operations to be configured *dynamically*. These occur when Web server extensions involve arbitrary processing pipeline configurations, so the number is essentially unlimited. This can happen if there is a large number of intermediary pipeline components that can be arranged into arbitrary sequences. If the operations on the data are known only during the execution of the program, dynamic construction of the pipeline from components provides an economical solution. There are many examples of when dynamic pipelines may be useful, including the following:

Advanced search engines: Search engines may construct data filters dynamically depending on the provided query string. For instance, a query such as "`(performance AND (NOT symphonic))`", requesting for Web pages containing the word "performance" but not the word "symphonic", could be implemented as a pipeline of a positive match component coupled with a negative match component.

Image servers: Image servers may construct filters dynamically depending on the operations requested by the user. As an example, a user may request that the image be cropped, scaled, rotated, and dithered.

Adaptive Web content: Web content can be dynamically delivered dependent on characteristics of the end-user. For example, a Personal Digital Assistant (PDA) should receive Web content overviews and smaller images, whereas a workstation could receive full multimedia enriched pages. Likewise, a home computer user may choose to block certain types of content.

Existing solutions that allow server developers to enhance Web server functionality dynamically are either too application specific, or too generic. For example, Netscape's NetDynamics focuses entirely on integrating Web servers with existing database applications. Conversely, Java-based Web servers (such as Sun's Java Server and W3C's Jigsaw) allow for arbitrary server extensibility, since Java applications are capable of dynamically executing arbitrary Java code. However, the problem of *how* to provide a dynamically configurable pipeline of operations is still left to be resolved by the server developers. Thus, developers are left to custom engineer their own solutions without the benefits of an application framework based on design patterns.

Structuring a server to process data in logical stages is known as the *Pipes and Filters* pattern [23]. While the pattern helps developers recognize how to organize the components of a processing pipeline, it does not provide a framework for doing so. Without one, the task of adapting a custom server to efficiently adopt new protocol features may be too costly, difficult, or result in high-maintenance code.

3.5.3 JAWS Protocol Pipeline Strategy Framework

The preceding discussion motivated the need for developers to extend server functionality without resorting to external processes. We also described how the *Pipes and Filters* pattern can be applied to create static and and dynamic information processing pipelines. JAWS' *Protocol Pipeline Framework* is designed to simplify the effort required to program these pipelines. This is accomplished by providing *task skeletons* of pipeline components. When the developer completes the pipeline component logic, the component can then be composed with other components to create a pipeline. The completed components are stored into a repository so that they are accessible while the server is running. This enables the server framework to dynamically create pipelines as necessary while the Web server executes.

Figure 15 provides an illustration of the structure of the framework. The figure depicts a *Protocol Handler* that utilizes the *Protocol Pipeline* to process incoming requests. A *Filter* component derives from *Protocol Pipeline*, and serves as a task skeleton for the pipeline component.

Pipeline implementors derive from *Filter* to create pipeline components. The `svc` method of each pipeline component first calls into the parent `svc`, which retrieves the data to be processed, and then performs its component specific logic upon the data. The parent invokes the `svc` method of the preceding component. Thus, the composition of the components causes a chain of calls pulling data through the pipeline. The chain ends at the component responsible for retrieving the raw input, which derives directly from *Protocol Pipeline* abstraction.

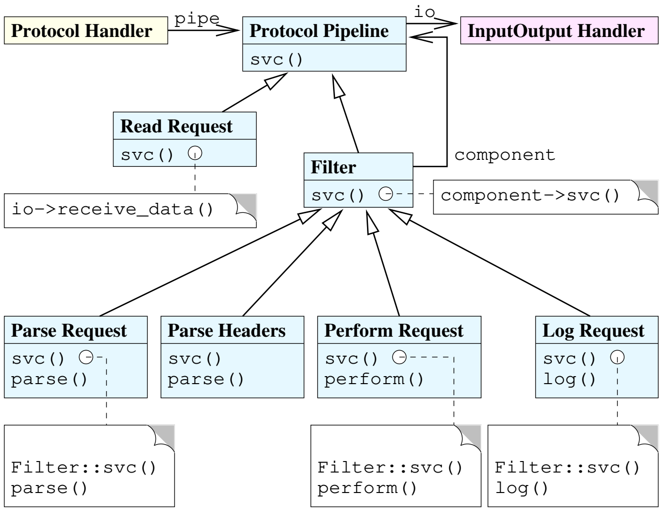

Figure 15: Structure of the Protocol Pipeline Framework

3.6 File Caching Strategies

3.6.1 Design Challenges

The results in [3] show that accessing the filesystem is a significant source of overhead for Web servers. Most distributed applications can benefit from caching, and Web servers are no exception. Therefore, it is not surprising that research on Web server performance focuses on file caching to achieve better performance [29, 30].

A cache is a storage medium that provides more efficient retrieval than the medium on which the desired information is normally located. In the case of a Web server, the cache resides in the server's main memory. Since memory is a limited resource, files only reside in the cache temporarily. Thus, the issues surrounding optimal cache performance are driven by *quantity* (how much information should be stored in the cache) and *duration* (how long information should stay in the cache).

3.6.2 Alternative Solution Strategies

Quantity and duration are strongly influenced by the size allocated to the cache. If memory is scarce, it may by undesirable to cache a few large files since caching many smaller files will give better average performance. If more memory is available, caching larger files may be feasible.

Least Recently Used (LRU) caching: This cache replacement strategy assumes most requests for cached files have *temporal* locality, *i.e.*, a requested file will soon be requested again. Thus, when the act of inserting a new file into the cached requires the removal of another file, the file that was *least recently used* is removed. This strategy is relevant to Web systems that serve content with temporal properties (such as daily news reports and stock quotes).

Least Frequently Used (LFU) caching: This cache replacement strategy assumes files that have been requested frequently are more likely to be requested again, another form of temporal locality. Thus, cache files that have been *least frequently used* are the first to be replaced in the cache. This strategy is relevant to Web systems with relatively static content, such as Lexis-Nexis and other databases of historical fact.

Hinted caching: This form of caching is proposed in [29]. This strategy stems from analysis of Web page retrieval patterns that seem to indicate that Web pages have *spatial* locality. That is, a user browsing a Web page is likely to browse the links within the page. Hinted caching is related to *pre-fetching*, though [29] suggests that the HTTP protocol be altered to allow statistical information about the links (or *hints*) to be sent back to the requester. This modification allows the client to decide which pages to pre-fetch. The same statistical information can be used to allow the server to determine which pages to *pre-cache*.

Structured caching: This refers to caches that have knowledge of the data being cached. For HTML pages, structured caching refers to storing the cached files to support hierarchical browsing of a single Web page. Thus, the cache takes advantage of the structure that is present in a Web page to determine the most relevant portions to be transmitted to the client (*e.g.*, a top level view of the page). This can potentially speed up Web access for clients with limited bandwidth and main memory, such as PDAs. Structured caching is related to the use of B-tree structures in databases, which minimize the number of disk accesses required to retrieve data for a query.

3.6.3 JAWS Cached Virtual Filesystem Framework

The solutions described above present several strategies for implementing a file cache. However, employing a fixed caching strategy does not always provide optimal performance [31]. The JAWS Cached Virtual Filesystem Framework addresses this issue in two ways. For one, it allows the cache replacement algorithms and cache strategy to be easily integrated into the framework. Furthermore, by utilizing strategy profiles, these algorithms and strategies can be dynamically selected in order to optimize performance under changing server load conditions.

Figure 16 illustrates to components in JAWS that collaborate to make the Cached Virtual Filesystem Framework. The *InputOutput* object instantiates a *Filecache Handle*, through which file interactions, such as reading and writing, are conducted. The *Filecache Handle* references a *Filecache Object*, which is managed by the *Filecache* component. The *Filecache Object* maintains information that is shared across all handles that reference it, such as the base address of the memory mapped file. The *Filecache* component is the heart of the Cached Virtual Filesystem Framework. It manages *Filecache Objects* through hashing, and follows the *State* pattern as it is choosing a *Cache Strategy* to fulfill the file request. The *Cache Strategy* component utilizes the *Strategy* pattern to allow different cache replacement algorithms, such as LRU and LFU, to be interchangeable.

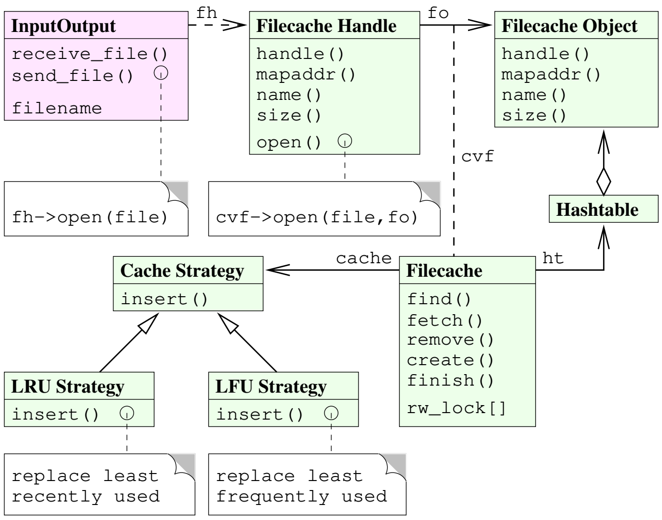

Figure 16: Structure of the Cached Virtual Filesystem

3.7 The JAWS Framework Revisited

Section 2 motivated the need for frameworks and patterns to build high-performance Web servers. We used JAWS as an example of how frameworks and patterns can enable programmers to avoid common pitfalls of developing Web server software. Together, patterns and frameworks support the reuse of integrated components and design abstractions [4].

Section 3 has described how the JAWS framework is architected and the strategies it provides. To articulate the organization of JAWS' design, we outlined the strategic and tactical design patterns that had the largest impact on the JAWS framework. The *Acceptor*, *Reactor*, *Proactor*, *Active Object*, and *Service Configurator* patterns are among the most significant strategic patterns. The *Strategy*, *Adapter*, *State*, and *Singleton* patterns are influential tactical patterns used in JAWS.

These patterns were chosen to provide the scaffolding of the JAWS architecture. The patterns described the necessary collaborating entities, which took the form of smaller components. From this, the major components of the framework, (*i.e.*, the *Concurrency Strategy*, *IO Strategy*, *Protocol Pipeline*, and the *Cached Virtual Filesystem* frameworks), were constructed and integrated into a skeleton application. Developers can use this skeleton to construct specialized Web server software systems by customizing certain sub-components of the framework (*e.g.*, *Filters* and *Cache Strategies*).

Figure 17 shows how all the patterns and components of the JAWS framework are integrated. The JAWS framework promotes the construction of high-performance Web servers in the following ways. First, it provides several pre-configured concurrency and I/O dispatching models, a file cache offering standard cache replacement strategies, and a framework for implementing protocol pipelines. Second, the patterns used in the JAWS framework help to decouple Web server strategies from (1) implementation details and (2) configuration time. This decoupling enables new strategies to be integrated easily into JAWS. Finally, JAWS is designed to allow the server to alter its behavior statically and dynamically. For instance, with an appropriate strategy profile, JAWS can accommodate different conditions that may occur during the run-time execution of a Web server. By applying these extensible strategies and components with dynamic adaptation, JAWS simplifies the development and configuration of high-performance Web servers.

4 Web Server Benchmarking Testbed and Empirical Results

Section 3 described the patterns and framework components that enable JAWS to adapt statically and dynamically to its environment. Although this flexibility is beneficial, the success of a Web server ultimately depends on how well it meets the performance demands of the Internet, as well as corporate Intranets. Satisfying these demands requires a thorough understanding of the key factors that affect Web server performance.

This section describes performance results obtained while systematically applying different load conditions on different configurations of JAWS. The results illustrate how no single configuration performs optimally for all load conditions. This demonstrates the need for flexible frameworks like JAWS that can be reconfigured to achieve optimal performance under changing load conditions.

4.1 Hardware Testbed

Our hardware testbed is shown in Figure 18. The testbed consists of two Micron Millennia PRO2 plus workstations. Each PRO2 has 128 MB of RAM and is equipped with 2 PentiumPro 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.

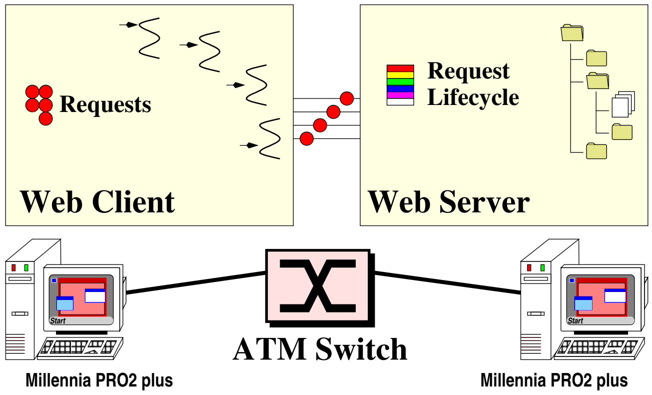

Figure 18: Benchmarking Testbed Overview

4.2 Software Request Generator

We used the WebSTONE [32] 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 in order 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

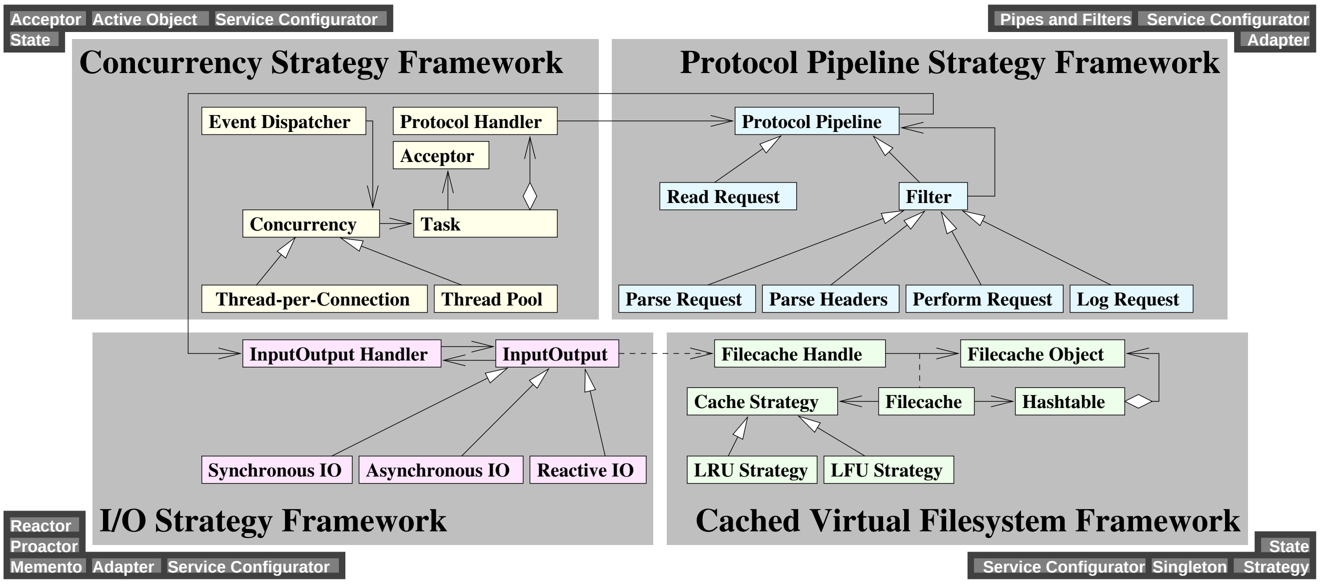

Figure 17: The JAWS Web Server Framework

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

Table 1: File Access Patterns

SPEC [33].

4.3 Experimental Results

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. For this experiment, three adaptations of JAWS were used.

1. **Synchronous Thread-per-Request:** In this adaptation, JAWS was configured to spawn a new thread to handle each incoming request, and to utilize synchronous I/O.

2. **Synchronous Thread Pool:** JAWS was configured to pre-spawn a thread pool to handle the incoming requests, while utilizing synchronous I/O.

3. **Asynchronous Thread Pool:** For this configuration, JAWS was configured to pre-spawn a thread pool to handle incoming requests, while utilizing `TransmitFile` for asynchronous I/O. `TransmitFile` is a custom Win32 function that sends file data over a network connection, either synchronously or asynchronously.

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 finishes receiving the requested file 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, in order to discover what impact file size has on performance.

4.3.1 Throughput Comparisons

Figures 19-23 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 21, 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 19-23 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 23), 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.

4.3.2 Latency Comparisons

Figures 19-23 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.

4.3.3 Summary of Benchmark Results

As illustrated in the results in this section, 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.

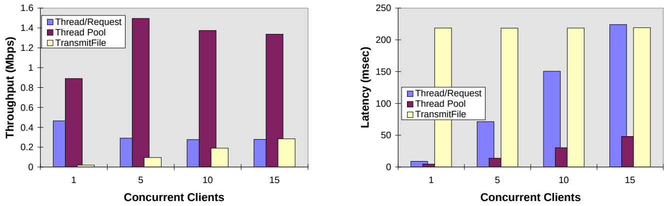

Figure 19: Experiment Results from 500 Byte File

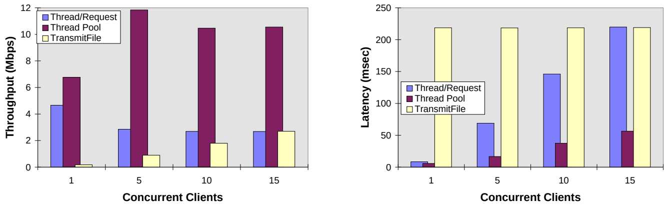

Figure 20: Experiment Results from 5K File

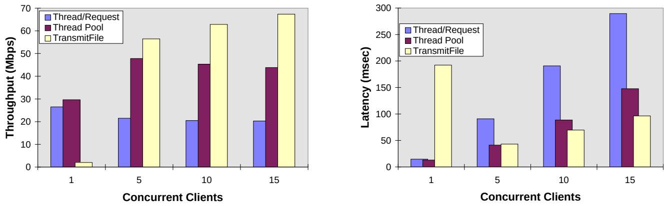

Figure 21: Experiment Results from 50K File

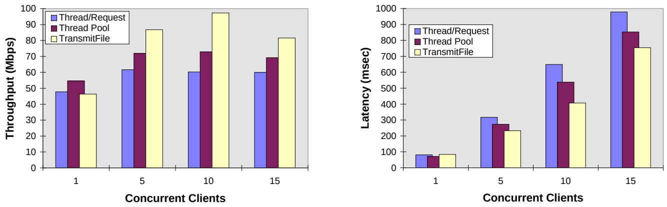

Figure 22: Experiment Results from 500K File

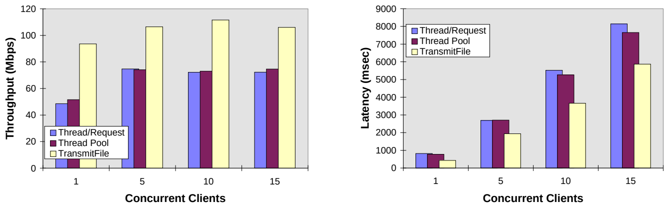

Figure 23: Experiment Results from 5M File

4.4 A Summary of Techniques for Optimizing Web Servers

From our research, we have found that it is possible to improve server performance with a superior server design (a similar observation was made in [34]). Thus, while it is undeniable that a "hard-coded" server (*i.e.*, one that uses fixed concurrency, I/O, and caching strategies) can provide excellent performance, a flexible server framework, such as JAWS, does not necessarily correlate with poor performance.

This section summarizes the most significant determinants of Web server performance. These observations are based on our studies [1, 3] of existing Web server designs and implementation strategies, as well as our experience tuning JAWS. These studies reveal the primary targets for optimizations to develop high performance Web servers.

Lightweight concurrency: Process-based concurrency mechanisms can yield poor performance, as seen in [3]. In multi-processor systems, a process-based concurrency mechanism might perform well, especially when the *number of processes are equal to the number of processors*. In this case, each processor can run a Web server process and context switching overhead is minimized.

In general, processes should be *pre-forked* to avoid the overhead of dynamic process creation. However, it is preferable to use lightweight concurrency mechanisms (*e.g.*, using POSIX threads) to minimize context switching overhead. As with processes, dynamic thread creation overhead can be avoided by *pre-spawning* threads into a pool at server start-up.

Specialized OS features: Often times, OS vendors will provide specialized programming interfaces which may give better performance. For example, Windows NT 4.0 provides the `TransmitFile` function, which uses the Windows NT virtual memory cache manager to retrieve the file data. `TransmitFile` allows data to be prepended and appended before and after the file data, respectively. This is particularly well-suited for Web servers since they typically send HTTP header data with the requested file. Hence, all the data to the client can be sent in a single system call, which minimizes mode switching overhead.

Usually, these interfaces must be benchmarked carefully against standard APIs to understand the conditions for which the special interface will give better performance. In the case of `TransmitFile`, our empirical data indicate that the asynchronous form of `TransmitFile` is the most efficient mechanism for transferring large files over sockets on Windows NT, as shown in Section 4.

Request lifecycle system call overhead: The request lifecycle in a Web server is defined as the sequence of instructions that must be executed by the server after it receives an HTTP request from the client and before it sends out the requested file. The time taken to execute the request lifecycle directly impacts the latency observed by clients. Therefore, it is important to minimize system call overhead and other processing in this path. The following describes various places in Web servers where such overhead can be reduced.

• **Reducing synchronization:** When dealing with concurrency, synchronization is often needed to serialize access to shared resources (such as the Cached Virtual Filesystem). However, the use synchronization penalizes performance. Thus, it is important to minimize the number of locks acquired (or released) during the request lifecycle. In [3], it is shown that servers that average a lower number of lock operations per request perform much better than servers that perform a high number of lock operations.

In some cases, acquiring and releasing locks can also result in *preemption*. Thus, if a thread reads in an HTTP request and then attempts to acquire a lock, it might be preempted, and may wait for a relatively long time before it is dispatched again. This increases the latency incurred by a Web client.

• **Caching files:** If the Web server does not perform file caching, at least two sources of overhead are incurred. First, there is overhead from the `open` system call. Second, there is accumulated overhead from iterative use of the `read` and `write` system calls to access the filesystem, unless the file is small enough to be retrieved or saved in a single call. Caching can be effectively performed using memory-mapped files, available in most forms of UNIX and on Windows NT.

• **Using "gather-write":** On UNIX systems, the `writev` system call allows multiple buffers to be written to a device in a single system call. This is useful for Web servers since the typical server response contains a number of header lines in addition to the requested file. By using "gather-write", header lines need not be concatenated into a single buffer before being sent, avoiding unnecessary data-copying.

• **Pre-computing HTTP responses:** Typical HTTP requests result in the server sending back the HTTP header, which contains the HTTP success code and the MIME type of the file requested, (*e.g.*, `text/plain`). Since such responses are part of the expected case they can be *pre-computed*. When a file enters the cache, the corresponding HTTP response can also be stored along with the file. When an HTTP request arrives, the header is thus directly available in the cache.

Logging overhead: Most Web servers support features that allow administrators to log the number of hits on various pages they serve. Logging is often done to estimate the load on the server at various times during the day. It is also commonly performed for commercial reasons, *e.g.*, Web sites might base their advertising rates on page hit frequencies. However, logging HTTP requests produces significant overhead for the following reasons:

• **Filesystem access:** A heavily loaded Web server makes a significant number of I/O calls, which stresses the filesystem and underlying hardware. Writing data to log files increases this stress and thus contributes to lower performance. Keeping log files and the HTTP files on separate filesystems and, if possible, on separate physical devices can limit this overhead.
• **Synchronization overhead:** A typical Web server has multiple active threads or processes serving requests. If these threads/processes are required to log requests to a common shared log file, access to this log file needs to be synchronized, *i.e.*, at most one thread/process can write to the shared log file at any time. This synchronization introduces additional overhead and is thus detrimental to performance. This overhead can be reduced by keeping multiple independent log files. If memory buffers are used, these should be stored in *thread-specific storage* to eliminate locking contention.
• **Reverse hostname lookups:** The IP address of the client is available to a Web server locally. However, the hostname is typically more useful information in the log file. Thus, the IP address of the client needs to be converted into the corresponding host name. This is typically done using *reverse DNS lookups*. Since these lookups often involve network I/O, they are very costly. Therefore, they should be avoided or completed asynchronously (using threads or asynchronous I/O).
• **Ident lookups:** The Ident protocol [35] allows a Web server to obtain the user name for a given HTTP connection. This typically involves setting up a new TCP/IP connection to the user's machine and thus involves a round-trip delay. Also, the ident lookup must be performed while the HTTP connection is active and therefore cannot be performed lazily. To achieve high performance, such lookups must thus be avoided whenever possible.

Transport layer optimizations: The following transport layer options should be configured to improve Web server performance over high-speed networks:
• **The listen backlog:** Most TCP implementations buffer incoming HTTP connections on a kernel-resident "listen queue" so that servers can dequeue them for servicing using `accept`. If the TCP listen queue exceeds the "backlog" parameter to the `listen` call, new connections are refused by TCP. Thus, if the volume of incoming connections is expected to be high, the capacity of the kernel queue should be increased by giving a higher backlog parameter (which may require modifications to the OS kernel).
• **Socket send buffers:** Associated with every socket is a send buffer, which holds data sent by the server, while it is being transmitted across the network. For high performance, it should be set to the highest permissible limit (*i.e.*, large buffers). On Solaris, this limit is 64k.
• **Nagle's algorithm (RFC 896):** Some TCP/IP implementations implement Nagle's Algorithm to avoid *congestion*. This can often result in data getting delayed by the network layer before it is actually sent over the network. Several latency-critical applications (such as X-Windows) disable this algorithm, (*e.g.*, Solaris supports the `TCP_NO_DELAY` socket option). Disabling this algorithm can improve latency by forcing the network layer to send packets out as soon as possible.

5 Concluding Remarks

As computing power and network bandwidth have increased dramatically over the past decade, so has the complexity involved in developing communication software. In particular, the design and implementation of high-performance Web server software is expensive and error-prone. Much of the cost and effort stems from the continual rediscovery and reinvention of fundamental design patterns and framework components. Moreover, the growing heterogeneity of hardware architectures and diversity of OS and network platforms make it hard to build correct, portable, and efficient Web servers from scratch.

Building a high-performance Web server requires an understanding of the performance impact of each subsystem in the server, *e.g.*, concurrency and event dispatching, server request processing, filesystem access, and data transfer. Efficiently implementing and integrating these subsystems requires developers to meet various design challenges and navigate through alternative solutions. Understanding the tradeoffs among these alternatives is essential to providing optimal performance. However, the effort required to develop *ad hoc* designs is often not cost effective as requirements (inevitably) change. As examples, the performance may prove to be inadequate, additional functionality may be required, or the software may need to be ported to a different architecture.

Object-oriented application frameworks and design patterns help to reduce the cost and improve the quality of software by leveraging proven software designs and implementations to produce reusable components that can be customized to meet new application requirements. The JAWS framework described in this paper demonstrates how the development of high-performance Web servers can be simplified and unified. The key to the success of JAWS is its ability to capture common communication software design patterns and consolidate these patterns into flexible framework components that efficiently encapsulate and enhance low-level OS mechanisms for concurrency, event demultiplexing, dynamic configuration, and file caching. The benchmarking results presented here serve to illustrate the effectiveness of using frameworks to develop high-performance applications. It brings to light that flexible software is not antithetical to performance.

The source code and documentation for JAWS are available at `www.cs.wustl.edu/~schmidt/ACE.html`.

References

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

[2] I. Pyarali, T. H. Harrison, and D. C. Schmidt, "Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging," *USENIX Computing Systems*, vol. 9, November/December 1996.

[3] 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.

[4] R. Johnson, "Frameworks = Patterns + Components," *Communications of the ACM*, vol. 40, Oct. 1997.

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

[6] D. C. Schmidt and T. Suda, "An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems," *IEE/BCS Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Systems)*, vol. 2, pp. 280–293, December 1994.

[7] D. C. Schmidt, "Applying Design Patterns and Frameworks to Develop Object-Oriented Communication Software," in *Handbook of Programming Languages* (P. Salus, ed.), MacMillan Computer Publishing, 1997.

[8] D. C. Schmidt, "A Family of Design Patterns for Application-level Gateways," *The Theory and Practice of Object Systems (Special Issue on Patterns and Pattern Languages)*, vol. 2, no. 1, 1996.

[9] T. H. Harrison, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-time CORBA Event Service," in *Proceedings of OOPSLA '97*, (Atlanta, GA), ACM, October 1997.

[10] D. C. Schmidt and C. Cleeland, "Applying Patterns to Develop Extensible and Maintainable ORB Middleware," *Communications of the ACM, to appear*, 1998.

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

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

[13] W. R. Stevens, *UNIX Network Programming, Second Edition*. Englewood Cliffs, NJ: Prentice Hall, 1997.

[14] 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.

[15] 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.

[16] 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.

[17] A. Stepanov and M. Lee, "The Standard Template Library," Tech. Rep. HPL-94-34, Hewlett-Packard Laboratories, April 1994.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

[25] P. Jain and D. C. Schmidt, "Service Configurator: A Pattern for Dynamic Configuration and Reconfiguration of Communication Services," in *The 3^{rd} Pattern Languages of Programming Conference (Washington University technical report #WUCS-97-07)*, February 1997.

[26] T. Berners-Lee, R. T. Fielding, and H. Frystyk, "Hypertext Transfer Protocol – HTTP/1.0," Informational RFC 1945, Network Working Group, May 1996. Available from http://www.w3.org/.

[27] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1," Standards Track RFC 2068, Network Working Group, January 1997. Available from http://www.w3.org/.

[28] D. C. Schmidt and C. D. Cranor, "Half-Sync/Half-Async: an Architectural Pattern for Efficient and Well-structured Concurrent I/O," in *Pattern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), Reading, MA: Addison-Wesley, 1996.

[29] J. C. Mogul, "Hinted caching in the Web," in *Proceedings of the Seventh SIGOPS European Workshop: Systems Support for Worldwide Applications*, 1996.

[30] S. Williams, M. Abrams, C. R. Standridge, G. Abdulla, and E. A. Fox, "Removal Policies in Network Caches for World Wide Web Documents," in *Proceedings of SIGCOMM '96*, (Stanford, CA), pp. 293–305, ACM, August 1996.

[31] E. P. Markatos, "Main memory caching of web documents," in *Proceedings of the Fifth International World Wide Web Conference*, May 1996.

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

[33] A. Carlton, "An Explanation of the SPECweb96 Benchmark." Standard Performance Evaluation Corporation whitepaper, 1996. Available from http://www.specbench.org/.

[34] H. F. Nielsen, J. Gettys, A. Baird-Smith, E. Prud'hommeaux, H. W. Lie, and C. Lilley, "Network Performance Effects of HTTP/1.1, CSS1, and PNG," in *To appear in Proceedings of ACM SIGCOMM '97*, 1997.

[35] M. S. Johns, "Identification Protocol," *Network Information Center RFC 1413*, Feb. 1993.