C++ Network Programming

Volume 1

Mastering Complexity with ACE and Patterns

Douglas C. Schmidt
Stephen D. Huston

Foreword by Steve Vinoski

C++ In-Depth Series • Bjarne Stroustrup

Library of Congress Cataloging-in-Publication Data

Schmidt, Douglas C.
C++ network programming / Douglas C. Schmidt, Stephen D. Huston.
p. cm.
Includes bibliographical references and index.
Contents: Vol. 1. Mastering complexity with ACE and patterns.
ISBN 0-201-60464-7 (v. 1 : pbk.)
1. C++ (Computer program language) 2. Object-oriented programming (Computer science) 3. Computer networks. I. Huston, Stephen D. II. Title.

QA 76.73.C153 S368 2002
005.2'762—dc21

2001053345

ISBN 0-201-60464-7

Text printed in the United States at Offset Paperback Manufacturers in Laflin, Pennsylvania.

10th Printing September 2009

Foreword

As I write this foreword I'm traveling through Europe, relying on the excellent European public transportation infrastructure. Being an American, I'm fascinated and amazed by this infrastructure. Wherever I land at an airport I have easy access to trains and buses that are fast, clean, reliable, on time, and perhaps most importantly, going directly to my destination. Departure and arrival announcements are available in multiple languages. Signs and directions are easy to follow, even for non-native speakers like me.

I live and work in the Boston area, and like most Americans I rely almost entirely on my automobile to get from one place to the next. Except for an occasional use of the Boston subway system, I use my car to get around because the public transportation infrastructure is too limited to get me to my destination. Since millions of others in Boston and elsewhere are in the same predicament, our highway infrastructure is now well past the point of coping with the traffic volume. I know I'd be appalled if I knew exactly how much of my life I've wasted sitting in traffic jams.

There are some interesting similarities between networked computing systems and transportation systems, the most significant of these being that the success of both depends on scalable infrastructure. Scalable transportation systems comprise not just obvious infrastructure elements, such as trains and rails or airplanes and airports. They also require scheduling, routing, maintenance, ticketing, and monitoring, for example, all of which must scale along with the physical transportation system itself. Similarly, networked computing requires not only host machines and networks—the physical computing and communication infrastructure—

but also software-based scheduling, routing, dispatching, configuration, versioning, authentication, authorization, and monitoring that allows the networked system to scale as necessary.

An ironic fact about infrastructure is that it's extremely difficult to do well, and yet the more transparent to the user it is, the more successful we consider it to be. Despite the rugged terrain of the Swiss Alps, for example, a few architects, engineers, and builders have applied their expertise to provide an efficient transportation system that millions of people in Switzerland use daily with ease. In fact, the system is so reliable and easy to use that you quickly take it for granted, and it becomes transparent to you. For example, when boarding the Swiss railway your focus is simply on getting from one point to another, not on the machinery used to get you there. Unless you're a tourist, you probably miss the fact that you're traversing a tunnel that took years to design and build, or ascending an incline so steep that the railway includes a cog rail to help the train climb. The rail infrastructure does flawlessly what it's supposed to do, and as a result, you don't even notice it.

This book is about infrastructure software, normally called *middleware*, for networked computing systems. It's called middleware because it's the "waist in the hourglass" that resides above the operating system and networks, but underneath the application. Middleware comes in a wide variety of shapes, sizes, and capabilities, ranging from J2EE application servers, asynchronous messaging systems, and CORBA ORBs to software that monitors sockets for small embedded systems. Middleware must support an ever-wider variety of applications, operating systems, networking protocols, programming languages, and data formats. Without middleware, taming the ever-increasing diversity and heterogeneity in networked computing systems would be tedious, error prone, and expensive.

Despite the variety of types of middleware, and the variety of issues that middleware addresses, different types of middleware tend to use the same patterns and common abstractions to master complexity. If you were to peek inside a scalable and flexible application server, messaging system, or CORBA ORB, for example, you would likely find that they employ similar techniques for tasks such as connection management, concurrency, synchronization, event demultiplexing, event handler dispatching, error logging, and monitoring. Just as the users of the Swiss railways far outnumber those who designed and built it, the number of users of successful middleware far exceeds the number of people who designed and built it. If you design, build, or use middleware, your success depends on knowing, understanding, and applying these common patterns and abstractions.

While many understand the need for scalability and flexibility in middleware, few can provide it as effectively as the ADAPTIVE Communication Environment (ACE) that Doug Schmidt and Steve Huston describe in this book. ACE is a widely used C++ toolkit that captures common patterns and abstractions used in a variety of highly successful middleware and networked applications. ACE has become the basis for many networked computing systems, ranging from real-time avionics applications to CORBA ORBs to mainframe peer-to-peer communication support.

Like all good middleware, ACE hides the complexity of the diverse and heterogeneous environments beneath it. What sets ACE apart from most other infrastructure middleware, however, is that even though it allows for maximum flexibility wherever needed by the application, it doesn't degrade the performance or scalability of the system. Being a long-time middleware architect myself, I know all too well that achieving both performance and flexibility in the same package is hard.

In a way, though, the flexibility and performance aspects of ACE don't surprise me. Due to my long-time association with Doug, I'm well aware that he is a pioneer in this area. The wide variety of scalable, high-performing, and flexible middleware that exists today clearly bears his mark and influence. His teaming with Steve, who's a gifted C++ developer and author whose work on ACE has led to many improvements over the years, has yielded a work that's a "must read" for anyone involved in designing, building, or even using middleware. The increasing pervasiveness of the World Wide Web and of interconnected embedded systems means that the number, scale, and importance of networked computing systems will continue to grow. It's only through understanding the key patterns, techniques, classes, and lessons that Doug and Steve describe in this book that we can hope to supply the middleware infrastructure to make it all transparent, efficient, and reliable.

Steve Vinoski
Chief Architect & Vice President, Platform Technologies
IONA Technologies
September 2001

CHAPTER 0

Design Challenges, Middleware Solutions, and ACE

CHAPTER SYNOPSIS

This chapter describes the paradigm shift that occurs when transitioning from *stand-alone application architectures* to *networked application architectures*. This shift yields new challenges in two categories: those in the problem space that are oriented to software architecture and design and those in the solution space that are related to software tools and techniques used to implement networked applications. This chapter first presents a domain analysis of design dimensions affecting the former category, and the middleware that is motivated by and applied to the latter category. The chapter then introduces the ACE toolkit and the example networked application that's used to illustrate the solutions throughout this book.

0.1 Challenges of Networked Applications

Most software developers are familiar with stand-alone application architectures, in which a single computer contains all the software components related to the graphical user interface (GUI), application service processing, and persistent data resources. For example, the stand-alone application architecture illustrated in Figure 0.1 consolidates the GUI, service processing, and persistent data resources within a single computer, with all peripherals attached directly. The flow of control in a stand-alone application resides solely on the computer where execution begins.

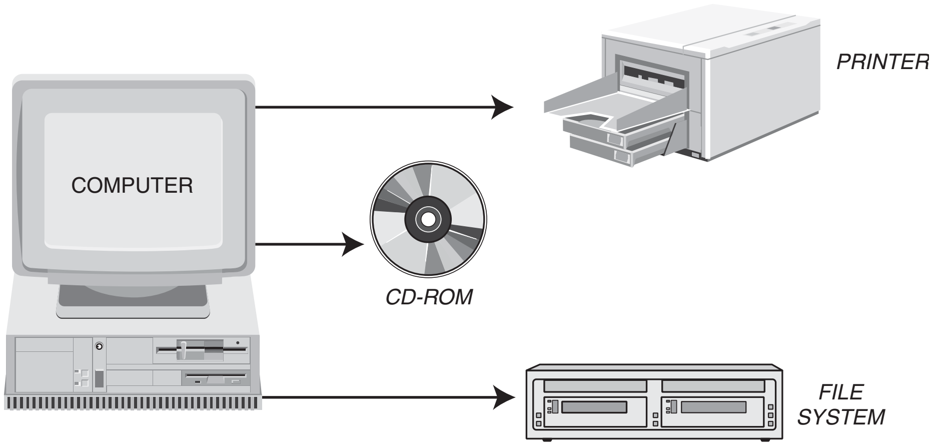


Figure 0.1: **A Stand-alone Application Architecture**

In contrast, networked application architectures divide the application system into *services* that can be shared and reused by multiple applications. To maximize effectiveness and usefulness, services are distributed among multiple computing devices connected by a network, as shown in Figure 0.2. Common network services provided to *clients* in such environments include distributed naming, network file systems, routing table management, logging, printing, e-mail, remote login, file transfer, Web-based e-commerce services, payment processing, customer relationship management, help desk systems, MP3 exchange, streaming media, instant messaging, and community chat rooms.

The networked application architecture shown in Figure 0.2 partitions the interactive GUI, instruction processing, and persistent data resources among a number of independent *hosts* in a network. At run time, the flow of control in a networked application resides on one or more of the hosts. All the system components communicate cooperatively, transferring data and execution control between them as needed. Interoperability between separate components can be achieved as long as compatible communication protocols are used, even if the underlying networks, operating systems, hardware, and programming languages are heterogeneous [HV99]. This delegation of networked application service responsibilities across multiple hosts can yield the following benefits:

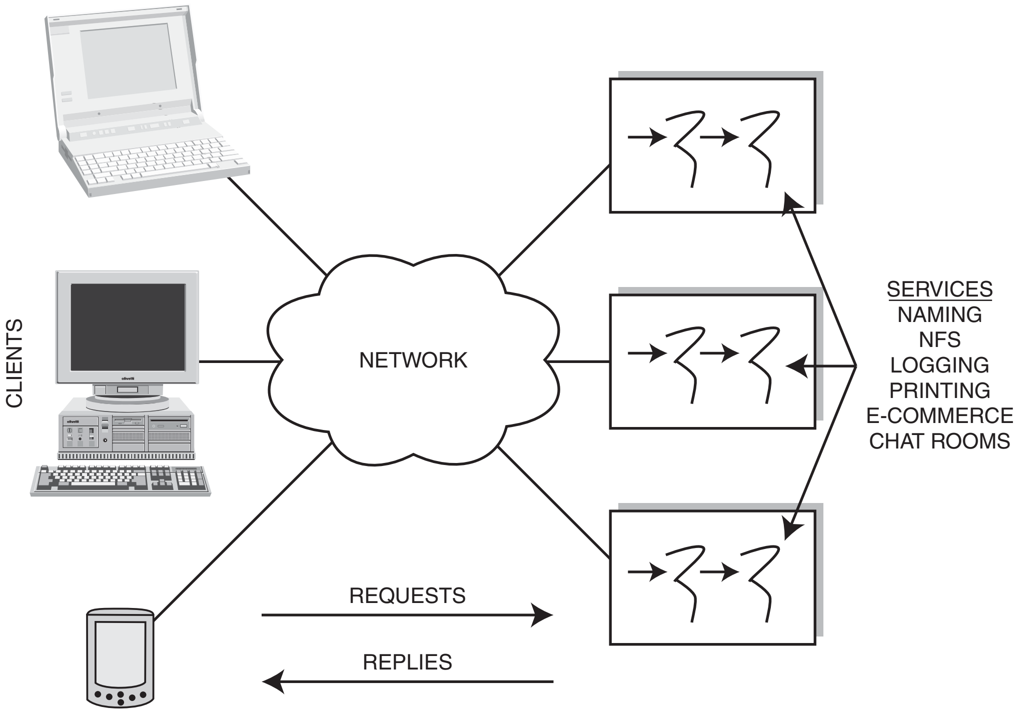


Figure 0.2: **A Common Networked Application Environment**

1. **Enhanced connectivity and collaboration** disseminates information rapidly to more potential users. This connectivity avoids the need for manual information transfer and duplicate entry.
2. **Improved performance and scalability** allows system configurations to be changed readily and robustly to align computing resources with current and forecasted system demand.
3. **Reduced costs** by allowing users and applications to share expensive peripherals and software, such as sophisticated database management systems.

Your job as a developer of networked applications is to understand the services that your applications will provide and the environment(s) available to provide them, and then

1. Design mechanisms that services will use to communicate, both between themselves and with clients.

2. Decide which architectures and service arrangements will make the most effective use of available environments.
3. Implement these solutions using techniques and tools that eliminate complexity and yield correct, extensible, high-performance, low-maintenance software to achieve your business's goals.

This book provides the information and tools you need to excel at these tasks.

Your job will not be easy. Networked applications are often much harder to design, implement, debug, optimize, and monitor than their stand-alone counterparts. You must learn how to resolve the inherent and accidental complexities [Bro87] associated with developing and configuring networked applications. *Inherent complexities* arise from key *domain* challenges that complicate networked application development, including

- Selecting suitable communication mechanisms and designing protocols to use them effectively
- Designing network services that utilize the available computing resources efficiently and reduce future maintenance costs
- Using *concurrency* effectively to achieve predictable, reliable, high performance in your system
- Arranging and configuring services to maximize system availability and flexibility.

Dealing with inherent complexity requires experience and a thorough understanding of the domain itself. There are many design tradeoffs related to these inherent complexity issues that we will investigate in Chapters 1 and 5.

Accidental complexities arise from limitations with tools and techniques used to develop networked application software, including

- The lack of type-safe, portable, and extensible native OS *APIs*
- The widespread use of *algorithmic decomposition*, which makes it unnecessarily hard to maintain and extend networked applications
- The continual rediscovery and reinvention of core networked application concepts and capabilities, which keeps software life-cycle costs unnecessarily high

Networked application developers must understand these challenges and apply techniques to deal with them effectively. Throughout this book we illustrate by example how ACE uses object-oriented techniques and C++ language features to address the accidental complexities outlined above.

0.2 Networked Application Design Dimensions

It's possible to learn programming APIs and interfaces without appreciating the key design dimensions in a domain. In our experience, however, developers with deeper knowledge of networked application domain fundamentals are much better prepared to solve key design, implementation, and performance challenges effectively. We therefore explore the core architectural design dimensions for networked application development first. We focus on servers that support multiple services, or multiple instances of a service, and that collaborate with many clients simultaneously, similar to the networked application environment shown in Figure 0.2.

The design dimensions discussed in this book were identified by a thorough *domain analysis* based on hands-on design and implementation experience with hundreds of production networked applications and systems developed over the past decade. A domain analysis is an inductive, feedback-driven process that examines an application domain systematically to identify its core challenges and design dimensions in order to map them onto effective solution techniques. This process yields the following benefits:

• **It defines a common vocabulary of domain abstractions,** which enables developers to communicate more effectively with each other [Fow97]. In turn, clarifying the vocabulary of the problem space simplifies the mapping onto a suitable set of patterns and software abstractions in the solution space. For example, a common understanding of network protocols, *event* demultiplexing strategies, and concurrency architectures allows us to apply these concepts to our discussions of wrapper facades, as well as to our discussions of ACE frameworks in [SH].

• **It enhances reuse** by separating design considerations into two categories:

1. Those that are specific to particular types of applications and
2. Those that are common to all applications in the domain.

By focusing on common design concerns in a domain, application and *middleware* developers can recognize opportunities for adapting or building reusable software class libraries. When the canonical control flows between these class libraries are factored out and reintegrated, they can form middleware frameworks, such as those in ACE, that can reduce subsequent application development effort significantly. In a mature domain,

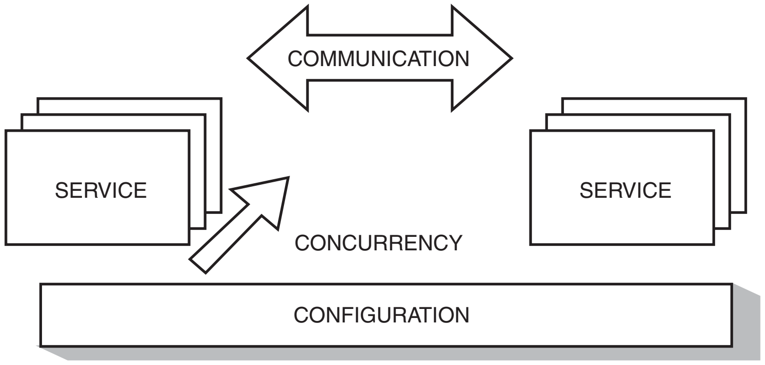


Figure 0.3: **Networked Application Design Dimensions**

application-specific design considerations can be addressed systematically by extending and customizing existing middleware frameworks via object-oriented language features, such as inheritance, dynamic binding, parameterized types, and exceptions.

Within the domain of networked applications, developers are faced with design decisions in each of the four dimensions depicted in Figure 0.3. These design dimensions are concerned mainly with managing inherent complexities. They are therefore largely independent of particular life-cycle processes, design methods and notations, programming languages, operating system platforms, and networking hardware. Each of these design dimensions is composed of a set of relatively independent alternatives. Although mostly orthogonal to each other, changes to one or more dimensions of your networked application can change its "shape" accordingly. Design changes therefore don't occur in isolation. Keep this in mind as you consider the following design dimensions:

1. **Communication dimensions** address the rules, form, and level of abstraction that networked applications use to interact.
2. **Concurrency dimensions** address the policies and mechanisms governing the proper use of processes and threads to represent multiple service instances, as well as how each service instance may use multiple threads internally.
3. **Service dimensions** address key properties of a networked application service, such as the duration and structure of each service instance.

4. **Configuration dimensions** address how networked services are identified and the time at which they are bound together to form complete applications. Configuration dimensions often affect more than one service, as well as the relationships between services.

We examine the first two dimensions in more depth in Chapters 1 and 5, respectively, while the third and fourth are discussed in [SH]. We illustrate the key vocabulary, design trade-offs, and solution abstractions first, followed by the platform capabilities related to each dimension, its associated accidental complexities, and the solutions provided by ACE, which evolved over the past decade in response to these design dimensions. As you'll see, the ACE toolkit uses time-proven object-oriented partitioning, *interface* design, data encapsulation patterns, and C++ features to enable the design dimensions of your networked applications to vary as independently and portably as possible.

0.3 Object-Oriented Middleware Solutions

Some of the most successful techniques and tools devised to address accidental and inherent complexities of networked applications have centered on object-oriented middleware, which helps manage the complexity and heterogeneity in networked applications. Object-oriented middleware provides reusable service/protocol component and framework software that functionally bridges the gap between

1. End-to-end application functional requirements and
2. The lower-level operating systems, networking *protocol stacks*, and hardware devices.

Object-oriented middleware provides capabilities whose qualities are critical to help simplify and coordinate how networked applications are connected and how they interoperate.

0.3.1 Object-Oriented Middleware Layers

Networking protocol stacks, such as TCP/IP [Ste93], can be decomposed into multiple layers, such as the physical, data-link, network, transport, session, presentation, and application layers defined in the OSI reference model [Bla91]. Likewise, object-oriented middleware can be decomposed

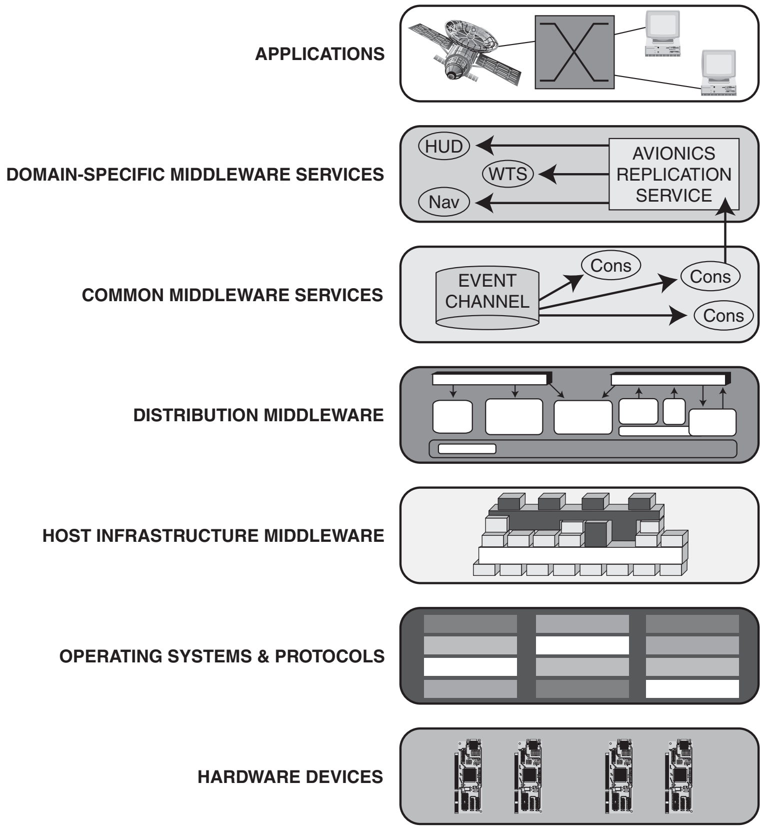

APPLICATIONS
DOMAIN-SPECIFIC MIDDLEWARE SERVICES
HUD
WTS
Nav
AVIONICS REPLICATION SERVICE
COMMON MIDDLEWARE SERVICES
EVENT CHANNEL
Cons
Cons
Cons
DISTRIBUTION MIDDLEWARE
HOST INFRASTRUCTURE MIDDLEWARE
OPERATING SYSTEMS & PROTOCOLS
HARDWARE DEVICES

Figure 0.4: **Object-Oriented Middleware Layers in Context**

into multiple layers [SS01], as shown in Figure 0.4. A common hierarchy of object-oriented middleware includes the layers described below:

Host infrastructure middleware encapsulates OS concurrency and inter-process communication (IPC) mechanisms to create object-oriented network programming capabilities. These capabilities eliminate many tedious, error-prone, and nonportable activities associated with developing networked applications via native OS APIs, such as Sockets or POSIX threads (Pthreads). Widely used examples of host infrastructure middleware include Java Packages [AGH00] and ACE.

Distribution middleware uses and extends host infrastructure middleware in order to automate common network programming tasks, such as connection and memory management, *marshaling* and *demarshaling*, end-point and request *demultiplexing*, synchronization, and multithreading. Developers who use distribution middleware can program distributed applications much like stand-alone applications, that is, by invoking operations on target objects without concern for their location, language, OS, or hardware [HV99]. At the heart of distribution middleware are *Object Request Brokers* (ORBs), such as COM+ [Box97], Java RMI [Sun98], and *CORBA* [Obj01].

Common middleware services augment distribution middleware by defining higher-level domain-independent services, such as event notification, logging, persistence, security, and recoverable transactions. Whereas distribution middleware focuses largely on managing end-system resources in support of an object-oriented distributed programming model, common middleware services focus on allocating, scheduling, and coordinating various resources throughout a distributed system. Without common middleware services, these end-to-end capabilities would have to be implemented *ad hoc* by each networked application.

Domain-specific middleware services satisfy specific requirements of particular domains, such as telecommunications, e-commerce, health care, process automation, or avionics. Whereas the other object-oriented middleware layers provide broadly reusable "horizontal" mechanisms and services, domain-specific services target vertical markets. From a "commercial off-the-shelf" (COTS) perspective, domain-specific services are the least mature of the middleware layers today. This is due in part to the historical lack of middleware standards needed to provide a stable base upon which to create domain-specific services.

Object-oriented middleware is an important tool for developing networked applications. It provides the following three broad areas of improvement for developing and evolving networked applications:

1. **Strategic focus,** which elevates application developer focus beyond a preoccupation with low-level OS concurrency and networking APIs. A solid grasp of the concepts and capabilities underlying these APIs is foundational to all networked application development. However, middleware helps abstract the details away into higher-level, more easily used artifacts. Without needing to worry as much about low-

level details, developers can focus on more strategic, application-centric concerns.

2. **Effective reuse,** which amortizes software life-cycle effort by leveraging previous development expertise and reifying implementations of key patterns [SSRB00, GHJV95] into reusable middleware frameworks. In the future, most networked applications will be assembled by integrating and scripting domain-specific and common "pluggable" middleware service components, rather than being programmed entirely from scratch [Joh97].
3. **Open standards,** which provide a portable and interoperable set of software artifacts. These artifacts help to direct the focus of developers toward higher-level software application architecture and design concerns, such as interoperable security, layered distributed resource management, and fault tolerance services. An increasingly important role is being played by open and/or standard COTS object-oriented middleware, such as CORBA, Java virtual machines, and ACE, which can be purchased or acquired via open-source means. COTS middleware is particularly important for organizations facing time-to-market pressures and limited software development resources.

Although distribution middleware, common middleware services, and domain-specific middleware services are important topics, they are not treated further in this book for the reasons we explore in the next section. For further coverage of these topics, please see either `http://ace.ece.uci.edu/middleware.html` or *Advanced CORBA Programming with C++* [HV99].

0.3.2 The Benefits of Host Infrastructure Middleware

Host infrastructure middleware is preferred over the higher middleware layers when developers are driven by stringent *quality of service (QoS)* requirements and/or cost containment. It's also a foundational area for advancing the state-of-the-art of middleware. These areas and their rationale are discussed below.

Meeting stringent QoS requirements. Certain types of applications need access to native OS IPC mechanisms and protocols to meet stringent efficiency and predictability QoS requirements. For example, multimedia applications that require long-duration, bidirectional bytestream communication services are poorly suited to the synchronous request/response paradigm provided by some distribution middleware [NGSY00]. Despite major advances [GS99, POS+00] in optimization technology, many conventional distribution middleware implementations still incur significant throughput and *latency* overhead and lack sufficient hooks to manipulate other QoS-related properties, such as *jitter* and dependability.

In contrast, host infrastructure middleware is often better suited to ensure end-to-end QoS because it allows applications to

- Omit functionality that may not be necessary, such as omitting marshaling and demarshaling in homogeneous environments
- Exert fine-grained control over communication behavior, such as supporting IP multicast transmission and asynchronous I/O and
- Customize networking protocols to optimize network *bandwidth* usage or to substitute shared memory communication in place of loopback network communication

By the end of the decade, we expect research and development (R&D) on distribution middleware and common services will reach a point where its QoS levels rival or exceed that of handwritten host infrastructure middleware and networked applications. In the meantime, however, much production software must be written and deployed. It's within this context that host infrastructure middleware plays such an important role by elevating the level of abstraction at which networked applications are developed without unduly affecting their QoS.

Cost containment. To survive in a globally competitive environment, many organizations are transitioning to object-oriented development processes and methods. In this context, host infrastructure middleware offers powerful and time-proven solutions to help contain the costs of the inherent and accidental complexities outlined in Section 0.1, page 4.

For example, adopting new compilers, development environments, debuggers, and toolkits can be expensive. Training software engineers can be even more expensive due to steep learning curves needed to become proficient with new technologies. Containing these costs is important when embarking on software projects in which new technologies are being evaluated or employed. Host infrastructure middleware can be an effective tool for leveraging existing OS and networking experience, knowledge, and skills while expanding development to new platforms and climbing the learning curve toward more advanced, cost-saving software technologies.

Advancing the state-of-the-practice by improving core knowledge. A solid understanding of host infrastructure middleware helps developers identify higher-level patterns and services so they can become more productive in their own application domains. There are many new technology challenges to be conquered beyond today's method- and message-oriented middleware technologies. Infrastructure middleware provides an important building block for future R&D for the following reasons:

- Developers with a solid grasp of the design challenges and patterns underlying host infrastructure middleware can become proficient with software technology advances more rapidly. They can then catalyze the adoption of more sophisticated middleware capabilities within a team or organization.
- Developers with a thorough understanding of what happens "under the covers" of middleware are better suited to identify new ways of improving their networked applications.

0.4 An Overview of the ACE Toolkit

The *ADAPTIVE Communication Environment* (ACE) is a widely used example of host infrastructure middleware. The ACE library contains ~240,000 lines of C++ code and ~500 classes. The ACE software distribution also contains hundreds of automated regression tests and example applications. ACE is freely available as open-source software and can be downloaded from `http://ace.ece.uci.edu/` or `http://www.riverace.com`.

To separate concerns, reduce complexity, and permit functional subsetting, ACE is designed using a layered architecture [BMR+96], shown in Figure 0.5. The foundation of the ACE toolkit is its combination of OS adaptation *layer* and C++ wrapper facades [SSRB00], which encapsulate core OS concurrent network programming mechanisms. The higher layers of ACE build upon this foundation to provide reusable *frameworks*, networked service components, and standards-based middleware. Together, these middleware layers simplify the creation, composition, configuration, and porting of networked applications without incurring significant performance overhead.

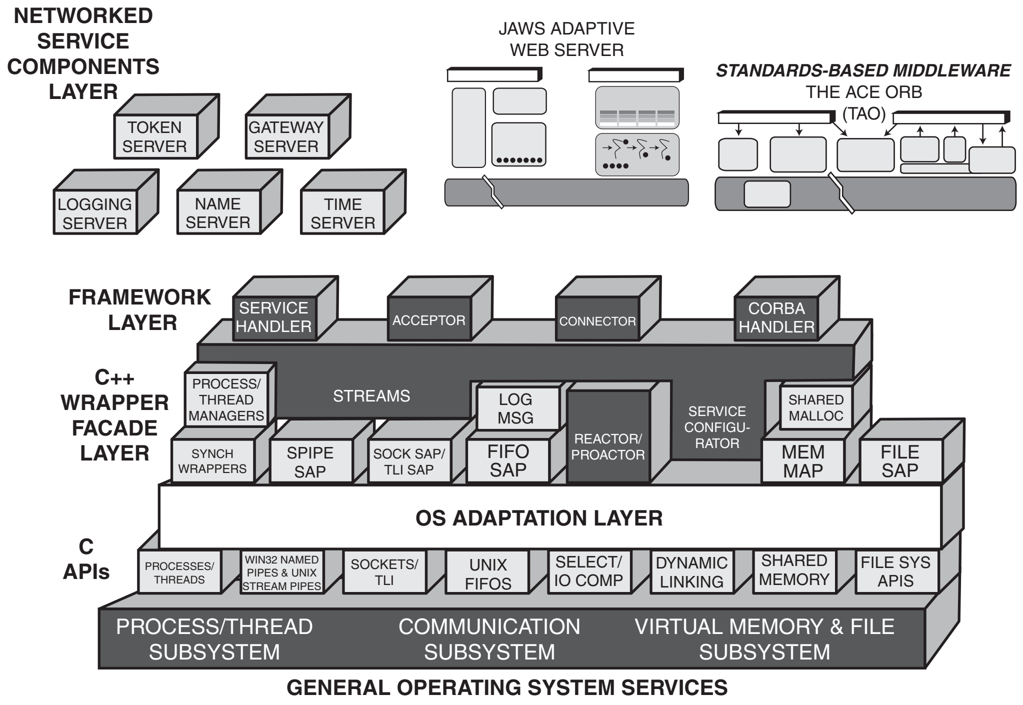


Figure 0.5: **The Layered Architecture of ACE**

This book focuses on the ACE wrapper facades for native OS IPC and concurrency mechanisms. The additional benefits of frameworks and a comprehensive description of the ACE frameworks are described in the second volume of *C++ Network Programming* [SH]. The remainder of this chapter outlines the structure and functionality of the various layers in ACE. Section B.1.4 on page 263 describes the standards-based middleware (TAO [SLM98] and JAWS [HS99]) that's based upon and bundled with ACE.

0.4.1 The ACE OS Adaptation Layer

The ACE OS adaptation layer constitutes approximately 10 percent of ACE (about 27,000 lines of code). It consists of a class called `ACE_OS` that contains over 500 C++ static methods. These methods encapsulate the native, C-oriented OS APIs that hide platform-specific details and expose a uniform interface to OS mechanisms used by higher ACE layers. The `ACE_OS` adaptation layer simplifies the portability and maintainability of ACE and ensures that only ACE developers—not applications developers—must understand the arcane platform-specific knowledge underlying the ACE wrapper facades. The abstraction provided by the `ACE_OS` class enables the use of a single source tree for all the OS platforms shown in Sidebar 1.

Sidebar 1: OS Platforms Supported by ACE

ACE runs on a wide range of operating systems, including:

- PCs, for example, Windows (all 32/64-bit versions), WinCE; Redhat, Debian, and SuSE Linux; and Macintosh OS X;
- Most versions of UNIX, for example, SunOS 4.x and Solaris, SGI IRIX, HP-UX, Digital UNIX (Compaq Tru64), AIX, DG/UX, SCO OpenServer, UnixWare, NetBSD, and FreeBSD;
- Real-time operating systems, for example, VxWorks, OS/9, Chorus, LynxOS, Pharlap TNT, QNX Neutrino and RTP, RTEMS, and pSoS;
- Large enterprise systems, for example, OpenVMS, MVS OpenEdition, Tandem NonStop-UX, and Cray UNICOS.

ACE can be used with all of the major C++ compilers on these platforms. The ACE Web site at `http://ace.ece.uci.edu` contains a complete, up-to-date list of platforms, along with instructions for downloading and building ACE.

0.4.2 The ACE C++ Wrapper Facade Layer

A wrapper facade consists of one or more classes that encapsulate functions and data within a type-safe object-oriented interface [SSRB00]. The ACE C++ wrapper facade layer resides atop its OS adaptation layer and provides largely the same functionality, as shown in Figure 0.5. Packaging this functionality as C++ classes, rather than stand-alone C functions, significantly reduces the effort required to learn and use ACE correctly. The ACE wrapper facades are designed carefully to minimize or eliminate performance overhead resulting from its increased usability and safety. The principles that guide ACE's design are discussed in Appendix A.

ACE provides an extensive set of wrapper facades, constituting nearly 50 percent of its total source base. Applications combine and refine these wrapper facades by selectively inheriting, aggregating, and/or instantiating them. In this book we show how the socket, file, concurrency, and synchronization wrapper facades are used to develop efficient, portable networked applications.

0.4.3 The ACE Framework Layer

The remaining ~40 percent of ACE consists of *object-oriented frameworks*, which are integrated sets of classes that collaborate to provide a reusable software architecture for a family of related applications [FS97]. Object-oriented frameworks are a key to successful systematic reuse because they complement and amplify other reuse techniques, such as class libraries, components, and patterns [Joh97]. By emphasizing the integration and collaboration of application-specific and application-independent classes, for example, the ACE frameworks enable larger-scale reuse of software than is possible by reusing individual classes or stand-alone functions. The frameworks in ACE integrate and augment its C++ wrapper facade classes by applying advanced concurrency and network programming patterns [BMR+96, SSRB00] to *reify* the canonical control flow and collaboration among families of related classes in ACE.

The following ACE frameworks support the efficient, robust, and flexible development and configuration of concurrent networked applications and services:

Event demultiplexing and dispatching frameworks. The ACE Reactor and Proactor frameworks implement the Reactor and Proactor patterns [SSRB00], respectively. The Reactor and Proactor frameworks automate the demultiplexing and dispatching of application-specific handlers in response to various types of I/O-based, timer-based, signal-based, and synchronization-based events.

Connection establishment and service initialization framework. The ACE Acceptor-Connector framework implements the *Acceptor-Connector pattern* [SSRB00]. This framework decouples the active and passive initialization roles from application processing performed by communicating peer services after initialization is complete.

Concurrency framework. ACE provides the Task framework that can be used to implement key concurrency patterns [SSRB00, Lea99], such as Active Object and Half-Sync/Half-Async, which simplify concurrent programming by decoupling method execution from method invocation and decoupling asynchronous and synchronous processing, respectively.

Service configurator framework. This framework implements the *Component Configurator pattern* [SSRB00] to support the configuration of applications whose services can be assembled dynamically late in their design cycle, for example, at installation time. It also supports the dynamic reconfiguration of services in an application at run time.

Streams framework. This framework implements the *Pipes and Filters pattern* [BMR$^+$96], wherein each processing step is encapsulated in a filtering module that can access and manipulate data flowing through the stream of modules. The ACE Streams framework simplifies the development of hierarchically layered services that can be composed flexibly to create certain types of networked applications, such as user-level protocol stacks and network management agents [SS94].

An in-depth discussion of the motivation, design, and use of the frameworks in ACE appears in *C++ Network Programming: Systematic Reuse with ACE and Frameworks* [SH]. Additional information on the ACE wrapper facades and frameworks is also available in *The ACE Programmer's Guide* [HJS].

0.4.4 The ACE Networked Service Components Layer

In addition to its host infrastructure middleware wrapper facades and frameworks previously described, ACE also provides a library of networked services that are packaged as components. A *component* is an encapsulated part of a software system that implements a specific service or set of services [Szy98]. Although these components aren't included in the ACE library itself, they are bundled with the ACE software distribution to provide the following capabilities:

- **Demonstrate common uses of ACE capabilities**—The components demonstrate how key ACE frameworks and classes can be used to develop flexible, efficient, and robust networked services.

- **Factor out reusable networked application building blocks**—These components provide reusable implementations of common networked application services, such as naming, event routing [Sch00], logging, time synchronization [SSRB00], and network locking.

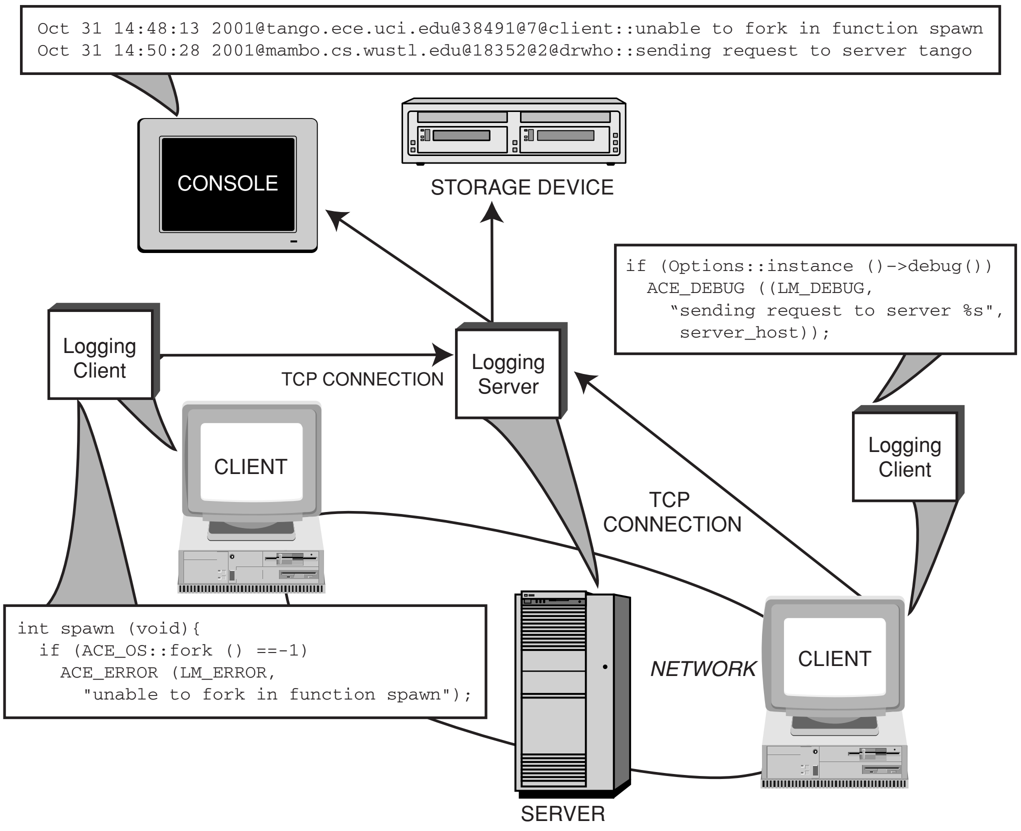
0.5 Example: A Networked Logging Service

Throughout this book we use a running example of a networked logging service to help illustrate key points and ACE capabilities. This service collects and records diagnostic information sent from one or more client applications. It's a departure from the usual way of logging to a Windows NT/2000 event log, which is not available on Windows 95 or 98. If you're an experienced UNIX programmer, however, you may be thinking this is a waste of time since SYSLOGD provides this type of service already. Yet this underscores a key benefit of the logging service: it's portable, so applications can log messages on all platforms that ACE supports.

The logging service example is a microcosm of the actual Logging Service in ACE. ACE's logging service can be configured dynamically via the Component Configurator pattern [SSRB00] and ACE Service Configurator framework [SH]. By applying the Adapter pattern [GHJV95], records can be redirected to a UNIX SYSLOGD or to the Windows NT/2000 event log, or both—even if the initiating application is on another type of OS platform. This book's logging service example is purposely scaled back so we can focus on mastering complexity. Figure 0.6 illustrates the application processes and server in our networked logging service. Below, we outline the key entities shown in Figure 0.6.

Client application processes run on client hosts and generate log records ranging from debugging messages to critical error messages. The logging information sent by a client application indicates the following:

1. The time the log record was created
2. The process identifier of the application
3. The priority level of the log record and
4. A string containing the logging message text, which can vary in size from 0 to a configurable maximum length, such as 4K bytes.



```
Oct 31 14:48:13 2001@tango.ece.uci.edu@38491@7@client::unable to fork in function spawn
Oct 31 14:50:28 2001@mambo.cs.wustl.edu@18352@2@drwho::sending request to server tango
```

```
if (Options::instance ()->debug())
  ACE_DEBUG ((LM_DEBUG,
    "sending request to server %s",
     server_host));
```

```
int spawn (void){
  if (ACE_OS::fork () ==-1)
    ACE_ERROR (LM_ERROR,
      "unable to fork in function spawn");
```

Figure 0.6: **Participants in the Networked Logging Service**

Logging servers collect and output log records received from client applications. A logging server can determine which client host sent each message by using addressing information it obtains from the Socket API. There's generally one logging server per system configuration, though they can be replicated to enhance fault tolerance.

Throughout the book, we refer to the networked logging service to make our discussion of domain analysis dimensions for networked applications more concrete. The architecture of our logging service is driven by this domain analysis. Just as real products change in scope as they progress through their life cycles, the logging service's design, functionality, scalability, and robustness will evolve as we progress through this book and [SH]. We'll continue developing this service incrementally to show solutions to common design challenges using many key patterns implemented by classes in ACE. Sidebar 2 describes how to build the ACE library so that you can experiment with the examples we present in this book.

Sidebar 2: Building ACE and Programs that Use ACE

ACE is open-source software, so you can download it from `http://ace.ece.uci.edu` and build it yourself. Here are some tips to help you understand the source examples we show, and how to build ACE, the examples, and your own applications:

- Install ACE in an empty directory. The top-level directory in the distribution is named `ACE_wrappers`. We refer to this top-level directory as `$ACE_ROOT`. Create an environment variable by that name containing the full path to the top-level ACE directory.
- The ACE source and header files reside in `$ACE_ROOT/ace`.
- The source and header files for this book's networked logging service examples reside in `$ACE_ROOT/examples/C++NPv1`.
- When compiling your programs, the `$ACE_ROOT` directory must be added to your compiler's file include path, which is often designated by the `-I` or `/I` compiler option.
- The `$ACE_ROOT/ACE-INSTALL.html` file contains complete instructions on building and installing ACE and programs that use ACE.

You can also purchase a prebuilt version of ACE from Riverace at a nominal cost. A list of the prebuilt compiler and OS platforms supported by Riverace is available at `http://www.riverace.com`.

0.6 Summary

This chapter described the challenges of developing networked applications and middleware that can run effectively in distributed computing environments. We introduced the inherent and accidental complexities encountered when developing software ranging from tightly constrained real-time and embedded systems [SLM98] to newly evolving middleware abstractions [MSKS00] and next-generation networked applications [SKKK00] with stringent QoS requirements. We presented a taxonomy of middleware layering, emphasizing the benefits of host infrastructure middleware, which is the focus of this book.

This chapter also introduced the results of a domain analysis of the key design dimensions for networked application architectures. These were grouped into four categories:

1. Communication protocols and mechanisms
2. Concurrency architectures
3. Service architectures and
4. Service configuration strategies.

This domain analysis has been refined while developing hundreds of networked applications and middleware components during the past decade. This analysis also guided the development of the ACE concurrent network programming toolkit. ACE exemplifies the principles and benefits gained through *refactoring* [FBB$^+$99] the recurring structure and behavior of networked applications into host infrastructure middleware. ACE's pattern-oriented software architecture constitutes an industrial-strength example of how proper object-oriented design and C++ usage can yield significant improvements in your development schedules and the quality, flexibility, and performance of your networked applications and middleware.

Finally, we introduced the networked logging service, which stores diagnostic information sent from one or more client applications. We use this example throughout the book to illustrate common design problems and their effective solutions using ACE. The next two parts of the book are organized as follows:

- **Part I**—Chapters 1 through 4 outline communication design alternatives and describe the object-oriented techniques used in ACE to programming OS IPC mechanisms effectively.
- **Part II**—Chapters 5 through 10 outline concurrency design alternatives and describe the object-oriented techniques used in ACE to program OS concurrency mechanisms effectively.

Throughout both parts of the book, we illustrate common problems that arise when developers design networked applications and when they program them using native OS IPC and concurrency APIs directly. We also show how ACE applies object-oriented design techniques, C++ features, and patterns to resolve these problems.

Index