Applying Patterns and Frameworks to Develop Object-Oriented Communication Software

Douglas C. Schmidt
schmidt@cs.wustl.edu
Department of Computer Science
Washington University, St. Louis, MO 63130

This paper appeared in the **Handbook of Programming Languages**, Volume I, edited by Peter Salus, MacMillan Computer Publishing, 1997.

1 Introduction

Communication software for next-generation distributed applications must be flexible and efficient. Flexibility is needed to support a growing range of multimedia datatypes, traffic patterns, and end-to-end quality of service (QoS) requirements. Efficiency is needed to provide low latency to delay-sensitive applications (such as avionics and call processing) and high performance to bandwidth-intensive applications (such as medical imaging and teleconferencing) over high-speed and mobile networks.

This paper outlines the key sources of complexity for communication software and describes how patterns and frameworks can alleviate much of this complexity. To focus the discussion, the paper explains how patterns and frameworks have been applied to develop high-performance, concurrent Web servers.

1.1 Sources of Complexity for Communication Software

Despite dramatic increases in computing power and network bandwidth, however, the cost of developing communication software remains high and the quality remains relatively low. Across the industry, this situation has produced a "communication software crisis," where computing hardware and networks get smaller, faster, and cheaper; yet communication software gets larger, slower, and more expensive to develop and maintain.

The challenges of communication software arise from *inherent* and *accidental* complexities [1]. Inherent complexities stem from fundamental challenges of developing communication software. Chief among these are detecting and recovering from network and host failures, minimizing the impact of communication latency, and determining an optimal partitioning of application service components and workload onto processing elements throughout a network.

The accidental complexities associated with communication software stem from limitations with conventional tools and techniques. For instance, low-level network programming interfaces like Sockets are tedious and error-prone. Likewise, higher-level distributed computing middleware like CORBA, DCOM, and Java RMI lack key features, such as asynchronous I/O and end-to-end QoS guarantees. Moreover, conventional higher-level middleware implementations are not yet optimized for applications with stringent performance requirements [2, 3].

Another source of accidental complexity arises from the widespread use of algorithmic design [4] to develop communication software. Although graphical user-interfaces (GUIs) are largely built using object-oriented (OO) design, communication software has traditionally been developed with algorithmic design. However, algorithmic design yields non-extensible software architectures that cannot be customized rapidly to meet changing application requirements. In an era of deregulation and stiff global competition, it is prohibitively expensive and time consuming to repeatedly develop applications from scratch using algorithmic design techniques.

1.2 Alleviating the Complexity of Communication Software with OO Frameworks and Patterns

OO techniques provide principles, methods, and tools that significantly reduce the complexity and cost of developing communication software [5]. The primary benefits of OO stem from its emphasis on modularity, reusability, and extensibility. Modularity encapsulates volatile implementation details behind stable interfaces. Reusability and extensibility enhance software by factoring out common object structures and functionality. This paper illustrates how to produce flexible and efficient communication software using OO *application frameworks* and *design patterns*.

A framework is a reusable, "semi-complete" application that can be specialized to produce custom applications [6]. A pattern represents a recurring solution to a software development problem within a particular context [7]. Patterns and frameworks can be applied together synergistically to improve the quality of communication software by capturing successful software development strategies. Patterns capture abstract designs and software architectures in a systematic format that can be readily comprehended by developers. Frameworks capture concrete designs, algorithms, and implementations in particular programming languages.

The examples in the paper focus on developing high-performance concurrent Web servers using the ACE framework [8]. ACE is an OO framework that provides components that implement core concurrency and distribution patterns [9] related to the domain of communication software. The framework and patterns in this paper are representative of solutions that have been successfully applied to communication systems ranging from telecommunication system management [9] to enterprise medical imaging [10] and real-time avionics [11].

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 ACE; Section 3 outlines the structure of the ACE framework; Section 4 illustrates how patterns and components in ACE can be applied to develop high-performance Web servers; and Section 5 presents concluding remarks.

2 Applying Patterns and Frameworks to Communication Software

2.1 Common Pitfalls of Developing Communication Software

2.1.1 Limitations of Low-level Native OS APIs

Developers of communication software confront recurring challenges that are largely independent of specific application requirements. For instance, applications like network file systems, email gateways, object request brokers, and Web servers all perform tasks related to connection establishment, service initialization, event demultiplexing, event handler dispatching, interprocess communication, shared memory management, static and dynamic component configuration, concurrency, synchronization, and persistence. Traditionally, these tasks have been 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 complex communication middleware and applications [12]. The following are common pitfalls associated with the use of native OS APIs:

Excessive low-level details: Developers must have intimate knowledge of low-level OS details. For instance, developers must carefully track which error codes are returned by each system call and handle these OS-specific problems in their applications. These details divert attention from the broader, more strategic application-related semantics and program structure.

Continuous re-discovery and re-invention 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, a Reactor [13] is a useful component for demultiplexing I/O events and dispatching their associated event handlers. However, these abstractions are often re-discovered and re-invented in an *ad hoc* manner by each developer or project. This process hampers productivity and creates incompatible components that cannot be reused readily 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, many networking applications are programmed with the Socket API [14]. 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 [15].

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 on different versions of Windows NT possesses incompatible bugs that cause sporadic failures when performing non-blocking connections and when shutting down processes.

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 the thread cancellation mechanism correctly in POSIX Pthreads. It is even harder to learn how to write a *portable* application using thread cancellation 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 Windows NT process only allows 64 thread local storage (TLS) keys. This number is inadequate for large-scale server applications that utilize many DLLs and thread local objects.

2.1.2 Limitations of Higher-level Distributed Object Computing Middleware

It is possible to alleviate some of the pitfalls with native OS APIs by using higher-level distributed computing middleware. Common examples of higher-level distributed computing middleware include CORBA [16], DCOM [17], and Java RMI [18]. Higher-level distributed computing middleware resides between clients and servers and eliminates many tedious, error-prone, and non-portable aspects of developing and maintaining distributed applications by automating common network programming tasks such as object location, object activation, parameter marshaling, fault recovery, and security.

However, higher-level distributed computing middleware is often only a partial solution, for the following reasons:

Lack of portability: Conventional higher-level middleware is not widely portable. For instance, the Object Adapter component in the CORBA 2.0 specification is woefully underspecified [19]. Therefore, servers written in CORBA are not portable among ORB products from different vendors. Likewise, DCOM is targeted for Win32 platforms and Java RMI is targeted for applications written in Java.

Lack of features: Conventional higher-level middleware focuses primarily on communication. Therefore, it does not cover other key issues associated with developing distributed applications. For instance, conventional higher-level middleware does not specify important aspects of high-performance and real-time distributed server development such as shared memory, asynchronous I/O, multi-threading, and synchronization [20].

Lack of performance: Conventional higher-level middleware incurs significant throughput and latency overhead [2, 21]. These overheads stem from excessive data copying, non-optimized presentation layer conversions, internal message buffering strategies that produce non-uniform behavior for different message sizes, inefficient demultiplexing algorithms, long chains of intra-ORB virtual method calls, and lack of integration with underlying real-time OS and network QoS mechanisms [22].

2.2 Overcoming Communication Software Pitfalls with Patterns and Frameworks

Successful developers and software organizations overcome the pitfalls described above by identifying the *patterns* that underly proven solutions and by reifying these patterns in *object-oriented application frameworks*. Together, patterns and frameworks help alleviate the continual re-discovery and re-invention of communication software concepts and components by capturing solutions to standard communication software development problems [7].

2.2.1 The Benefits of Patterns

Patterns are particularly useful for documenting the structure and participants in common micro-architectures for concurrency and communication such as Reactors [13], Active Objects [23], and Brokers [24]. These patterns are generalizations of object-structures that have proven useful to build flexible and efficient event-driven and concurrent communication software frameworks and applications.

Traditionally, communication software 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 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, explicitly capturing and documenting communication software patterns 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 applications.

2.2.2 The Benefits of Frameworks

Although knowledge of patterns helps to reduce development effort and maintenance costs, reuse of patterns alone is not sufficient to create flexible and efficient communication software. While patterns enable reuse of abstract design and architecture knowledge, abstractions documented as patterns do not directly yield reusable code. 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 communication software components by implementing common design patterns and factoring out common implementation roles.

2.2.3 Relationship Between Frameworks 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 [25]. The resulting application skeletons can be customized by inheriting and instantiating from reuseable components in the frameworks.

The scope of reuse in a framework can be significantly larger than using traditional function libraries or conventional OO class libraries. The increased level of reuse stem from the fact that frameworks are tightly integrated with key communication software tasks such as service initialization, error handling, flow control, event processing, and concurrency control.

In general, frameworks enhance class libraries 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 (A) are typically low-level, relatively independent, and general components like Strings, complex numbers, arrays, and bitsets. 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 1 (B), this reduces the amount of application-specific code since much of the domain-specific processing is factored into the generic components in the framework.

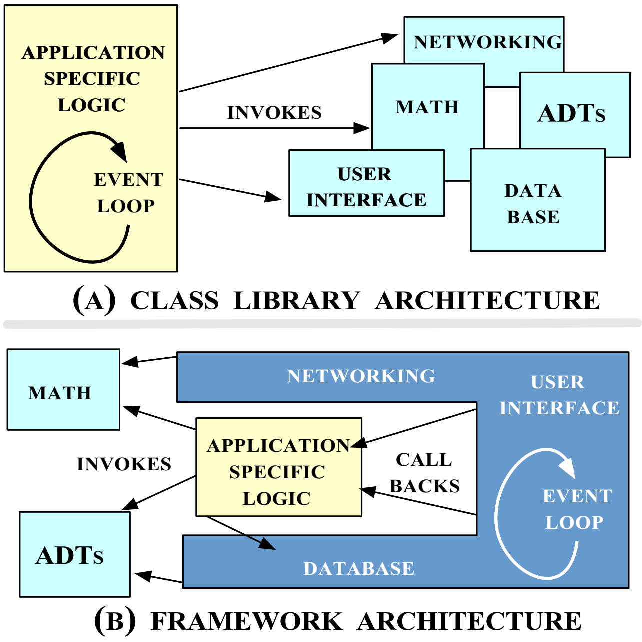

Figure 1: Differences Between Class Libraries and OO Frameworks

Frameworks are active and exhibit "inversion of control" at run-time: Class libraries are typically *passive*, *i.e.*, they perform their processing by borrowing threads of control from self-directed application objects. This is illustrated in Figure 1 (A), where the application-specific logic manages the event loop. In contrast, frameworks are *active*, *i.e.*, they manage the flow of control within an application via event dispatching patterns like Reactor [13] and Observer [7]. The callback-driven run-time architecture of a framework is shown in Figure 1 (B). This "inversion of control" is referred to as *The Hollywood Principle* [26], *i.e.*, "don't call us, we'll call you."

In practice, frameworks and class libraries are complementary technologies. Frameworks often utilize class libraries internally to simplify the development of the framework. For instance, portions of ACE use the string and vector containers provided by the C++ Standard Template Library [27] 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 are being successfully applied to communication software, the remainder of this paper examines the structure and use of the ACE framework [8].

3 Overview of ACE

ACE is an object-oriented (OO) framework that implements core concurrency and distribution patterns [9] for communication software. ACE provides a rich set of reusable C++ wrappers and framework components that are targeted for developers of high-performance, real-time services and applications across a wide range of OS platforms, including Win32, most versions of UNIX, and many real-time operating systems. The components in ACE provide reusable implementations of the following common communication software tasks:

- *Connection establishment and service initialization* [28];
- *Event demultiplexing and event handler dispatching* [13, 29, 30];
- *Interprocess communication* [15] and *shared memory management*;
- *Static and dynamic configuration [8, 31] of communication services*;
- *Concurrency and synchronization* [29, 23];
- *Distributed communication services* – such as naming, event routing [9], logging, time synchronization, and network locking;
- *Higher-level distributed computing middleware components* – such as Object Request Brokers (ORBs) [20], Web servers [32], and electronic medical imaging systems [33].

This section outlines the structure and functionality of the ACE framework. Section 4 illustrates how components and patterns in ACE can be applied to build high-performance, concurrent Web servers.

3.1 The Structure and Functionality of ACE

The ACE framework contains ∼150,000 lines of C++ code divided into ∼450 classes. To separate concerns and to reduce the complexity of the framework, ACE is designed using a layered architecture. Figure 2 illustrates the relationships between the key components in ACE.

The lower layers of ACE contain an *OS adapter* and *C++ wrappers* that portably encapsulate core OS communication and concurrency services. The higher layers of ACE extend the C++ wrappers to provide reusable *frameworks*, *self-contained distributed service components*, and *higher-level distributed computing middleware components*. Together, these layers and components simplify the creation, composition, and configuration of communication systems, *without* incurring significant performance overhead. The role of each layer is outlined below.

3.1.1 The ACE OS Adaptation Layer

The *OS adaptation layer* constitutes approximately 13% of ACE, *i.e.*, ∼20,000 lines of code. This layer resides directly atop the native OS APIs written in C. The OS adaptation layer shields the other layers and components in ACE from platform-specific dependencies associated with the following OS APIs:

Concurrency and synchronization: ACE's adaptation layer encapsulates OS APIs for multi-threading, multi-processing, and synchronization.

Interprocess communication (IPC) and shared memory: ACE's adaptation layer encapsulates OS APIs for local and remote IPC and shared memory.

Event demultiplexing mechanisms: ACE's adaptation layer encapsulates OS APIs for synchronous and asynchronous demultiplexing I/O-based, timer-based, signal-based, and synchronization-based events.

Explicit dynamic linking: ACE's adaptation layer encapsulates OS APIs for explicit dynamic linking, which allows application services to be configured at installation-time or run-time.

File system mechanisms: ACE's adaptation layer encapsulates OS file system APIs for manipulating files and directories.

The portability of ACE's OS adaptation layer enables it to run on a wide range of operating systems. The OS platforms supported by ACE include Win32 (WinNT 3.5.x, 4.x, Win95, and WinCE using MSVC++ and Borland C++), most versions of UNIX (SunOS 4.x and 5.x; SGI IRIX 5.x and 6.x; HP-UX 9.x, 10.x, and 11.x; DEC UNIX 3.x and 4.x, AIX 3.x and 4.x, DG/UX, Linux, SCO, UnixWare, NetBSD, and FreeBSD), real-time operating systems (VxWorks, Chorus, LynxOS, and pSoS), and MVS OpenEdition.

Because of the abstraction provided by ACE's OS adaptation layer, a single source tree is used for all these platforms. This design greatly simplies the portability and maintainability of ACE.

3.1.2 The ACE C++ Wrapper Layer

It is possible to program highly portable C++ applications directly atop ACE's OS adaptation layer. However, most ACE developers use the *C++ wrappers* layer shown in Figure 2. The ACE C++ wrappers simplify application development by providing typesafe C++ interfaces that encapsulate and enhance the native OS concurrency, communication, memory management, event demultiplexing, dynamic linking, and file system APIs.

The C++ wrappers provided by ACE are quite comprehensive, constituting ∼50% of its source code. Applications can combine and compose these wrappers by selectively inheriting, aggregating, and/or instantiating the following components:

Concurrency and synchronization components: ACE abstracts native OS multi-threading and multi-processing mechanisms like mutexes and semaphores to create higher-level OO concurrency abstractions like Active Objects [23] and Polymorphic Futures [34].

IPC and filesystem components: The ACE C++ wrappers encapsulate local and/or remote IPC mechanisms [15] such as sockets, TLI, UNIX FIFOs and STREAM pipes, and Win32 Named Pipes. In addition, the ACE C++ wrappers encapsulate the OS filesystem APIs.

Memory management components: The ACE memory management components provide a flexible and extensible abstraction for managing dynamic allocation and deallocation of interprocess shared memory and intraprocess heap memory.

The C++ wrappers provide many of the same features as the OS adaptation layer in ACE. However, these features are structured in terms of C++ classes and objects, rather than stand-alone C functions. This OO packaging helps to reduce the effort required to learn and use ACE correctly [35].

For instance, the use of C++ improves application robustness because the C++ wrappers are strongly typed. Therefore, compilers can detect type system violations at compile-time rather than at run-time. In contrast, it is not possible to detect

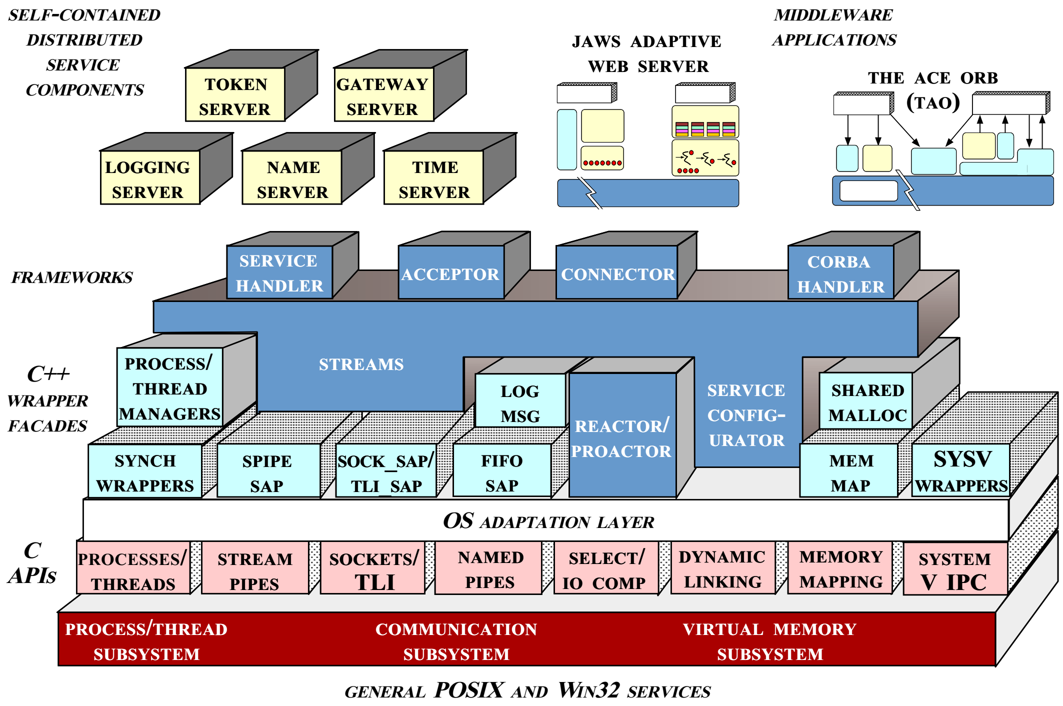

Figure 2: The Layering Structure of Components in the ACE Framework

typesystem violations for C-level OS APIs, such as sockets or filesystem I/O, until run-time.

ACE employs a number of techniques to minimize or eliminate performance overhead. For instance, ACE uses C++ inlining extensively to eliminate method call overhead that would otherwise be incurred from the additional typesafety and levels of abstraction provided by its OS adaptation layer and the C++ wrappers In addition, ACE avoids the use of virtual methods for performance-critical wrappers, such as `send`/`recv` methods for socket and file I/O.

3.1.3 The ACE Framework Components

The remaining ~40% of ACE consists of communication software framework components that integrate and enhance the C++ wrappers. These framework components support the flexible configuration of concurrent communication applications and services [8]. The framework layer in ACE contains the following components:

Event demultiplexing components: The ACE Reactor [13] and Proactor [30] are extensible, object-oriented demultiplexers that dispatch application-specific handlers in response to various types of I/O-based, timer-based, signal-based, and synchronization-based events.

Service initialization components: The ACE Connector and Acceptor components [28] decouple the active and passive initialization roles, respectively, from application-specific tasks that communication services perform once initialization is complete.

Service configuration components: The ACE Service Configurator [31] supports the configuration of applications whose services may be assembled dynamically at installation-time and/or run-time.

Hierarchically-layered stream components: The ACE Streams components [8, 36] simplify the development of communication software applications, such as user-level protocol stacks, that are composed of hierarchically-layered services.

ORB adapter components: ACE can be integrated seamlessly with single-threaded and multi-threaded CORBA implementations via its ORB adapters [10].

The ACE framework components facilitate the development of communication software that can be updated and extended without the need to modify, recompile, relink, or often restart running applications [8]. This flexibility is achieved in ACE by combining (1) C++ language features, such as templates, inheritance, and dynamic binding, (2) design patterns, such as Abstract Factory, Strategy, and Service Configurator [7, 31], and (3) OS mechanisms, such as dynamic linking and multi-threading.

3.1.4 Self-contained Distributed Service Components

In addition to its C++ wrappers and framework components, ACE provides a standard library of distributed services that are packaged as self-contained components. Although these service components are not strictly part of the ACE framework library, they play two important roles:

Factoring out reusable distributed application building blocks: These service components provide reusable implementations of common distributed application tasks such as naming, event routing [9], logging, time synchronization, and network locking.

Demonstrating common use-cases of ACE components: The distributed services also demonstrate how ACE components like Reactors, Service Configurators, Acceptors and Connectors, Active Objects, and IPC wrappers can be used effectively to develop flexible, efficient, and reliable communication software.

3.1.5 Higher-level Distributed Computing Middleware Components

Developing robust, extensible, and efficient communication applications is challenging, even when using a communication framework like ACE. In particular, developers must still master a number of complex OS and communication concepts such as:

- Network addressing and service identification.
- Presentation conversions, such as encryption, compression, and network byte-ordering conversions between heterogeneous end-systems with alternative processor byte-orderings.
- Process and thread creation and synchronization.
- System call and library routine interfaces to local and remote interprocess communication (IPC) mechanisms.

It is possible to alleviate some of the complexity of developing communication applications by employing higher-level distributed computing middleware, such as CORBA [16], DCOM [17], or Java RMI [18]. Higher-level distributed computing middleware resides between clients and servers and automates many tedious and error-prone aspects of distributed application development, including:

- Authentication, authorization, and data security.
- Service location and binding.
- Service registration and activation.
- Demultiplexing and dispatching in response to events.

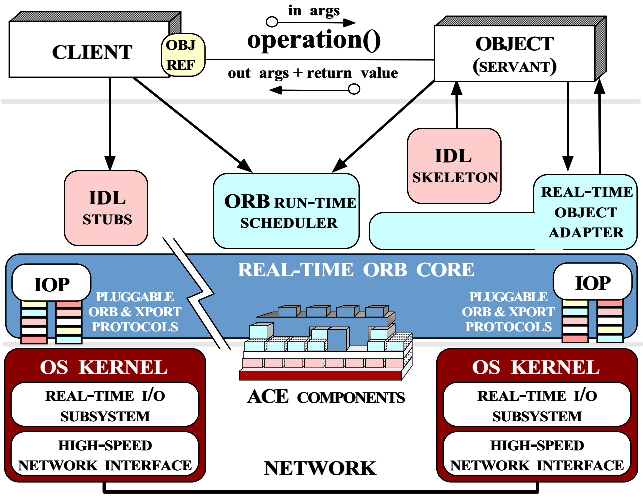

Figure 3: Components in the TAO Real-time ORB

- Implementing message framing atop bytestream-oriented communication protocols like TCP.
- Presentation conversion issues involving network byte-ordering and parameter marshaling.

To provide developers of communication software with these features, the following higher-level middleware applications are bundled with the ACE release:

The ACE ORB (TAO): TAO [22] is a real-time implementation of CORBA built using the framework components and patterns provided by ACE. TAO contains the network interface, OS, communication protocol, and CORBA middleware components and features shown in Figure 3. TAO is based on the standard OMG CORBA reference model [16], with the enhancements designed to overcome the shortcomings of conventional ORBs [3] for high-performance and real-time applications. TAO, like ACE, is freely available at `www.cs.wustl.edu/~schmidt/TAO.html`.

JAWS: JAWS [37] is a high-performance, adaptive Web server built using the framework components and patterns provided by ACE. Figure 4 illustrates the major structural components and design 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 by combining and extending components in ACE. JAW is also freely available at `www.cs.wustl.edu/~jxh/research/`. The examples in Section 4 are based on the design of JAWS.

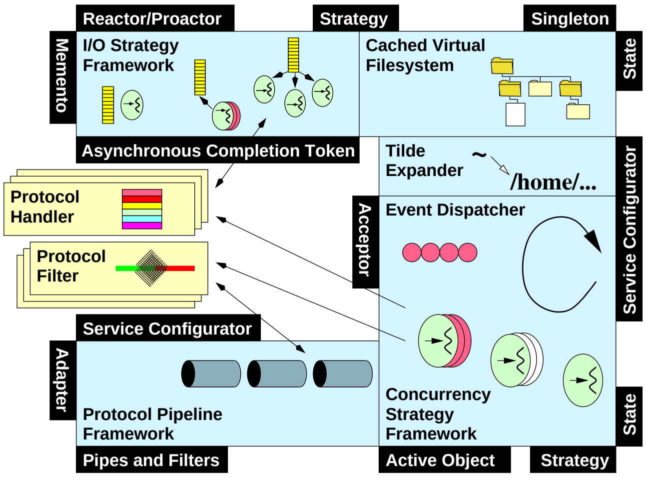

Figure 4: Architectural Overview of the JAWS Framework

4 Developing High-performance Web Servers with Patterns and Framework Components

The benefits of applying frameworks and patterns to communication software is best introduced by example. This section describes the structure and functionality high-performance Web servers developed using the patterns and framework components in ACE. Many error handling details are omitted to keep the code examples concise. In addition, the examples focus on features that are portable across multiple OS platforms, though noteworthy platform-specific features of ACE (such as asynchronous I/O and I/O completion ports) are described where appropriate.

4.1 Overview of a Web System

Figure 5 illustrates the general architecture of a Web system. The diagram provides a layered view of the architectural components required for an *HTTP client* to retrieve an HTML file from an *HTTP server*. Through *GUI* interactions, the client application user instructs the HTTP client to retrieve a file. The *requester* is the active component of the client that communicates over the *network*. It issues a request for the file to the server with the appropriate syntax of the *transfer protocol*, in this case HTTP. Incoming requests to the *HTTP server* are received by the *dispatcher*, which is the request demultiplexing engine of the server. It is responsible for creating new threads or processes (for concurrent Web servers) or managing sets of socket handles (for single-threaded concurrent servers). Each request is processed by a *handler*, which goes through a

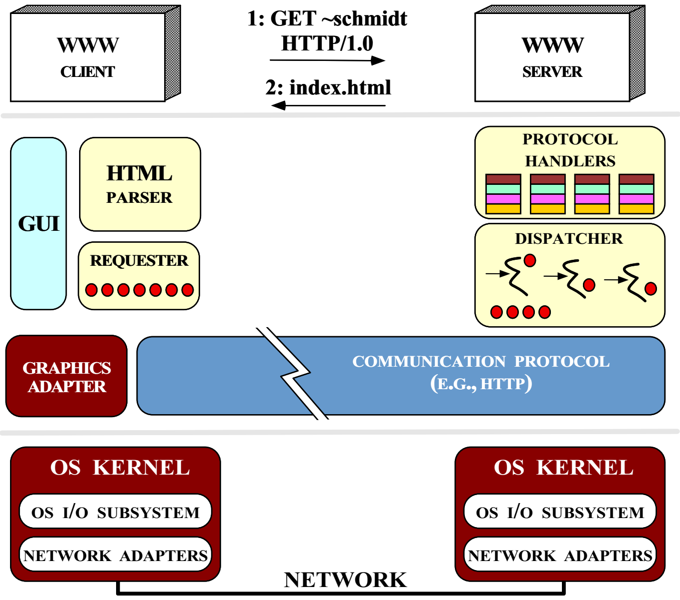

Figure 5: General Architecture of a Web System

lifecycle of parsing the request, logging the request, fetching file status information, updating the cache, sending the file, and cleaning up after the request is done. When the response returns to the client with the requested file, it is parsed by an *HTML parser* so that the file may be rendered. At this stage, the *requester* may issue other requests on behalf of the client, *e.g.*, in order to maintain a client-side cache.

4.2 Overview of an OO Web Server Communication Software Architecture

Figure 6 illustrates the general object-oriented communication software architecture for the Web servers covered in this section. The roles performed by the components in this architec-

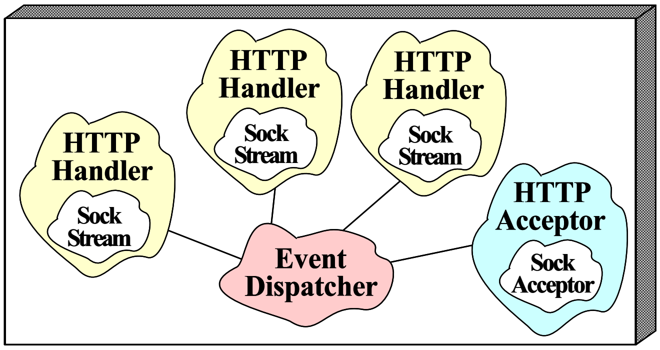

Figure 6: Object-oriented Communication Software Architecture for a Web Server

ture include the following:

The Event Dispatcher: This component encapsulates the Web server's concurrency strategies (such as thread-per-request or thread pool) and request dispatching strategies (such as synchronous Reactive or asynchronous Proactive dispatching). The ACE framework allows these strategies to be customized according to platform characteristics such as user-level vs. kernel-level threading in the OS, the number of CPUs in the endsystem, and the existence of special-purpose OS support for HTTP processing (*e.g.*, the Windows NT `TransmitFile` system call [32]).

An HTTP Handler: This component is created for each client HTTP connection (*e.g.*, from a Web browser). It parses HTTP requests and performs the work specified by the requests (*e.g.*, retrieving Web pages). An `HTTP Handler` contains an ACE `SOCK Stream`, which encapsulates the data transmission capabilities of TCP/IP sockets.

The HTTP Acceptor: This component is a factory that accepts connections from clients and creates `HTTP Handlers` to process the requests from clients. There is typically one `HTTP Acceptor` per server, though certain concurrency strategies (such as thread pool) allocate multiple Acceptors to leverage OS multi-threading capabilities. An `HTTP Acceptor` contains an ACE `SOCK Acceptor`, which encapsulates the passive connection establishment capabilities of TCP/IP sockets.

The `SOCK Acceptor` and `SOCK Stream` are C++ wrappers provided by ACE. They shield applications from non-portable, tedious, and error-prone aspects of developing communication software using the native OS socket interfaces written in C. Other ACE components will be introduced throughout this section, as well.

4.3 Design Patterns for Web Server Communication Software

The communication software architecture diagram in Figure 6 explains *how* the Web server is structured, but not *why* it is structured this particular way. To understand why the Web server contains roles such as Event Dispatcher, Acceptor, and Handler requires a deeper understanding of the design patterns underlying the domain of communication software, in general, and Web servers, in particular. Figure 7 illustrates the *strategic* and *tactical* patterns related to Web servers. These patterns are summarized below.

4.3.1 Strategic Patterns in JAWS

The following patterns are strategic because they are ubiquitous to the domain of communication software. Therefore, they significantly influence the software architecture of Web servers.

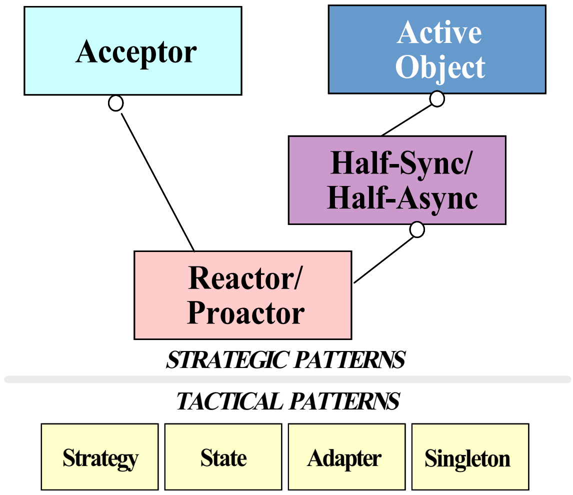

Figure 7: Common Design Patterns in Web Servers

The Acceptor pattern: This pattern [28] decouples passive connection establishment from the service performed once the connection is established. Figure 8 illustrates the structure of the Acceptor pattern in the context of Web servers. The `HTTP Acceptor` is a factory that creates, accepts, and activates a new `HTTP Handler` whenever the `Event Dispatcher` notifies it that a connection has arrived from a client. All the Web server implementations described below use the Acceptor pattern to decouple connection establishment from HTTP protocol processing.

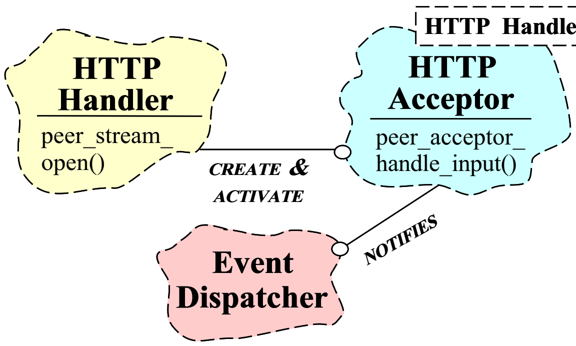

Figure 8: Structure of the Acceptor Pattern

The Reactor pattern: This pattern [13] decouples the synchronous event demultiplexing and event handler notification dispatching logic of server applications from the service(s) performed in response to events. Figure 9 illustrates the structure of the Reactor pattern in the context of Web servers. Both the `HTTP Acceptor` and `HTTP Handler` inherit from the abstract `Event Handler` interface and register themselves with the `Initiation Dispatcher` for input events (*i.e.*, connections and HTTP requests), respectively. The `Initiation Dispatcher` invokes the `handle_input` notification hook method of these `Event Handler` subclass objects when their associated events occur. The Reactor pattern is used by most of the Web server concurrency models presented in Section 4.4.

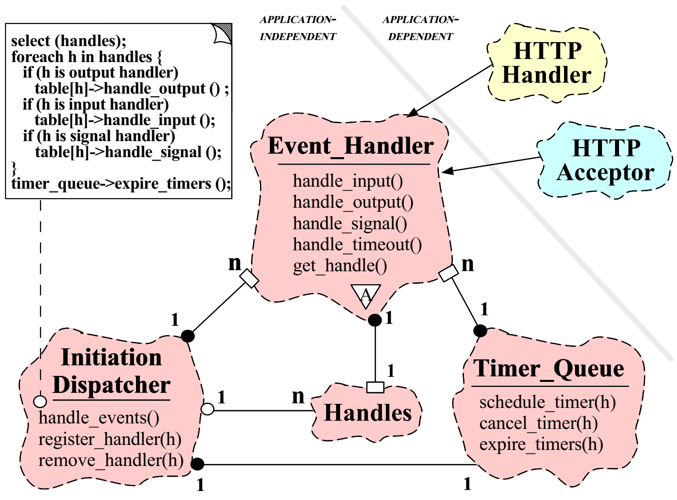

Figure 9: Structure of the Reactor Pattern

The Proactor pattern: This pattern [30] decouples the asynchronous event demultiplexing and event handler completion dispatching logic of server applications from the service(s) performed in response to events. Figure 10 illustrates the structure of the Proactor pattern in the context of Web servers. As before, both the `HTTP Acceptor` and `HTTP Handler` inherit from the abstract `Event Handler` interface. The difference is that this `Event Handler` defines *completion* hooks rather than *initiation* hooks. Therefore, when asynchronous invoked `accept` and `read` operations complete, the `Completion Dispatcher` invokes the appropriate completion hook method of these `Event Handler` subclass objects. The Proactor pattern is used in the asynchronous variant of the thread pool in Section 4.4.3.

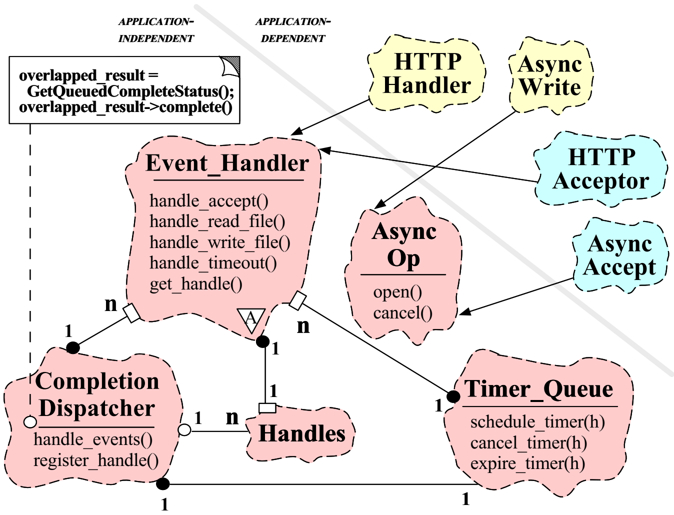

Figure 10: Structure of the Proactor Pattern

The Active Object pattern: This pattern [23] decouples method invocation from method execution, allowing methods to run concurrently. Figure 11 illustrates the structure of the Active Object pattern in the context of concurrent Web servers. The `Client Interface` transforms method requests (such as `get_request`) into `Method Objects` that are stored on an `Activation Queue`. The `Scheduler`, which runs in a separate thread from the client,

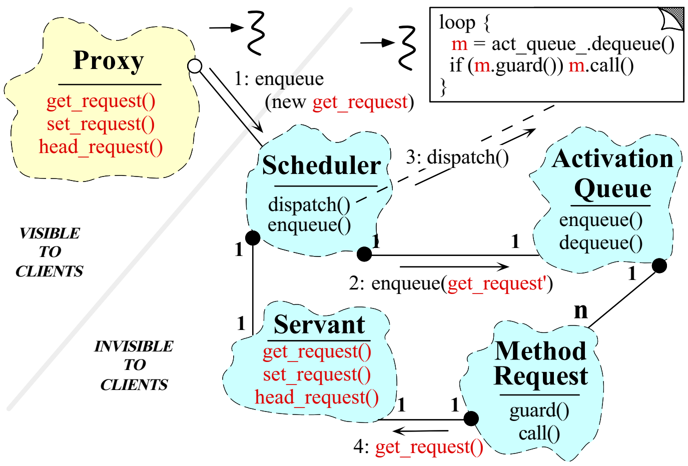

Figure 11: Structure of the Active Object Pattern

dequeues these `Method Objects` and transforms them back into method calls to perform the specified HTTP processing. The Active Object pattern is used in the thread-per-request model in Section 4.4.2, the thread pool models in Section 4.4.3, and the thread-per-connection model in Section 4.4.4,

The Half-Sync/Half-Async pattern: This pattern [29] decouples synchronous I/O from asynchronous I/O in a system in order to simplify concurrent programming effort without degrading execution efficiency. Figure 12 illustrates the structure of the Half-Sync/Half-Async pattern in the context the user-level queue-based thread pool model in Section 4.4.3. In

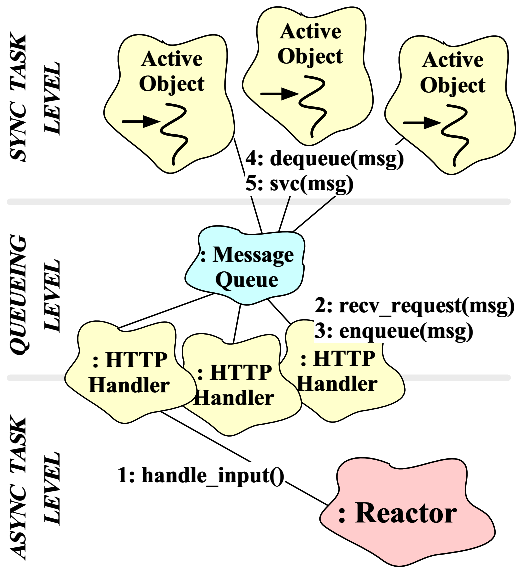

Figure 12: Structure of the Half-Sync/Half-Async Pattern

this design, the `Reactor` is responsible for reading HTTP requests from clients and and enqueueing valid requests on a `Message Queue`. This `Message Queue` feeds the pool of `Active Objects` that process the requests concurrently.

4.3.2 Tactical Patterns in JAWS

Web servers also utilize many *tactical* patterns. For instance, the following patterns are widely used in ACE and JAWS:

The Strategy pattern: This pattern defines a family of algorithms, encapsulates each one, and make them interchangeable. JAWS uses this pattern extensively to configure different concurrency and event dispatching 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. ACE uses this pattern in its OS adaptation layer to encapsulate the accidental complexity of the myriad native OS APIs in a uniform manner.

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

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

In contrast to the strategic patterns describe earlier, tactical patterns are domain-independent and have a relatively localized impact on a software design. For instance, Singleton is a tactical pattern that is often used to consolidate all option processing used to configure 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 environments.

4.4 Implementing Web Server Concurrency Models with ACE

Existing Web servers use a wide range of concurrency strategies to implement the role of the `Event Dispatcher`. These strategies include single-threaded concurrency (*e.g.*, Roxen), multi-process concurrency (*e.g.*, Apache), and multi-thread concurrency (*e.g.*, PHTTPD, Zeus, and JAWS). This section examines common Web concurrency models including *single-threaded Reactive*, *thread-per-request*, *thread pool*, and *thread-per-connection*. Each of these models is discussed below, focusing on the patterns they use and outlining how they can be implemented using ACE components.

Note how each concurrency model reuses most of the same patterns (*e.g.,* Reactor, Acceptor, and Active Object) and ACE components (*e.g.*, ACE `Reactor`, `HTTP Acceptor`, and `HTTP Handler`), by simply restructing these core architectural building blocks in different configurations. This high degree of consistency is common in applications and frameworks that are explicitly built using patterns. When patterns are used to structure and document applications and frameworks, nearly every class plays a well-defined role and collaborates effectively with its related classes.

4.4.1 The Single-threaded Reactive Web Server Model

In the single-threaded reactive model, all connections and HTTP requests are handled by the same thread of control. This thread is responsible for demultiplexing requests from different clients and dispatching event handlers to perform HTTP processing. If each request is processed in its entirety, the Reactive Web server is deemed *iterative*. If the processing of each request is split into chunks that are performed separately the Reactive Web server is deemed a *single-threaded concurrent* server.

The single-threaded reactive model is a highly portable model for implementing the `Event Dispatcher` role in a Web server. This model runs on any OS platform that supports event demultiplexing mechanisms such as `select` or `WaitForMultipleObjects`. The structure of a Reactive Web server based on the ACE `Reactor` is shown in Figure 13.

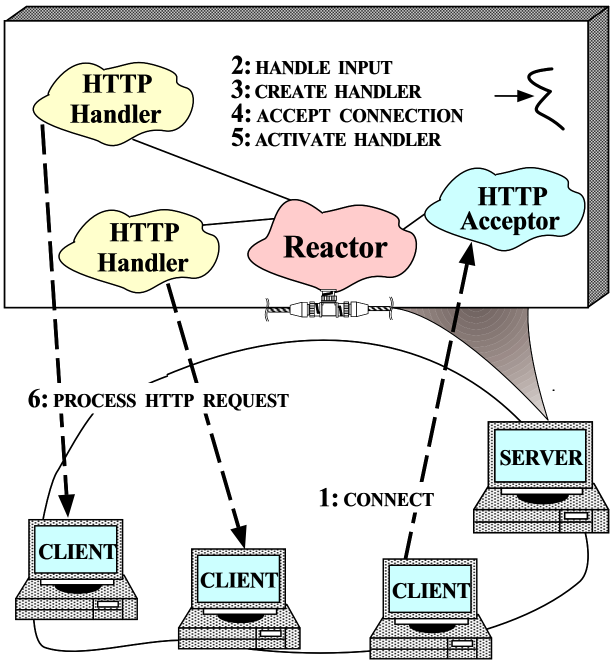

Figure 13: Single-threaded Reactive Web Server Model

The ACE `Reactor` is an OO implementation of the Reactor pattern that waits synchronously in a single-thread of control for the occurrence of various types of events (such as socket data, signals, or timeouts). When these events occur, the ACE `Reactor` demultiplexes the event to a pre-registered ACE `Event Handler` object and then dispatches the appropriate upcall method (*e.g.*, `handle_input`, `handle_signal`, `handle_timeout`) on the object.

Figure 14 illustrates how the Reactor pattern is used to trigger the acceptance of HTTP connections from clients. When a connection event arrives from a client the ACE `Reactor` invokes the `handle_input` factory method hook on the `HTTP Acceptor`. This hook accepts the connection and creates a new `HTTP Handler` object that processes the client request. Since this model is single-threaded and driven entirely by reactive I/O, each `HTTP Handler` must register with the ACE `Reactor`. The Reactor can then trigger the processing of HTTP requests from clients. When an HTTP `GET` request arrives, the ACE `Reactor` invokes the `handle_input` hook method on the `HTTP Handler`. This hook processes the request by retrieving the URI from the HTTP `GET` request and transferring the specified file to the client.

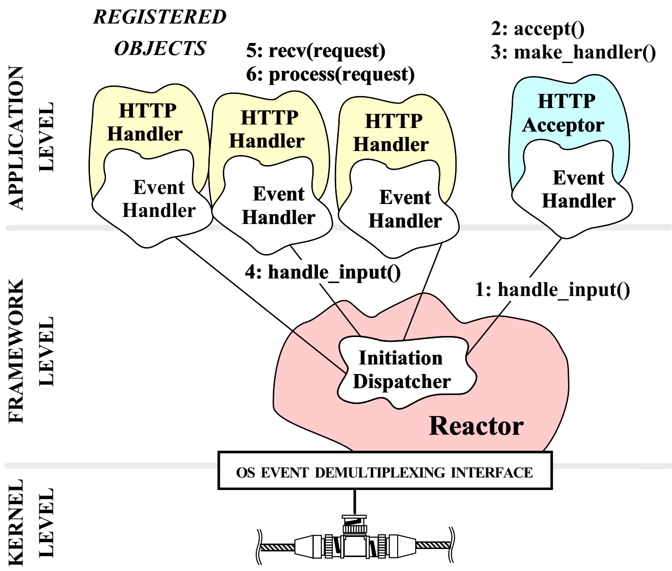

Figure 14: Accepting Connections and Processing HTTP Requests with the Reactor

To avoid blocking the server for extended periods of time, each I/O request can be broken into small chunks and sent separately. Therefore, the State pattern is typically used to maintain each `HTTP Handler`'s state (*e.g.*, awaiting the `GET` request, transmitting the n^{th} chunk, closing down, etc.). Likewise, the `Timer Queue` capabilities of the ACE `Reactor` can be used to prevent denial of service attacks where erroneous or malicious clients establish connections and consume Web server resources (*e.g,.* socket handles), but never send data to or receive data from the server.

The main advantages of the single-threaded reactive model are its portability and its low overhead for processing very small files. It is not suitable for high-performance Web servers, however, since it does not utilize parallelism effectively. In particular, all HTTP processing is serialized at the OS event demultiplexing level. This prevents Web servers from leveraging the parallelism available in the OS (*e.g.*, asynchronous I/O) and hardware (*e.g.*, DMA to intelligent I/O peripherals).

4.4.2 The Thread-per-Request Web Server Model

In the thread-per-request model, a new thread is spawned to handle each incoming request. Only one thread blocks on the acceptor socket. This acceptor thread is a factory that creates a new handler thread to process HTTP requests from each client.

The thread-per-request model is a widely used model for implementing multi-threaded Web servers. This model runs on any OS platform that supports preemptive multi-threading. The structure of a thread-per-request Web server based on the ACE `Reactor` and ACE Active Objects is shown in Figure 15.

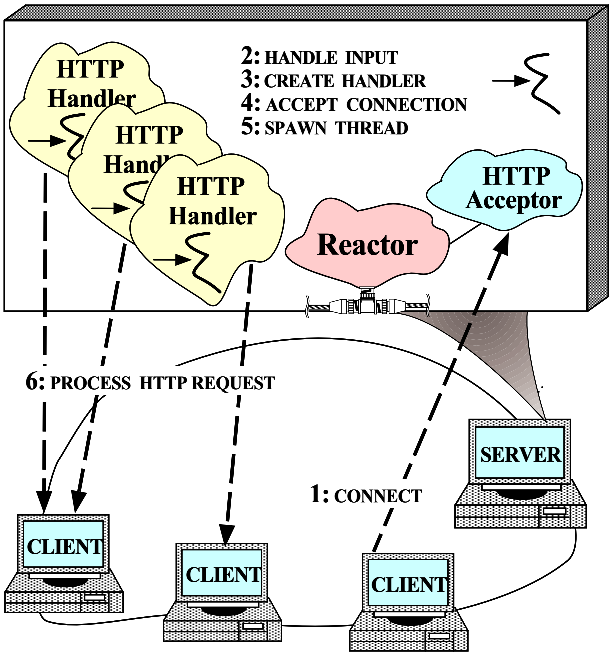

Figure 15: Thread-per-Request Web Server Model

Figure 15 illustrates how the ACE `Reactor` and `HTTP Acceptor` components can be reused for the thread-per-request model, *i.e.*, the ACE `Reactor` blocks in the main thread waiting for connection events. When a connection event occurs it notifies the `HTTP Acceptor` factory, which creates a new `HTTP Handler`.

The primary difference between the thread-per-request model and the single-threaded reactive model is that a new thread is spawned in each `HTTP Handler` to process every client request concurrently. Thus, the `HTTP Handler` plays the role of an Active Object, *i.e.*, the ACE `Reactor` thread that accepts the connection and invokes the `HTTP Handler` executes concurrently with the threads that perform HTTP processing. In HTTP 1.0, the lifecycle of an `HTTP Handler` Active Object is complete once the file transfer operation is finished.

The thread-per-request model is useful for handling requests for large files from multiple clients. It is less useful for small files from a single client due to the overhead of creating a new thread for each request. In addition, thread-per-request can consume a large number of OS resources if many clients perform requests simultaneously during periods of peak load.

4.4.3 The Thread Pool Web Server Model

In the thread pool model, a group of threads are pre-spawned during Web server initialization. Each thread blocks on the same acceptor socket, waiting for connections to arrive from clients. Pre-spawning eliminates the overhead of creating a new thread for each request. It also bounds the number of OS resources consumed by a Web server.

The thread pool model is generally the most efficient way to implement the `Event Dispatcher` in high-performance Web servers [32]. This model is most effective on OS platforms (such as Windows NT and Solaris 2.6) that permit simultaneous calls to the `accept` function on the same acceptor socket. On platforms that do not allow this (such as most SVR4 implementations of UNIX) it is necessary to explicitly serialize `accept` with an ACE `Mutex` synchronization object.

There are several variations of the thread pool model. Figure 16 and Figure 17 illustrate the *kernel-level* and *user-level* queue-based synchronous thread pool models, respectively. Figure 18 illustrates the asynchronous thread pool model. Each of these variants is outlined below:

The kernel-level queue-based synchronous thread pool: As shown in Figure 16, this model does not use a Reactor. Instead, each thread in the pool directly invokes the `handle_input` method of the `HTTP Acceptor`, which blocks awaiting client connections on the acceptor socket handle, which is associated with a TCP *socket listen queue* maintained in the OS kernel [38]. When clients connect, the OS selects a thread from the pool of `HTTP Acceptors` to `accept` the connection out of the kernel's listen queue. Once a connection is established, the acceptor "morphs" into an `HTTP Handler`, which performs a synchronous `read` on the newly connected handle. After the HTTP request has been read, the thread performs the necessary computation and filesystem operations to service the request. The requested file is then transmitted synchronously to the client. After the data transmission completes, the thread returns to the pool and reinvokes `HTTP Acceptor`'s `handle_input` method.

Client requests can execute concurrently until the number of simultaneous requests exceed the number of threads in the pool. At this point, additional requests are queued in the kernel's socket listen queue until a thread in the pool finishes

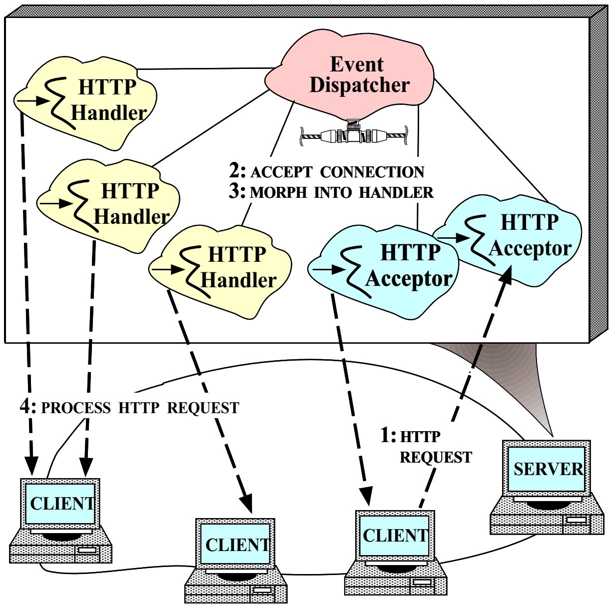

Figure 16: Kernel-level Queue-based Synchronous thread pool Web Server Model

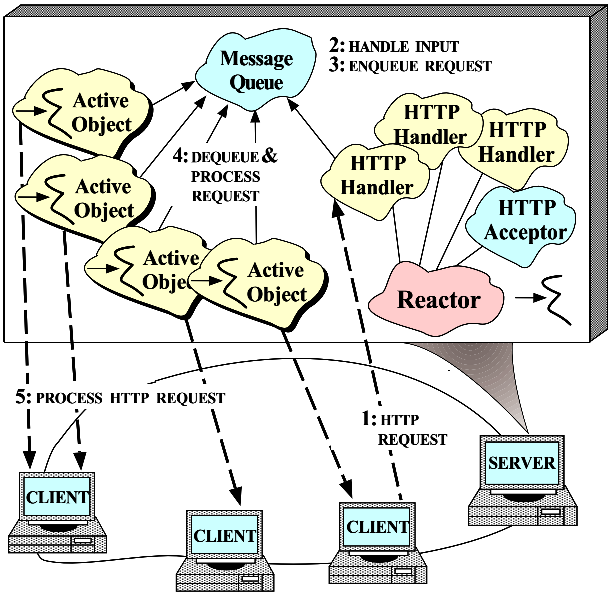

Figure 17: User-level Queue-based Synchronous Thread Pool Web Server Model

its processing and becomes available. To reduce latency, the thread pool can be configured to always have threads available to service new requests. However, the number of threads needed to support this policy can be very high during peak loads as threads block in long-duration synchronous I/O operations.

One drawback with the handle-based thread pool model is that the size of the socket listen queue is relatively small (*i.e.*, around 8 to 10 connections on most OS platforms). Therefore, high volume servers that receive hundreds of Web hits per second may not be able to accept connections fast enough to keep the kernel from rejecting clients. Moreover, it is not possible to prioritize which connections are dropped since the kernel does not distinguish among different clients.

The user-level queue-based synchronous thread pool: As shown in Figure 17, this model uses the Half-Sync/Half-Async pattern, which combines the Reactor and Active Object patterns. In this model, the ACE `Reactor` thread accepts connections from clients (via the `HTTP Acceptor`) and manages all the `HTTP Handlers`.

When HTTP requests arrive from clients they are validated briefly by the associated `HTTP Handler` in the ACE `Reactor` thread and then enqueued in the thread-safe ACE `Message Queue`. Unlike the kernel-level queue-based thread pool model described above, this queue is implemented in user-space and joins the "async" and "sync" layers in the Web server. Each Active Object in the thread pool invokes the `dequeue` method of the request queue, which blocks awaiting client requests.

Once an HTTP request has been dequeued by a thread in the pool this thread performs the necessary computation and filesystem operations to service the request. The requested data is then transmitted synchronously to the client. After the data transmission completes the thread returns to the pool and reinvokes `dequeue` method to retrieve another HTTP request.

In contrast with the kernel-level queue-based thread pool model, the user-level queue-based thread pool design makes it possible to accept (or reject) all incoming connections rapidly and prioritize how each client is processed. The primary drawback stems from the extra context switching and synchronization required to manage the queue in user-space within the Web server process.

The asynchronous thread pool: As shown in Figure 18, this model uses the ACE `Proactor`, which manages an I/O completion port. An I/O completion port is a thread-safe queue of I/O completion notifications that resides in the OS kernel (in contrast, the queue-based thread pool managed the thread in user-space). Each I/O operation is initiated and "handed off" to the kernel, where it runs to completion. Therefore, the initiating thread does not block. When these operations complete asynchronously, the kernel queues the resulting notifications at the appropriate I/O completion port.

Like the synchronous thread pool model, the asynchronous thread pool is created during Web server initialization. Unlike the synchronous model, however, the threads wait on an I/O completion port rather than waiting on `accept`. The OS queues up results from all asynchronous operations (*e.g.*, asyn-

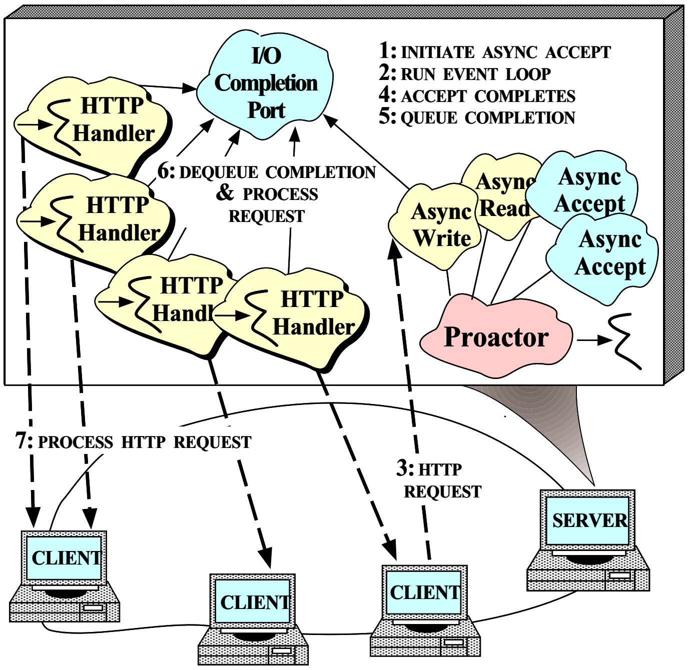

Figure 18: Asynchronous Thread Pool Web Server Model

chronous accepts, reads, and writes) on the I/O completion port. The result of each asynchronous operation is handled by a thread selected by the OS from the pool of threads waiting on the completion port. The thread that dequeues the completion notification need not be the same one that initiated the operation.

The asynchronous thread pool model is typically less resource intensive and provides more uniform latency under heavy workloads than synchronous thread pool models [32]. It is also more scalable since the same programming model works for a single thread, as well as multiple threads. The primary drawback with the asynchronous thread pool is that it is not portable to platforms that lack asynchronous I/O and proactive event dispatching. Windows NT 4.0 is the main contemporary operating system that support I/O completion ports in its OS API. The ACE `Proactor` encapsulates the Windows NT 4.0 I/O completion port asynchronous demultiplexing mechanism within a typesafe C++ wrapper.

4.4.4 The Thread-per-Connection Web Server Model

In the thread-per-connection model the newly created handler thread is responsible for the lifetime of the entire client session, rather than just a single request from the client. As with the thread-per-request model, only one thread blocks on the acceptor socket. This acceptor thread is a factory that creates a new handler thread to interact with each client for the duration of the session. Therefore, the new thread may serve multiple requests from a client before terminating.

The thread-per-connection model is not appropriate for HTTP 1.0 since protocol establishes a new connection for each request. Thus, thread-per-connection is equivalent to thread-per-request in HTTP 1.0. This model is applicable in HTTP 1.1, however, since it supports persistent connections [39, 40]. Figure 19 illustrates the thread-per-connection model.

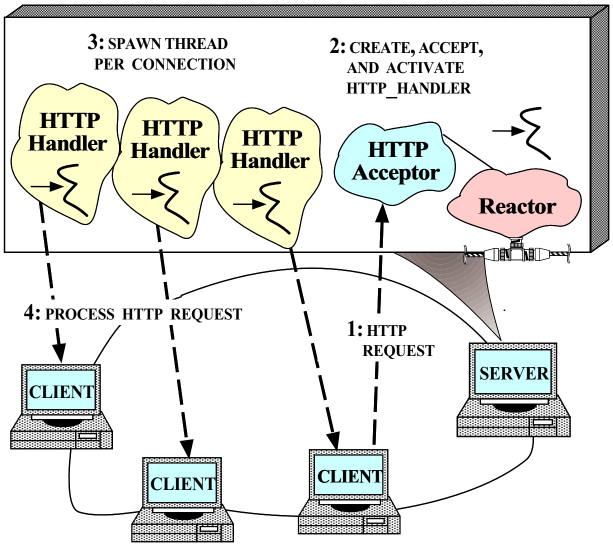

Figure 19: Thread-per-Connection Web Server Model

thread-per-connection provides good support for prioritization of client requests. For instance, higher priority clients can be associated with higher priority threads. Thus, request from higher priority clients will be served ahead of requests from lower priority clients since the OS can preempt lower priority threads. One drawback to thread-per-connection is that connections receiving considerably more requests than others can become a performance bottleneck. In contrast, the thread-per-request and thread pool models provide better support for load balancing.

5 Concluding Remarks

Computing power and network bandwidth has increased dramatically over the past decade. However, the design and implementation of communication software remains expensive and error-prone. Much of the cost and effort stems from the continual re-discovery and re-invention of fundamental design patterns and framework components across the software industry. Moreover, the growing heterogeneity of hardware architectures and diversity of OS and network platforms make it hard to build correct, portable, and efficient applications from scratch.

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 ACE framework described in this paper illustrates how the development of communication software, such as high-performance Web servers, can be simplified and unified. The key to the success of ACE 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 interprocess communication, event demultiplexing, dynamic configuration, concurrency, and synchronization.

The ACE C++ wrappers, framework components, distributed services, and higher-level distributed computing middleware components described in this paper are freely available at `www.cs.wustl.edu/~schmidt/ACE.html`. This URL contains complete source code, documentation, and example applications, including JAWS. ACE has been used in research and development projects at many universities and companies. For instance, it has been used to build avionics systems at Boeing [11]; telecommunication systems at Bellcore [13], Ericsson [41], and Motorola [9]; medical imaging systems at Siemens [31] and Kodak [10]; and many academic research projects.

References

[1] F. P. Brooks, *The Mythical Man-Month.* Reading, MA: Addison-Wesley, 1975.

[2] A. Gokhale and D. C. Schmidt, "Measuring the Performance of Communication Middleware on High-Speed Networks," in *Proceedings of SIGCOMM '96*, (Stanford, CA), pp. 306–317, ACM, August 1996.

[3] D. C. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale, "Alleviating Priority Inversion and Non-determinism in Real-time CORBA ORB Core Architectures," in *Proceedings of the* 4^{th} *IEEE Real-Time Technology and Applications Symposium*, (Denver, CO), IEEE, June 1998.

[4] G. Booch, *Object Oriented Analysis and Design with Applications (*2^{nd} *Edition).* Redwood City, California: Benjamin/Cummings, 1993.

[5] M. E. Fayad and D. C. Schmidt, "Object-Oriented Application Frameworks," *Communications of the ACM*, vol. 40, October 1997.

[6] R. Johnson and B. Foote, "Designing Reusable Classes," *Journal of Object-Oriented Programming*, vol. 1, pp. 22–35, June/July 1988.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] D. C. Schmidt and C. Cleeland, "Applying Patterns to Develop Extensible ORB Middleware," *IEEE Communications Magazine*, to appear 1999.

[13] 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.

[14] 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.

[15] D. C. Schmidt, T. H. Harrison, and E. Al-Shaer, "Object-Oriented Components for High-speed Network Programming," in *Proceedings of the* 1^{st} *Conference on Object-Oriented Technologies and Systems*, (Monterey, CA), USENIX, June 1995.

[16] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.2 ed., Feb. 1998.

[17] D. Box, *Essential COM.* Addison-Wesley, Reading, MA, 1997.

[18] A. Wollrath, R. Riggs, and J. Waldo, "A Distributed Object Model for the Java System," *USENIX Computing Systems*, vol. 9, November/December 1996.

[19] D. C. Schmidt and S. Vinoski, "Object Adapters: Concepts and Terminology," *C++ Report*, vol. 9, November/December 1997.

[20] D. C. Schmidt, A. Gokhale, T. Harrison, and G. Parulkar, "A High-Performance Endsystem Architecture for Real-time CORBA," *IEEE Communications Magazine*, vol. 14, February 1997.

[21] A. Gokhale and D. C. Schmidt, "Measuring and Optimizing CORBA Latency and Scalability Over High-speed Networks," *Transactions on Computing*, vol. 47, no. 4, 1998.

[22] D. C. Schmidt, D. L. Levine, and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, pp. 294–324, Apr. 1998.

[23] 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.

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

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

[26] J. Vlissides, "The Hollywood Principle," *C++ Report*, vol. 8, Feb. 1996.

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

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] P. Jain, S. Widoff, and D. C. Schmidt, "The Design and Performance of MedJava – A Distributed Electronic Medical Imaging System Developed with Java Applets and Web Tools," in *Proceedings of the* 4^{rd} *Conference on Object-Oriented Technologies and Systems*, USENIX, Apr. 1998.

[34] R. H. Halstead, Jr., "Multilisp: A Language for Concurrent Symbolic Computation," *ACM Trans. Programming Languages and Systems*, vol. 7, pp. 501–538, Oct. 1985.

[35] D. C. Schmidt, "IPC_SAP: An Object-Oriented Interface to Interprocess Communication Services," *C++ Report*, vol. 4, November/December 1992.

[36] 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.

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

[38] W. R. Stevens, *TCP/IP Illustrated, Volume 2*. Reading, Massachusetts: Addison Wesley, 1993.

[39] J. C. Mogul, "The Case for Persistent-connection HTTP," in *Proceedings of ACM SIGCOMM '95 Conference in Computer Communication Review*, (Boston, MA, USA), pp. 299–314, ACM Press, August 1995.

[40] 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. www.w3.org/.

[41] D. C. Schmidt and P. Stephenson, "Experiences Using Design Patterns to Evolve System Software Across Diverse OS Platforms," in *Proceedings of the* 9^{th} *European Conference on Object-Oriented Programming*, (Aarhus, Denmark), ACM, August 1995.