Developing Flexible and High-performance Web Servers with Frameworks and Patterns

Douglas C. Schmidt
schmidt@uci.edu
and
James C. Hu
jxh@entera.com

The goal of this paper is to illustrate how frameworks and patterns address complexities that arise in the design and implementation of high-performance distributed software systems. These complexities are both *inherent* (*e.g.*, latency reduction and throughput preservation), and *accidental* (*e.g.*, the continuous reinvention of key concepts and components). This paper explains how complexities occurring in the development of high-performance Web servers can be alleviated with the use of design patterns and object-oriented application frameworks. These techniques were applied to the development our high-performance adaptive Web server framework, JAWS. JAWS exemplifies how a framework can remain flexible without sacrificing performance.

Additional Key Words and Phrases: WWW, design patterns, distributed software systems, object-oriented application frameworks

1. APPLYING PATTERNS AND FRAMEWORKS TO WEB SERVERS

Developers of Web servers strive to build fast, scalable, and configurable systems. This paper describes some common pitfalls encountered by these developers and how to avoid these pitfalls. Common pitfalls include (1) coping with tedious and error-prone low-level programming details, (2) lack of portability, and (3) the complexity of navigating the wide range of server design alternatives. By carefully utilizing patterns and frameworks, these hazards can be avoided, by allowing developers to leverage reuse of design and code.

1.1 Common Pitfalls of Developing Web Server Software

Web servers perform the following tasks: 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,

Address: UC Irvine Department of Electrical & Computer Engineering 616E Engineering Tower Irvine, CA 92697-2625 Tel: (949) 824-1901; Fax: (949) 824-2321

these tasks are implemented in an *ad hoc* manner using low-level native OS application programming interfaces (APIs), such as Win32 or UNIX/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 [Schmidt and Cleeland 1998]. 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 servers. Such details divert attention from the broader, more strategic issues, such as protocol semantics and server structure. For example, UNIX developers who use the `wait` system call must distinguish between return errors due to no child processes being present and errors from signal interrupts. In the latter case, the `wait` must be reissued.

Reinvention of incompatible 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* development process hampers productivity and creates incompatible components that are not readily reusable within and across projects.

High potential for errors: Programming to low-level OS APIs is tedious and error-prone due to their lack of type-safety. For example, most Web servers are programmed with the Socket API [McKusick et al. 1996]. 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 POSIX asynchronous I/O [POSIX1003.1c 1995] 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 [Stevens 1997]. This number is inadequate for heavily accessed Web servers that process hundreds of simultaneous clients.

1.2 Overcoming Web Server Pitfalls with Patterns and Frameworks

Software reuse is a a widely touted method of reducing development effort. Reuse leverages the application domain knowledge and prior effort of experienced developers. When applied effectively, reuse can avoid recreating and revalidating 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 reinvent and reimplement the overall software architecture and much of the control logic 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* [Fayad and Schmidt 1997]. Patterns and frameworks help alleviate the continual rediscovery and reinvention of key Web server concepts and components by capturing solutions to common software development problems [Gamma et al. 1995].

In practice, frameworks, class libraries, and components are complementary technologies [Fayad and Schmidt 1997]. 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 [Stepanov and Lee 1994] 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.

2. THE JAWS WEB SERVER FRAMEWORK

Figure 1 illustrates the major structural components and design patterns that comprise the JAWS Adaptive Web Server (JAWS) framework. JAWS is designed to allow the customization of various Web server strategies in response to environmental factors. These factors include *static* factors (*e.g.*, number of available CPUs, support for kernel-level threads, and availability of asynchronous I/O in the OS), as well as *dynamic* factors (*e.g.*, Web traffic patterns and workload characteristics).

2.1 Components and Patterns in JAWS

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 [Schmidt 1994]. The collaborations among JAWS components and frameworks are guided by a family of patterns, which are listed along the borders in Figure 1. An outline of the key frameworks, components, and patterns in JAWS is presented below.[1]

[1]Due to space limitations it is not possible to describe each pattern in detail. The references provide additional information on each pattern mentioned below.

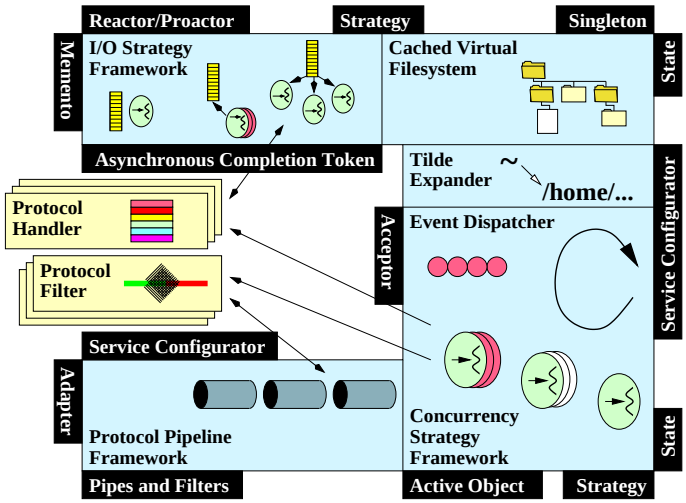

Fig. 1. Architectural Overview of the JAWS Framework

Event Dispatcher: This component is responsible for coordinating JAWS' *Concurrency Strategy* with its *I/O Strategy*. As illustrated in Figure 2, the passive

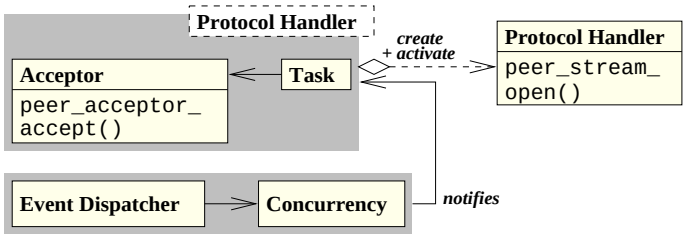

Fig. 2. Structure of the Acceptor Pattern in JAWS

establishment of connection events with Web clients follows the *Acceptor* pattern [Schmidt 1997]. New incoming HTTP request events are serviced by a concurrency strategy. As events are processed, they are dispatched to the *Protocol Handler*, which is parameterized by an I/O strategy. JAWS ability to dynamically bind to a particular concurrency strategy and I/O strategy from a range of alternatives follows the *Strategy* pattern [Gamma et al. 1995].

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 [Gamma et al. 1995] or pre-determined at initialization-time. The *Service Configurator* pattern [Jain and Schmidt 1997] 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 [Lavender and Schmidt 1996]. This is illustrated in Figure 3.

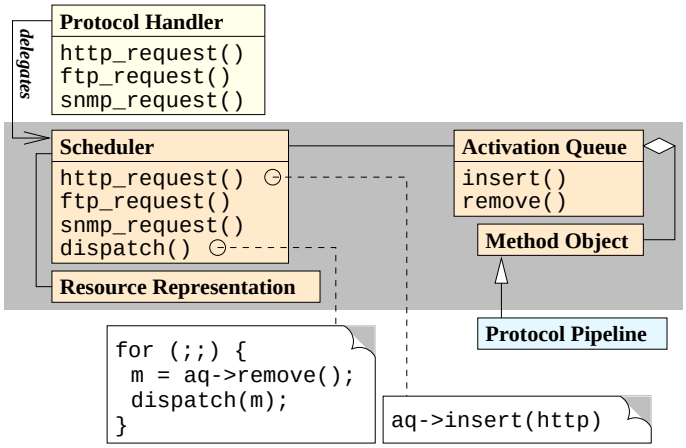

Fig. 3. Structure of the Active Object Pattern in JAWS

I/O Strategy: This framework implements various I/O mechanisms, such as asynchronous, synchronous and reactive I/O. Multiple I/O mechanisms can be used simultaneously. In JAWS, asynchronous I/O is implemented using the *Asynchronous Completion Token* [Pyarali et al. 1997] pattern and *Proactor* [Harrison et al. 1997] pattern, as illustrated in Figure 4. Reactive I/O is accomplished through the *Reactor* pattern [Schmidt 1995]. Reactive I/O utilizes the *Memento* pattern [Gamma et al. 1995] to capture and externalize the state of a request so that it can be restored at a later time.

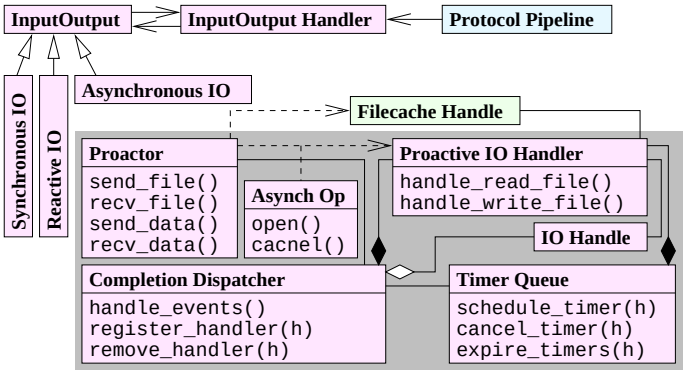

Fig. 4. Structure of the Proactor Pattern in JAWS

Protocol Handler: This framework allows system developers to apply the JAWS framework to a variety of Web system applications. A *Protocol Handler* is parameterized by a concurrency strategy and an I/O strategy. These strategies are decoupled from the protocol handler using the *Adapter* [Gamma et al. 1995] 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, DICOM, and SFP [Object Management Group 1997]) 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 [Buschmann et al. 1996] for input processing. Pipeline components can be linked dynamically at run-time using the *Service Configurator* pattern, as shown in Figure 5.
Cached Virtual Filesystem: This component improves Web server performance by reducing the overhead of filesystem access. Various caching strategies, such as LRU, LFU, Hinted, and Structured, can be selected following the *Strategy* pattern [Gamma et al. 1995]. This allows different caching strategies to be profiled and selected based on their performance. Moreover, optimal strategies to be configured statically or dynamically using the *Service Configurator* pattern, as shown in Figure 5. The cache for each Web server is instantiated using the *Singleton* pattern [Gamma et al. 1995].

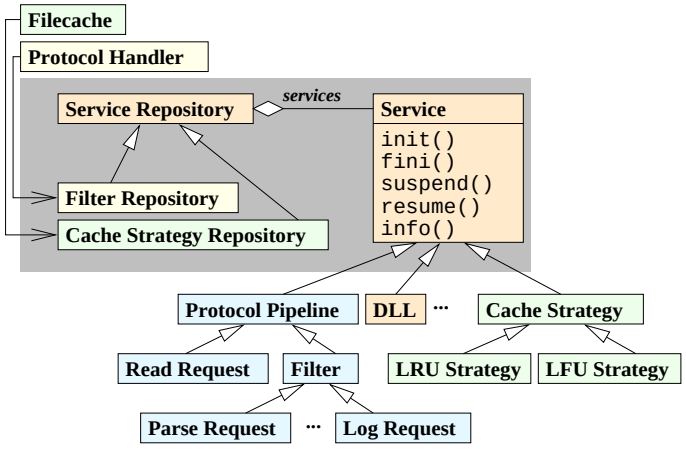

Fig. 5. The Service Configurator Pattern in JAWS

Tilde Expander: This component is another cache component that uses a perfect hash table [Schmidt 1990] 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.

2.2 JAWS Web Server Performance

Our research [Hu et al. 1997; Hu et al. 1998] demonstrates that it is possible to improve server performance through superior server design (a similar observation was made in [Nielsen et al. 1997]). Thus, while a "hard-coded" server, *i.e.*, one that uses fixed concurrency, I/O, and caching strategies, can provide excellent performance, a flexible server framewor like JAWS need necessarily not perform poorly.

Figure 6 below illustrates how the flexible nature of the JAWS framework enables it to adapt from its baseline performance to perform as well as, and in some cases better than, state of the art commercial Web servers produced by Zeus and Netscape. We achieved this level of performance through systematic benchmarking

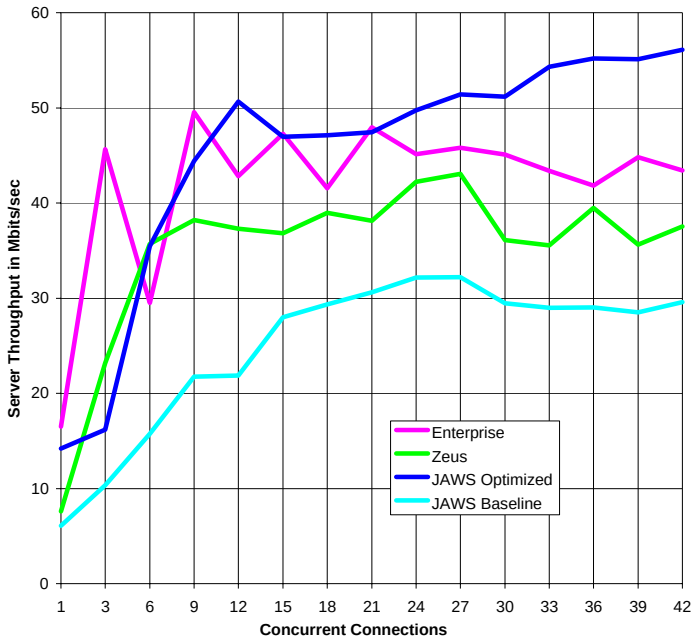

Fig. 6. Comparative Performance for JAWS

of different configurations of JAWS under different server load conditions. We then selected the combination of features that yielded the best overall performance [Hu et al. 1998].

3. CONCLUDING REMARKS

Computing power and network bandwidth has increased dramatically over the past decade. However, the development of high-performance Web servers has remained expensive and error-prone. Much of the cost and effort stems from the repeated

rediscovery and reinvention of fundamental design patterns and framework components. Moreover, the growing heterogeneity of hardware architectures and diversity of OS and network platforms makes it hard to build correct, portable, and efficient Web servers from scratch.

In general, OO application frameworks and patterns help to reduce the cost and improve the quality of communication software [Schmidt and Fayad 1997]. In the context of Web servers, these benefits accrue from leveraging proven software designs and reusable components that can be customized to meet new application requirements.

The JAWS framework described in this article exemplifies how high-performance Web server software development can be simplified and unified. One measure of success of the JAWS framework is illustrated by the fact that it outperforms other commercial and non-commercial Web servers. Commercial Web servers, such as Netscape Enterprise and Zeus, provide excellent performance, but their techniques for doing so remain behind the veils of proprietary software. Free Web servers, such as Apache and PHTTPD, provide good performance but the lack the architectural reuse that the JAWS framework provides.

The JAWS framework is freely available at `www.cs.wustl.edu/~schmidt/ACE.html`. This URL contains complete source code, documentation, and further technical information on JAWS.

REFERENCES

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

Fayad, M. E. and Schmidt, D. C. 1997. Object-Oriented Application Frameworks. *Communications of the ACM 40*, 10 (October).

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

Harrison, T., Pyarali, I., Schmidt, D. C., and Jordan, T. 1997. 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).

Hu, J., Mungee, S., and Schmidt, D. C. 1998. Principles for Developing and Measuring High-performance Web Servers over ATM. In *Proceeedings of INFOCOM '98* (March/April 1998).

Hu, J., Pyarali, I., and Schmidt, D. C. 1997. 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* (November 1997). IEEE.

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

Lavender, R. G. and Schmidt, D. C. 1996. Active Object: an Object Behavioral Pattern for Concurrent Programming. In J. O. Coplien, J. Vlissides, and N. Kerth Eds., *Pattern Languages of Program Design*. Reading, MA: Addison-Wesley.

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

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

Object Management Group. 1997. *Control and Management of Audio/Video Streams: OMG RFP Submission* (1.2 ed.). Object Management Group.

POSIX1003.1c. 1995. Information Technology – Portable Operating System Interface (POSIX) – Part 1: System Application: Program Interface (API) [C Language].

Pyarali, I., Harrison, T. H., and Schmidt, D. C. 1997. Asynchronous Completion Token: an Object Behavioral Pattern for Efficient Asynchronous Event Handling. In R. Martin, F. Buschmann, and D. Riehle Eds., *Pattern Languages of Program Design*. Reading, MA: Addison-Wesley.

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

Schmidt, D. C. 1994. ACE: an Object-Oriented Framework for Developing Distributed Applications. In *Proceedings of the* 6^{th} *USENIX C++ Technical Conference* (Cambridge, Massachusetts, April 1994). USENIX Association.

Schmidt, D. C. 1995. Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching. In J. O. Coplien and D. C. Schmidt Eds., *Pattern Languages of Program Design*, pp. 529–545. Reading, MA: Addison-Wesley.

Schmidt, D. C. 1997. Acceptor and Connector: Design Patterns for Initializing Communication Services. In R. Martin, F. Buschmann, and D. Riehle Eds., *Pattern Languages of Program Design*. Reading, MA: Addison-Wesley.

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

Schmidt, D. C. and Fayad, M. E. 1997. Lessons Learned: Building Reusable OO Frameworks for Distributed Software. *Communications of the ACM 40*, 10 (October).

Stepanov, A. and Lee, M. 1994. The Standard Template Library. Technical Report HPL-94-34 (April), Hewlett-Packard Laboratories.

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