An Architectural Overview of the ACE Framework

A Case-study of Successful Cross-platform Systems Software Reuse

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

This article will appear in a special issue of USENIX Login edited by Peter Salus, 1998.

1 Introduction

Communication software for next-generation distributed applications should possess the following qualities:

- *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, high performance to bandwidth-intensive applications, and predictability to real-time applications.
- *Reliability* is needed to ensure that applications are robust, fault tolerant, and highly available.
- *Portability* is needed to reduce the effort required to support applications on heterogeneous OS platforms and compilers.

This article describes the software architecture of ACE [1], which is a freely available, open source C++ framework targeted for developers of high-performance and real-time communication services and applications.

The ACE framework provides an integrated set of components that help developers navigate between the "Scylla and Charybdis" limitations of (1) low-level native OS APIs, which are inflexible and non-portable and (2) higher-level distributed object computing middleware, which are often inefficient and unreliable. This article describes the structure and functionality of ACE, outlines several complex communication middleware applications that have been developed with ACE, and summarizes the key lessons learned developing and deploying reusable the OO communication software components and frameworks in ACE.

2 Overview of ACE

ACE is an object-oriented (OO) framework that implements core concurrency and distribution patterns [2] 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* [3];
- *Event demultiplexing and event handler dispatching* [4, 5, 6];
- *Interprocess communication* [7] and *shared memory management*;
- *Static and dynamic configuration [8, 9] of communication services*;
- *Concurrency and synchronization* [5, 10];
- *Distributed communication services* – such as naming, event routing [2], logging, time synchronization, and network locking;
- *Higher-level distributed computing middleware components* – such as Object Request Brokers (ORBs) [11], Web servers [12], and electronic medical imaging systems [13].

This section outlines the structure and functionality of the ACE framework.

2.1 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

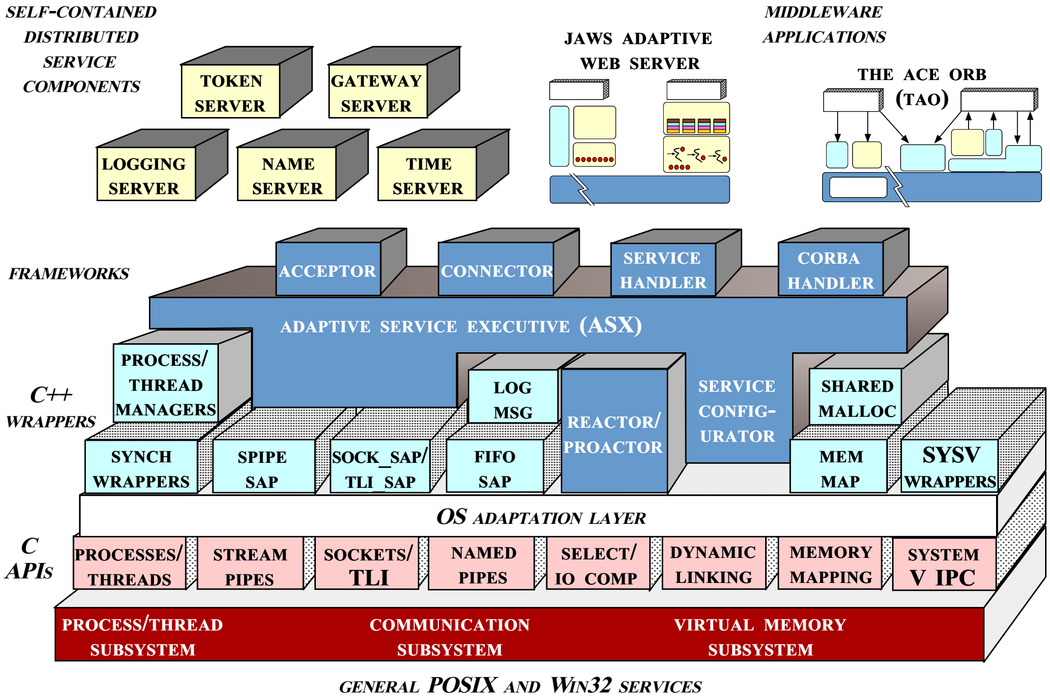

Figure 1: The Layering Structure of Components in the ACE Framework

layered architecture. Figure 1 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.

2.1.1 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.

2.1.2 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 1. 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 [10] and Polymorphic Futures [14].

IPC and filesystem components: The ACE C++ wrappers encapsulate local and/or remote IPC mechanisms [7] 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 [15].

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 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 the 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.

2.1.3 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 [4] and Proactor [6] 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 [3] 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 [9] 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, 1] 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 [16].

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 [17, 9], and (3) OS mechanisms, such as dynamic linking and multi-threading.

2.1.4 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:

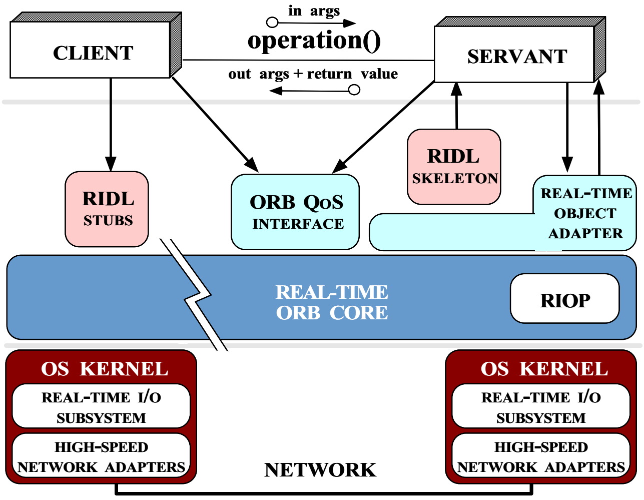

Figure 2: Components in the TAO Real-time ORB

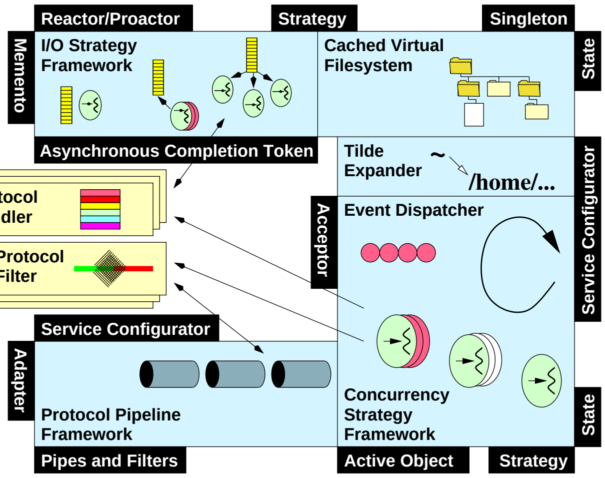

Figure 3: Architectural Overview of the JAWS Framework

Factoring out reusable distributed application building blocks: These service components provide reusable implementations of common distributed application tasks such as naming, event routing [2], 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.

2.1.5 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 [18], DCOM [19], or Java RMI [20]. 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.
- 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 [21] 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 2. TAO is based on the standard OMG CORBA reference model [18], with the enhancements designed to overcome the shortcomings of conventional ORBs [22] for high-performance and real-time applications. TAO, like ACE, is freely available at `www.cs.wustl.edu/~schmidt/TAO.html`.

JAWS: JAWS [23] is a high-performance, adaptive Web server built using the framework components and patterns provided by ACE. Figure 3 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/`.

3 Lessons Learned Developing and Deploying ACE

This section summarizes the lessons I've learned during the past seven years developing the reusable OO communication software components in the ACE framework and deploying ACE in a wide range of commercial applications in the avionics, telecommunications, and medical domains.

Software reuse fails largely to non-technical reasons: In theory, organizations recognize the importance of reuse as a means to reduce cycle-time and improve software quality. In practice, many factors conspire to make it hard to achieve systematic software reuse. Most of the impediments are largely political, economical, organizational, and psychological, rather than technical. For instance, teams that develop reusable middleware platforms are often viewed with suspicion by application development teams, who resent the fact that they are no longer empowered to make key architectural decisions.

Successful reuse-in-the-large requires prerequisites: In my experience, large-scale reuse of software works best when the following conditions apply:

- **The marketplace is highly competitive:** In a competitive environment, time-to-market is crucial. Therefore, it is essential to leverage existing software to substantially reduce development effort and cycle time. When a market place is not competitive, however, there is often a tendency to reinvent rather than reuse.

- **The application domain is challenging:** Components that are relatively easy to develop, such as generic linked lists, stacks, or queues, are often rewritten from scratch, rather than reused. In contrast, developers are generally willing to reuse highly complex components, such as dynamic scheduling frameworks [24] or real-time ORBs [21], since building complete solutions from scratch is too difficult, costly, and time-consuming.

- **The corporate culture is supportive:** It is hard to develop high-quality reusable components and frameworks. In particular, it is hard to reap the benefits of reuse immediately. A great deal of effort must be expended initially to produce efficient, flexible, and well-documented reusable software artifacts. Thus, an organization must support an effective process in order for reuse to flourish. For instance, developers must be rewarded, not punished, for taking the time to build robust reusable components. Moreover, the reuse process must reward production of concrete software artifacts, rather than endless abstract meta-models or high-level design documents.

In my experience, these prerequisites often do not exist in contemporary organizations. In such cases, I've observed that organizations often fall victim to the "not-invented-here" syndrome and redevelop most of their software components from scratch. Unfortunately, increasing deregulation and global competition make it hard to succeed with this type of development process.

Iteration and incremental growth is essential: It is crucial for organizations to explicitly recognize that good components, frameworks, and software architectures require time to craft, hone, and apply. In general, developing, using, and reusing software requires a mature organization that can distinguish key sources of variability and commonality in its application domain. Identifying and separating these concerns requires multiple iterations.

For reuse to succeed in-the-large, management must have the vision and resolve to support the incremental evolution of reusable software. Fred Brook's observation that "Plan to throw the first one away, you will anyway" [25] applies as much today as it did 20 years ago. Moreover, in my experience, "the best is often the enemy of the good" when it comes to deploying reusable software frameworks and components. Often, an 80% solution that can be deployed and evolved incrementally is preferable to waiting for a 100% solution that never ships.

There's no substitute for hands-on experience: Developing high quality communication software is hard; developing high quality reusable communication software is even harder. The principles, methods, and skills required to develop reusable software simply cannot be learned by generalities. Instead, developers must learn through hands-on experience how to design, implement, optimize, validate, maintain, and enhance reusable software components and frameworks. Only by activately engaging in these activities will developers truly internalize good development practices and patterns.

Integrate infrastructure developers with application developers: Most useful components and frameworks originate from solving real problems in a particular application domain, such as telecommunications, medical imaging, avionics, and Web programming. A time-honored way of producing effective reusable components, therefore, is to *generalize* them from working systems and applications. This was how ACE evolved.

I've found that creating "component teams," which build reusable frameworks in isolation from application teams, is often counter-productive. Without intimate feedback from application developers, the software artifacts produced by component teams rarely solve real problems and are unlikely to be reused systematically.

Design to an architecture rather than program to a particular middleware technology "standard": It is very risky to expect that emerging industry middleware standards, like CORBA, DCOM, or Java RMI, will automatically eliminate the complexity of developing communication software. No single solution is a panacea, nor are "standards" necessarily ubiquitous or implemented consistently.

Therefore, for complex communication software systems it is essential to design and use *architectures* that can transcend any specific middleware technology standard. I've found it is much more effective to devise a common software architecture that can be instantiated on multiple middleware platforms, rather than programming directly to a particular middleware API, which can rapidly become obsolete.

OS API "wars" are largely irrelevant: ACE's OS adaptation layer makes the selection of the native OS API, *e.g.*, POSIX vs. Win32 vs. real-time operating systems, largely an implementation detail. Using ACE, it is straightforward to develop highly portable communication software that runs efficiently on a wide range of operating systems and C++ compilers. Moreover, ACE provides this portability without incurring the performance penalties associated with interpreted virtual machines.[1] Thus, the portability provided by ACE allows developers to select an OS platform based on features, price, performance, development tools, and ease of integration with other applications.

Beware of simple(-minded) solutions to complex software problems: Trying to apply overly simple solutions to complex problems is an exercise in frustration and a recipe for failure. For instance, attempting to translate the software implementations entirely from high-level SDL specifications or "analysis rules" rarely succeed for complex communication systems. Likewise, using trendy OO design methodologies, modeling notations, and programming languages is no guarantee of success. In my experience, there's simply no substitute for employing skilled software developers, which leads to the following final "lesson learned."

Respect and reward quality developers and architects: Ultimately, reusable components and frameworks are only as good as the people who build and use them. Developing robust, efficient, and reusable middleware requires teams with a wide range of skills. We need expert analysts and designers who have mastered design patterns, software architectures, and communication protocols to alleviate the inherent and accidental complexities of communication software. Moreover, we need expert programmers who can implement these patterns, architectures, and protocols in reusable frameworks and components.

In my experience, it is exceptionally hard to find high quality software developers. Ironically, many companies treat their developers as interchangeable, "unskilled labor," who can be replaced easily. Over time, companies who respect and reward their high quality software developers are increasingly outperforming those who do not.

4 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 patterns and framework components across the software industry. However, the growing heterogeneity of hardware architectures, the diversity of OS and network platforms, and global competition make it increasingly costly to build correct, portable, and efficient applications from scratch.

Object-oriented application frameworks and 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 article illustrates how the development of communication software like ORBs and Web servers, can be significantly simplified and unified.

The widespread adoption of ACE is a testament to the power of an open source software process and to the benefits of systematic software reuse in complex communication systems. One key to the success of ACE has been its ability to capture common communication software design patterns and consolidate these patterns into flexible framework components. The framework components efficiently encapsulate and enhance low-level OS mechanisms for interprocess communication, event demultiplexing, dynamic configuration, concurrency, synchronization, and file system access.

The ACE C++ wrappers, framework components, distributed services, and higher-level distributed computing middleware components described in this article are freely available at `www.cs.wustl.edu/~schmidt/ACE.html`. This URL contains complete source code, documentation, and

[1] However, a Java version [26] of many ACE components is also available at `www.cs.wustl.edu/~schmidt/JACE.html`.

example applications, including JAWS and TAO.

ACE has been used in research and development projects at many universities and companies. For instance, ACE has been used to build real-time avionics systems at Boeing [27]; telecommunication systems at Bellcore [4], Ericsson [28], Motorola [2], and Lucent; medical imaging systems at Siemens [9] and Kodak [16]; and distributed simulation systems at SAIC/DARPA. It is also widely used for research projects and classroom instruction.

A description of many of the projects using the ACE, TAO, and JAWS frameworks are available at `www.cs.wustl.edu/~schmidt/ACE-users.html`. In addition, `comp.soft-sys.ace` is a USENET newsgroup devoted to ACE-related topics.

Acknowledgements

Much of the success of ACE is due to the dedication of the core development team at Washington University, as well as the hundreds of developers throughout the Internet who contribute their time and effort to improve ACE. I greatly appreciate their help, as well as the support of USENIX, which has sponsored some of the research on TAO's real-time scheduling service.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[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] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

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

[16] 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.

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

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

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

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

[21] 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.

[22] 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 Fourth IEEE Real-Time Technology and Applications Symposium*, (Denver, CO), IEEE, June 1998.

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

[24] C. D. Gill, D. L. Levine, and D. C. Schmidt, "Evaluating Strategies for Real-Time CORBA Dynamic Scheduling," *submitted to the International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*.

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

[26] P. Jain and D. Schmidt, "Experiences Converting a C++ Communication Software Framework to Java," *C++ Report*, vol. 9, January 1997.

[27] 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.

[28] 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.