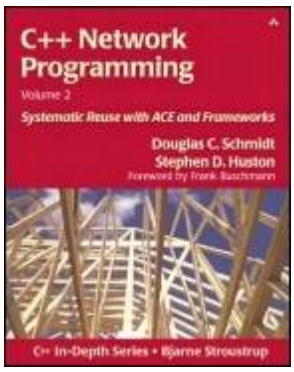


Table of Contents

C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks

By Douglas C. Schmidt, Stephen D. Huston

Publisher : Addison Wesley
Pub Date : October 29, 2002
ISBN : 0-201-79525-6
Pages : 384

Do you need to develop flexible software that can be customized quickly? Do you need to add the power and efficiency of frameworks to your software? The ADAPTIVE Communication Environment (ACE) is an open-source toolkit for building high-performance networked applications and next-generation middleware. ACE's power and flexibility arise from object-oriented frameworks, used to achieve the systematic reuse of networked application software. ACE frameworks handle common network programming tasks and can be customized using C++ language features to produce complete distributed applications.

C++ Network Programming, Volume 2, focuses on ACE frameworks, providing thorough coverage of the concepts, patterns, and usage rules that form their structure. This book is a practical guide to designing object-oriented frameworks and shows developers how to apply frameworks to concurrent networked applications. C++ Networking, Volume 1, introduced ACE and the wrapper facades, which are basic network computing ingredients. Volume 2 explains how frameworks build on wrapper facades to provide higher-level communication services.

Written by two experts in the ACE community, this book contains:

-
-
- An overview of ACE frameworks
-
- Design dimensions for networked services
-
- Descriptions of the key capabilities of the most important ACE frameworks

-
- Numerous C++ code examples that demonstrate how to use ACE frameworks
-

C++ Network Programming, Volume 2, teaches how to use frameworks to write networked applications quickly, reducing development effort and overhead. It will be an invaluable asset to any C++ developer working on networked applications.

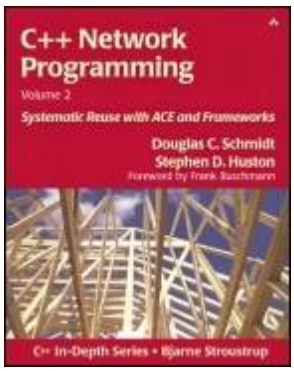


Table of Contents

C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks

By Douglas C. Schmidt, Stephen D. Huston

Publisher : Addison Wesley
Pub Date : October 29, 2002
ISBN : 0-201-79525-6
Pages : 384

Copyright
Foreword
About This Book
- Intended Audience
- Structure and Content
- Related Material
- Acknowledgments

Chapter 1. Object-Oriented Frameworks for Network Programming
- Section 1.1. An Overview of Object-Oriented Frameworks
- Section 1.2. Comparing Software Development and Reuse Techniques
- Section 1.3. Applying Frameworks to Network Programming
- Section 1.4. A Tour through the ACE Frameworks
- Section 1.5. Example: A Networked Logging Service
- Section 1.6. Summary

Chapter 2. Service and Configuration Design Dimensions
- Section 2.1. Service and Server Design Dimensions
- Section 2.2. Configuration Design Dimensions
- Section 2.3. Summary

Chapter 3. The ACE Reactor Framework
- Section 3.1. Overview
- Section 3.2. The ACE_Time_Value Class
- Section 3.3. The ACE_Event_Handler Class
- Section 3.4. The ACE Timer Queue Classes
- Section 3.5. The ACE_Reactor Class

Section 3.6. Summary

Chapter 4. ACE Reactor Implementations
Section 4.1. Overview
Section 4.2. The ACE_Select_Reactor Class
Section 4.3. The ACE_TP_Reactor Class
Section 4.4. The ACE_WFMO_Reactor Class
Section 4.5. Summary

Chapter 5. The ACE Service Configurator Framework
Section 5.1. Overview
Section 5.2. The ACE_Service_Object Class
Section 5.3. The ACE_Service_Repository Classes
Section 5.4. The ACE_Service_Config Class
Section 5.5. Summary

Chapter 6. The ACE Task Framework
Section 6.1. Overview
Section 6.2. The ACE_Message_Queue Class
Section 6.3. The ACE_Task_Class
Section 6.4. Summary

Chapter 7. The ACE Acceptor-Connector Framework
Section 7.1. Overview
Section 7.2. The ACE_Svc_Handler Class
Section 7.3. The ACE_Acceptor Class
Section 7.4. The ACE_Connector Class
Section 7.5. Summary

Chapter 8. The ACE Proactor Framework
Section 8.1. Overview
Section 8.2. The Asynchronous I/O Factory Classes
Section 8.3. The ACE_Handler Class
Section 8.4. The Proactive Acceptor-Connector Classes
Section 8.5. The ACE_Proactor Class
Section 8.6. Summary

Chapter 9. The ACE Streams Framework
Section 9.1. Overview
Section 9.2. The ACE_Module Class
Section 9.3. The ACE_Stream Class
Section 9.4. Summary

Glossary
Bibliography

Copyright

Library of Congress Cataloging-in-Publication Data

Schmidt, Douglas C.

C++ network programming / Douglas C. Schmidt, Stephen D. Huston.

p. cm.

Foreword

The ADAPTIVE Communication Environment (ACE) toolkit has achieved enormous success in the area of middleware for networked computing. Due to its flexibility, performance, platform coverage, and other key properties, ACE enjoys broad acceptance by the networked application software community, as evidenced by its use in thousands of applications, in scores of countries, and in dozens of domains. ACE has also received considerable attention beyond the middleware community since it's an open-source role model for high-quality and well-designed pattern-oriented software architectures.

But why is ACE so successful? Addressing this question properly takes some thought. To start off, let's reconsider the Foreword from C++ Network Programming: Mastering Complexity with ACE and Patterns (C++NPv1) and resume the mass transit analogy presented there by my colleague Steve Vinoski. Steve's right that a high-quality mass transit system consists of more than just aircraft, airports, trains, train stations, and rails. It also needs less obvious infrastructure, such as scheduling, routing, ticketing, maintenance, and monitoring. But even a complete collection of ingredients is still not sufficient to develop an effective mass transit system. Arranging these ingredients so they seamlessly fulfill their primary objective?ast and reliable transportation of people?s equally important. Would you use a mass transit system whose ticketing was located in a train maintenance location or an airport hangar, or whose planned and actual scheduling and routing weren't available to the public? I doubt it!

The success of mass transit systems depends on more than the knowledge of the infrastructure parts that are provided?t depends on how these different parts must be connected and integrated with their environment. This knowledge enables architects of mass transit systems to integrate individual parts into higher-level building blocks and to connect these building blocks effectively. For example, ticketing, information points, baggage offices, and boarding are integrated in train stations located at city centers or major suburban centers. Likewise, airports are often located near large cities and connected by frequent express trains.

Even mass transit centers themselves are arranged so that activities can be performed effectively. For example, when you enter a train station or airport via the main entrance, you find ticket agents, information centers, and timetables. You also find shops to satisfy your travel needs. As you enter the main train hall or airport concourse, you find other information centers, up-to-date scheduling information, and the platforms and gates for boarding the trains and planes. Mass transit centers thus not only provide all necessary services to begin and end a journey, they also organize their internal "control flows" effectively. While the core structures and control flows in most train stations and airports are similar, their concrete realization can differ widely. Yet we all recognize these mass transit center patterns immediately since they follow key invariants that we've learned through years of experience.

So what's the connection between successful mass transit system design and the success of ACE? The answer is simple: In addition to the basic network computing ingredients (the wrapper facades that Doug and Steve introduced in C++NPv1), ACE also includes useful object-oriented frameworks that build upon these wrapper facades and provide useful higher-level communication services, such as event demultiplexing and dispatching, connection management, service configuration, concurrency, and hierarchically layered stream processing. The ACE framework services satisfy many networked software needs by organizing the structures and internal control flows of your applications effectively via key patterns learned through years of experience.

The ACE frameworks offer you a number of important benefits:

About This Book

Software for networked applications must possess the following qualities to be successful in today's competitive, fast-paced computing industry:

-
- Affordability, to ensure that the total ownership costs of software acquisition and evolution are not prohibitively high
-
-
- Extensibility, to support successions of quick updates and additions to address new requirements and take advantage of emerging markets
-
-
- Flexibility, to support a growing range of multimedia data types, traffic patterns, and end-to-end quality of service (QoS) requirements
-
-
- Portability, to reduce the effort required to support applications on heterogeneous OS platforms and compilers
-
-
- Predictability and efficiency, to provide low latency to delay-sensitive real-time applications, high performance to bandwidth-intensive applications, and usability over low-bandwidth networks, such as wireless links
-
-
- Reliability, to ensure that applications are robust, fault tolerant, and highly available
-
-
- Scalability, to enable applications to handle large numbers of clients simultaneously
-

Writing high-quality networked applications that exhibit these qualities is hard?t's expensive, complicated, and error prone. The patterns, C++ language features, and object-oriented design principles presented in C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns (C++NPv1) help to minimize complexity and mistakes in networked applications by refactoring common structure and functionality into reusable wrapper facade class libraries. The key benefits of reuse will be lost, however, if large parts of the application software that uses these class libraries?r worse, the class libraries themselves?ust be rewritten for each new project.

Historically, many networked application software projects began by

7.

Intended Audience

This book is intended for "hands on" C++ developers or advanced students interested in understanding how to design object-oriented frameworks and apply them to develop networked applications. It builds upon material from C++NPv1 that shows how developers can apply patterns to master complexities arising from using native OS APIs to program networked applications. It's therefore important to have a solid grasp of the following topics covered in C++NPv1 before reading this book:

-
- Networked application design dimensions, including the alternative communication protocols and data transfer mechanisms discussed in Chapter 1 of C++NPv1
-
-
- Internet programming mechanisms, such as TCP/IP connection management and data transfer APIs [Ste98] discussed in Chapter 2 of C++NPv1
-
-
- Concurrency design dimensions, including the use of processes and threads, iterative versus concurrent versus reactive servers, and threading models [Ste99] discussed in Chapters 5 through 9 of C++NPv1
-
-
- Synchronization techniques necessary to coordinate the interactions of processes and threads on various OS platforms [KSS96, Lew95, Ric97] discussed in Chapter 10 of C++NPv1
-
-
- Object-oriented design and programming techniques [Boo94, Mey97] that can simplify OS APIs and avoid programming mistakes through the use of patterns, such as Wrapper Facade [POSA2] and Proxy [POSA1, GoF] discussed in Chapter 3 and Appendix A of C++NPv1
-

The ACE frameworks are highly flexible and powerful, due in large part to their use of C++ language features [Bja00]. You should therefore be familiar with C++ class inheritance and virtual functions (dynamic binding) as well as templates (parameterized types) and the mechanisms your compiler(s) offer to instantiate them. ACE provides a great deal of assistance in overcoming differences between C++ compilers. As always, however, you need to know the capabilities of your development tools and how to use them. Knowing your tools makes it easier to follow the source code examples in this book and to build and run them on your systems. Finally, as you read the examples in this book, keep in mind the points noted in Sidebar 7 (page 46) regarding UML diagrams and C++ code.

Structure and Content

Our C++NPv1 book addressed how to master certain complexities of developing networked applications, focusing on the use of ACE's wrapper facades to avoid problems with operating system APIs written in C. This book (which we call C++NPv2) elevates our focus to motivate and demystify the patterns, design techniques, and C++ features associated with developing and using the ACE frameworks. These frameworks help reduce the cost and improve the quality of networked applications by reifying proven software designs and patterns into frameworks that can be reused systematically across projects and enterprises. The ACE frameworks expand reuse technology far beyond what can be achieved by reusing individual classes or even class libraries.

This book presents numerous C++ applications to reinforce the design discussions by showing concrete examples of how to use the ACE frameworks. These examples provide step-by-step guidance that can help you apply key object-oriented techniques and patterns to your own networked applications. The book also shows how to enhance your design skills, focusing on the key concepts and principles that shape the design of successful object-oriented frameworks for networked applications and middleware.

The chapters in the book are organized as follows:

-
- Chapter 1 introduces the concept of an object-oriented framework and shows how frameworks differ from other reuse techniques, such as class libraries, components, patterns, and model-integrated computing. We then outline the frameworks in the ACE toolkit that are covered in subsequent chapters.
-
-

- Chapter 2 completes the domain analysis begun in C++NPv1, which covered the communication protocols and mechanisms, and the concurrency architectures used by networked applications. The focus in this book is on the service and configuration design dimensions that address key networked application properties, such as duration and structure, how networked services are identified, and the time at which they are bound together to form complete applications.
-
-

- Chapter 3 describes the design and use of the ACE Reactor framework, which implements the Reactor pattern [POSA2] to allow event-driven applications to demultiplex and dispatch service requests that are delivered to an application from one or more clients.
-
-

- Chapter 4 then describes the design and use of the most common implementations of the ACE_Reactor interface, which support a wide range of OS event demultiplexing mechanisms, including select(), WaitForMultipleObjects(), XtAppMainLoop(), and /dev/poll.
-
-

- Chapter 5 describes the design and use of the ACE Service Configurator framework. This framework implements the Component Configurator pattern [POSA2] to allow an application to link/unlink its component service implementations at run time without having to modify, recompile, or relink the application statically

Related Material

This book is based on ACE version 5.3, released in the fall of 2002. ACE 5.3 and all the sample applications described in our books are open-source software. Sidebar 3 (page 19) explains how you can obtain a copy of ACE so you can follow along, see the actual ACE classes and frameworks in complete detail, and run the code examples interactively as you read the book.

To learn more about ACE, or to report errors you find in the book, we recommend you subscribe to the ACE mailing list, ace-users@cs.wustl.edu. You can subscribe by sending a request to ace-users-request@cs.wustl.edu. Include the following command in the body of the e-mail (the subject is ignored):

```
subscribe ace-users [emailaddress@domain]
```

You must supply emailaddress@domain only if your message's From address is not the address you wish to subscribe. If you use this alternate address method, the list server will require an extra authorization step before allowing you to join the list.

Postings to the ace-users list are also forwarded to the comp.soft-sys.ace USENET newsgroup, along with postings to several other ACE-related mailing lists. Reading the messages via the newsgroup is a good way to keep up with ACE news and activity if you don't require immediate delivery of the 30 to 50 messages that are posted daily on the mailing lists.

Archives of postings to the comp.soft-sys.ace newsgroup are available at http://groups.google.com/. Enter comp.soft-sys.ace in the search box to go to a list of archived messages. Google has a complete, searchable archive of over 40,000 messages. You can also post a message to the newsgroup from Google's site.

Acknowledgments

Champion reviewing honors go to Alain Decamps, Don Hinton, Alexander Maack, Chris Uzdavinis, and Johnny Willemsen, who reviewed the book multiple times and provided extensive, detailed comments that improved its form and content substantially. Many thanks also to the official reviewers, Timothy Culp, Dennis Mancl, Phil Mesnier, and Jason Pasion, who read the entire book and gave us many helpful comments. Many other ACE users provided feedback on this book, including Marc M. Adkins, Tomer Amiaz, Vi Thuan Banh, Kevin Bailey, Stephane Bastien, John Dilley, Eric Eide, Andrew Finnell, Dave Findlay, Jody Hagins, Jon Harnish, Jim Havlicek, Martin J3ohnson, Christopher Kohlhoff, Alex Libman, Harald Mitterhofer, Llori Patterson, Nick Pratt, Dieter Quehl, Tim Rozmajzl, Irma Rastegayeva, Eamonn Saunders, Harvinder Sawhney, Christian Schuhegger, Michael Searles, Kalvinder Singh, Henny Sipma, Stephen Sturtevant, Leo Stutzmann, Tommy Svensson, Bruce Trask, Dominic Williams, and Vadim Zaliva.

We are deeply indebted to all the members, past and present, of the DOC groups at Washington University in St. Louis and the University of California, Irvine, as well as the team members at Riverace Corporation and Object Computing Inc., who developed, refined, and optimized many of the ACE capabilities presented in this book. This group includes Everett Anderson, Alex Arulanthu, Shawn Atkins, John Aughey, Luther Baker, Jaiganesh Balasubramanian, Darrell Brunsch, Don Busch, Chris Cleeland, Angelo Corsaro, Chad Elliot, Sergio Flores-Gaitan, Chris Gill, Pradeep Gore, Andy Gokhale, Priyanka Gontla, Myrna Harbibson, Tim Harrison, Shawn Hannan, John Heitmann, Joe Hoffert, James Hu, Frank Hunleth, Prashant Jain, Vishal Kachroo, Ray Klefstad, Kitty Krishnakumar, Yamuna Krishnamurthy, Michael Kircher, Fred Kuhns, David Levine, Chanaka Liyanaarachchi, Michael Moran, Ebrahim Moshiri, Sumedh Mungee, Bala Natarajan, Ossama Othman, Jeff Parsons, Kirthika Parameswaran, Krish Pathayapura, Irfan Pyarali, Sumita Rao, Carlos O'Ryan, Rich Siebel, Malcolm Spence, Marina Spivak, Naga Surendran, Steve Totten, Bruce Trask, Nanbor Wang, and Seth Widoff.

We also want to thank the thousands of C++ developers from over 50 countries who've contributed to ACE for over a decade. ACE's excellence and success is a testament to the skills and generosity of many talented developers and the forward-looking companies that had the vision to contribute their work to ACE's open-source code base. Without their support, constant feedback, and encouragement, we would never have written this book. In recognition of the efforts of the ACE open-source community, we maintain a list of all contributors at http://ace.ece.uci.edu/ACE-members.html.

We are also grateful for the support from colleagues and sponsors of our research on patterns and development of the ACE toolkit, notably the contributions of Ron Akers (Motorola), Steve Bachinsky (SAIC), John Bay (DARPA), Detlef Becker (Siemens), Frank Buschmann (Siemens), Dave Busigo (DARPA), John Buttitto (Sun), Becky Callison (Boeing), Wei Chiang (Nokia Inc.), Joe Cross (Lockheed Martin), Lou DiPalma (Raytheon), Bryan Doerr (Savvis), Karlheinz Dorn (Siemens), Scott Ellard (Madison), Matt Emerson (Escient Convergence Group, Inc.), Sylvester Fernandez (Lockheed Martin), Nikki Ford (DARPA), Andreas Geisler (Siemens), Helen Gill (NSF), Inc.), Jody Hagins (ATD), Andy Harvey (Cisco), Sue Kelly (Sandia National Labs), Gary Koob (DARPA), Petri Koskelainen (Nokia Inc.), Sean Landis (Motorola), Patrick Lardieri (Lockheed Martin), Doug Lea (SUNY Oswego), Joe Loyall (BBN), Kent Madsen (EO Thorpe), Ed Margand (DARPA), Mike Masters (NSWC), Major Ed Mays (U.S. Marine Corps), John Mellby (Raytheon), Jeanette Milos (DARPA), Stan Moyer (Telcordia), Ivan Murphy (Siemens), Russ Noseworthy (Object Sciences), Adam Porter (U. of Maryland), Dieter Quehl (Siemens), Vijay Raghavan (Vanderbilt U.), Lucie Robillard (U.S. Air Force), Craig Rodrigues (BBN), Rick Schantz (BBN), Andreas Schulke (Siemens), Steve Shaffer (Kodak), Tom Shields (Raytheon), Dave Sharp (Boeing), Naval Sodha (Ericsson), Paul Stephenson (Ericsson), Tatsuya Suda (UCI), Umar Syyid (Storetrax, Inc.), Janos Sztipanovits (Vanderbilt U.), Gautam Thaker (Lockheed Martin), Lothar Werzinger (Krones), and Don Winter (Boeing).

Chapter 1. Object-Oriented Frameworks for Network Programming

CHAPTER SYNOPSIS

Object-oriented frameworks help reduce the cost and improve the quality of networked applications by reifying software designs and pattern languages that have proven effective in particular application domains. This chapter illustrates what frameworks are and compares them with other popular software development techniques, such as class libraries, components, patterns, and model-integrated computing. It then illustrates the process of applying frameworks to networked applications and outlines the ACE frameworks that are the focus of this book. These frameworks are based on a pattern language [POSA1, POSA2] that has been applied to thousands of production networked applications and middleware worldwide.

1.1 An Overview of Object-Oriented Frameworks

Even as computing power and network bandwidth increase dramatically, the development of networked application software remains expensive, time consuming, and error prone. The cost and effort stems from the growing demands placed on networked software, as well as the continual rediscovery and reinvention of core software design and implementation artifacts throughout the software industry. Moreover, the heterogeneity of hardware architectures, diversity of OS and network platforms, and stiff global competition makes it increasingly hard to build high-quality networked application software from scratch.

The key to building high-quality networked software in a time-to-market-driven environment is the ability to reuse successful software designs and implementations that have already been developed. Reuse has been a popular topic of debate and discussion for over 30 years in the software community [McI68]. There are two general types of reuse:

-
- Opportunistic reuse, in which developers cut and paste code from existing programs to create new ones. Opportunistic reuse works in a limited way for individual programmers or small groups. It doesn't scale up across business units or enterprises, however, and therefore doesn't significantly reduce development cycle time and cost or improve software quality. Worse, opportunistic reuse can actually impede development progress since cut-and-paste code often begins to diverge as it proliferates, forcing developers to fix the same bugs multiple times in multiple places.
-
-
- Systematic reuse, which is an intentional and concerted effort to create and apply multiuse software architectures, patterns, frameworks, and components throughout a product line [CN02]. In a well-honed systematic reuse process, each new project leverages time-proven designs and implementations, only adding new code that's specific to a particular application. This type of reuse is essential to increase software productivity and quality by breaking the costly cycle of rediscovering, reinventing, and revalidating common software artifacts.
-

Middleware [SS02] is a class of software that can increase systematic reuse levels significantly by functionally bridging the gap between the end-to-end functional requirements of networked applications and the underlying operating systems and network protocol stacks. Middleware provides capabilities that are critical to networked applications because they automate common network programming tasks. Developers who use middleware can therefore program their networked applications more like stand-alone applications, rather than wrestling with the many tedious and error-prone details associated with low-level OS event demultiplexing, message buffering and queueing, marshaling and demarshaling, and connection management mechanisms. Popular examples of middleware include Java virtual machines (JVMs), Enterprise JavaBeans (EJB), .NET, the Common Object Request Broker Architecture (CORBA), and the ADAPTIVE Communication Environment (ACE).

Systematically developing high-quality, reusable middleware for networked applications presents many hard technical challenges, including

-
- Detecting and recovering from transient and partial failures of networks and hosts in an application-independent manner
-

1.2 Comparing Software Development and Reuse Techniques

Object-oriented frameworks don't exist in isolation. Class libraries, components, patterns, and model-integrated computing are other techniques that are being applied to reuse software and increase productivity. This section compares frameworks with these techniques to illustrate their similarities and differences, as well as to show how the techniques can be combined to enhance systematic reuse for networked applications.

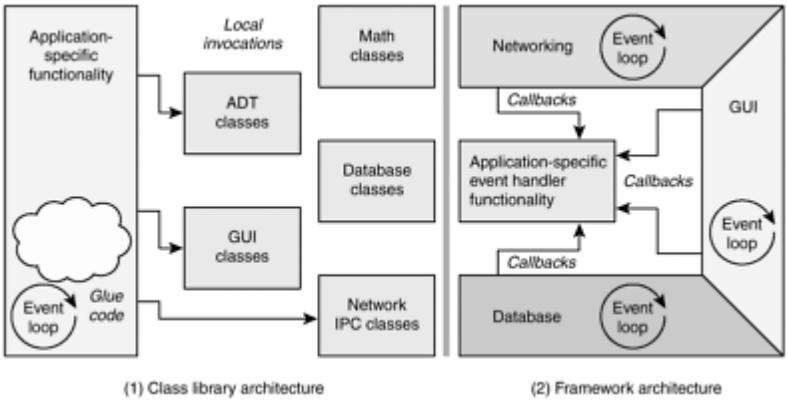
1.2.1 Comparing Frameworks and Class Libraries

A class is a general-purpose, reusable building block that specifies an interface and encapsulates the representation of its internal data and the functionality of its instances. A library of classes was the most common first-generation object-oriented development technique [Mey97]. Class libraries generally support reuse-in-the-small more effectively than function libraries since classes emphasize the cohesion of data and methods that operate on the data.

Although class libraries are often domain independent and can be applied widely, their effective scope of reuse is limited because they don't capture the canonical control flow, collaboration, and variability among families of related software artifacts. The total amount of reuse with class libraries is therefore relatively small, compared with the amount of application-defined code that must be rewritten for each application. The need to reinvent and reimplement the overall software architecture and much of the control logic for each new application is a prime source of cost and delay for many software projects.

The C++ standard library [Bja00] is a good case in point. It provides classes for strings, vectors, and other containers. Although these classes can be reused in many application domains, they are relatively low level. Application developers are therefore responsible for (re)writing much of the "glue code" that performs the bulk of the application control flow and class integration logic, as shown in Figure 1.2 (1).

Figure 1.2. Class Library versus Framework Architectures



Frameworks are a second-generation development technique [Joh97] that extends the benefits of class libraries in several ways. Most importantly, classes in a framework collaborate to provide a reusable architecture for a family of related applications. Class collaboration in a framework yields "semi-complete" applications that embody domain-specific object structures and functionality. Frameworks can be classified by various means, such as the blackbox and whitebox distinctions described in Sidebar 1 (page 6).

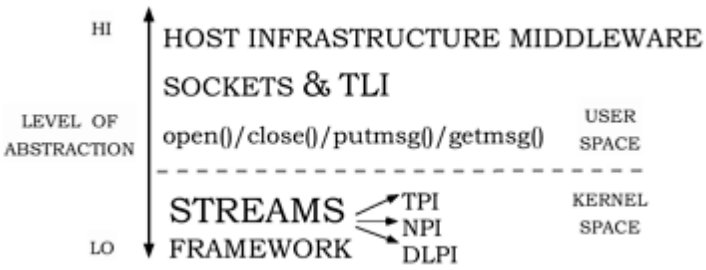
1.3 Applying Frameworks to Network Programming

One reason why it's hard to write robust, extensible, and efficient networked applications is that developers must master many complex networking programming concepts and mechanisms, including

-
- Network addressing and service identification/discovery
-
-
- Presentation layer conversions, such as marshaling, demarshaling, and encryption, to handle heterogeneous hosts with alternative processor byte orderings
-
-
- Local and remote interprocess communication (IPC) mechanisms
-
-
- Event demultiplexing and event handler dispatching
-
-
- Process/thread lifetime management and synchronization
-

Application programming interfaces (APIs) and tools have evolved over the years to simplify the development of networked applications and middleware. Figure 1.6 illustrates the IPC APIs available on OS platforms ranging from UNIX to many real-time operating systems. This figure shows how applications can access networking APIs for local and remote IPC at several levels of abstraction. We briefly discuss each level of abstraction below, starting from the lower-level kernel APIs to the native OS user-level networking APIs and the host infrastructure middleware.

Figure 1.6. Levels of Abstraction for Network Programming



Kernel-level networking APIs. Lower-level networking APIs are available in an OS kernel's I/O subsystem. For example, the UNIX putmsg() and getmsg() system functions can be used to access the Transport Provider Interface (TPI) [OSI92b] and the Data Link Provider Interface (DLPI) [OSI92a] available in System V STREAMS [Rit84]. It's also possible to develop network services, such as routers [KMC+00], network file systems [WLS+85], or even Web servers [JKN+01], that reside entirely within an OS kernel.

Programming directly to kernel-level networking APIs is rarely portable between different OS platforms, however.

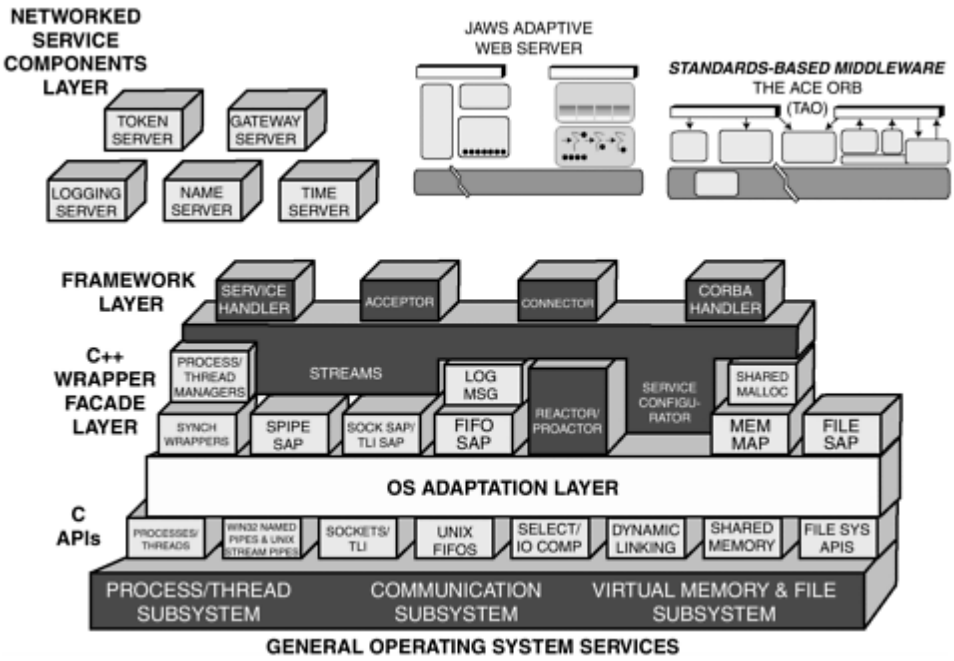
1.4 A Tour through the ACE Frameworks

1.4.1 An Overview of ACE

ACE is a highly portable, widely used, open-source host infrastructure middleware toolkit. The source code is freely available from http://ace.ece.uci.edu/ or http://www.riverace.com/. The core ACE library contains roughly a quarter million lines of C++ code that comprises approximately 500 classes. Many of these classes cooperate to form ACE's major frameworks. The ACE toolkit also includes higher-level components, as well as a large set of examples and an extensive automated regression test suite.

To separate concerns, reduce complexity, and permit functional subsetting, ACE is designed using a layered architecture [POSA1], shown in Figure 1.7. The capabilities provided by ACE span the session, presentation, and application layers in the OSI reference model [Bla91]. The foundation of the ACE toolkit is its combination of an OS adaptation layer and C++ wrapper facades, which together encapsulate core OS network programming mechanisms to run portably on all the OS platforms shown in Sidebar 2 (page 16). The higher layers of ACE build on this foundation to provide reusable frameworks, networked service components, and standards-based middleware.

Figure 1.7. The Layered Architecture of ACE



1.4.2 A Synopsis of the ACE Frameworks

The ACE frameworks are an integrated set of classes that can be instantiated and customized to provide complete networked applications and service components. These frameworks help to transfer decades of accumulated knowledge directly from the ACE developers to ACE users in the form of expertise embodied in well-tested and reusable C++ software artifacts. The ACE frameworks implement a pattern language for programming concurrent object-oriented networked applications. Figure 1.8 illustrates the ACE frameworks. To illustrate how the ACE frameworks rely on and use each other, the lines between boxes represent a dependency in the direction of the arrow. Each framework is outlined below.

Figure 1.8. The Key Frameworks in ACE

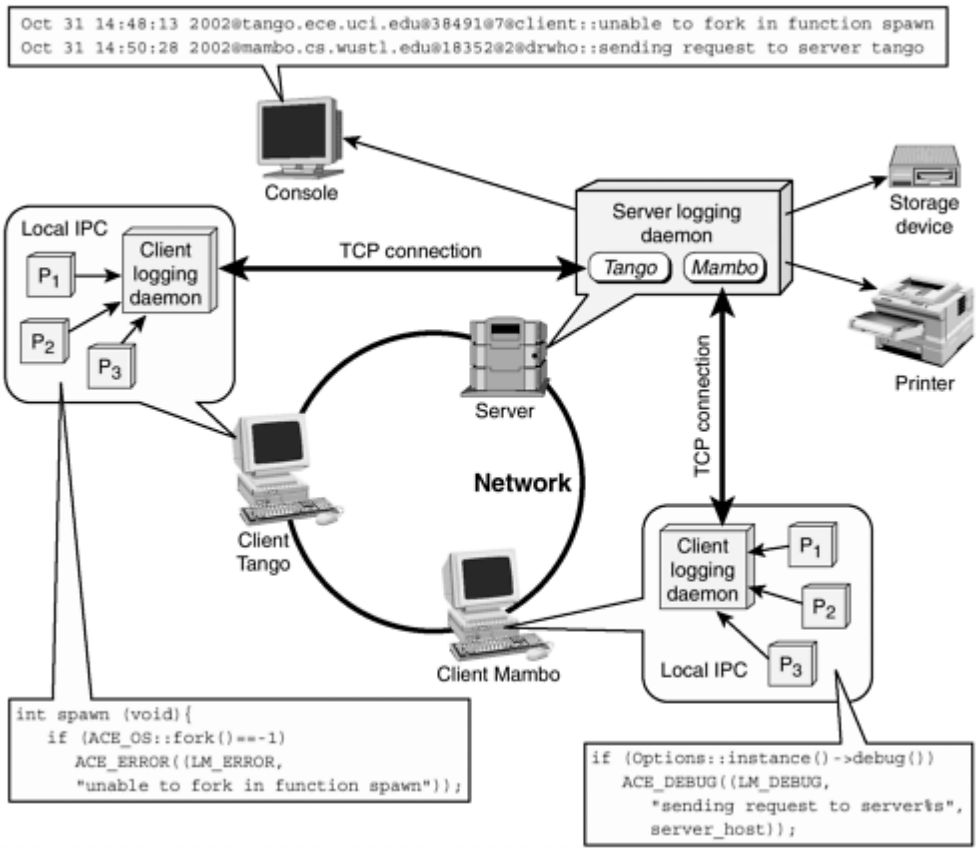
1.5 Example: A Networked Logging Service

It's been our experience that the principles, methods, and skills required to develop and use reusable networked application software cannot be learned solely by generalities or toy examples. Instead, programmers must learn concrete technical skills and gain hands-on experience by developing and using real frameworks and applications. We therefore illustrate key points and ACE capabilities throughout this book by extending and enhancing the networked logging service example introduced in C++NPv1, which collects and records diagnostic information sent from one or more client applications.

The logging service in C++NPv1 used many of ACE's wrapper facades in a two-tier client/server architecture. This book's logging service examples use a more powerful architecture that illustrates a broader complement of capabilities and patterns, and demonstrates how ACE's frameworks can help achieve efficient, predictable, and scalable networked applications. This service also helps to demonstrate key design and implementation considerations and solutions that will arise when you develop your own concurrent object-oriented networked applications.

Figure 1.10 illustrates the application processes and daemons in our networked logging service, which we outline below.

Figure 1.10. Processes and Daemons in the Networked Logging Service



Client application processes (such as P1,P2, and P3) run on client hosts and generate log records ranging from debugging messages to critical error messages. The logging information sent by a client application contains the time the log record was created, the process identifier of the application, the priority level of the log record, and a variable-sized string containing the log record text message. Client applications send these log records to a client logging daemon running on their local host.

1.6 Summary

Networked application software has been developed manually from scratch for decades. The continual rediscovery and reinvention of core concepts and capabilities associated with this process has kept the costs of engineering and evolving networked applications too high for too long. Improving the quality and quantity of systematic software reuse is essential to resolve this problem.

Middleware is a class of software that's particularly effective at providing systematically reusable artifacts for networked applications. Developing and using middleware is therefore an important way to increase reuse. There are many technical and nontechnical challenges that make middleware development and reuse hard, however. This chapter described how object-oriented frameworks can be applied to overcome many of these chal-lenges. To make the most appropriate choice of software development technologies, we also described the differences between frameworks and class libraries, components, patterns, and model-integrated computing. Each technology plays a part in reducing software development costs and life cycles and increasing software quality, functionality, and performance.

The result of applying framework development principles and patterns to the domain of networked applications has yielded the ACE frameworks. These frameworks handle common network programming tasks and can be customized via C++ language features to produce complete networked applications. When used together, the ACE frameworks simplify the creation, composition, configuration, and porting of networked applications without incurring significant performance overhead. The rest of this book explains how and why the ACE frameworks were developed and shows many examples of how ACE uses C++ features to achieve its goals.

An intangible, but valuable, benefit of ACE is its transfer of decades of accumulated knowledge from ACE framework developers to ACE framework users in the form of expertise embodied in well-tested C++ classes that implement time-proven networked application software development strategies. These frameworks took scores of person-years to develop, optimize, and mature. Fortunately, you can take advantage of the expertise embodied in these frameworks without having to independently rediscover or reinvent the patterns and classes that underlie them.

Chapter 2. Service and Configuration Design Dimensions

CHAPTER SYNOPSIS

A service is a set of functionality offered to a client by a server. Common services available on the Internet today include

- Web content retrieval services, such as Apache and Google
- Software distribution services, such as Castanet, Citrix, or Softricity
- Electronic mail and network news transfer services
- File access on remote machines
- Network time synchronization
- Payment processing services
- Streaming audio/video services, such as RealPlayer and QuickTime

Networked applications can be created by configuring their constituent services together at various points of time, such as compile time, static link time, installation time, or run time.

Chapters 1 and 5 of C++NPv1 provided a domain analysis of the communication protocols and mechanisms and the concurrency architectures used by networked applications. This chapter expands that coverage to analyze other design dimensions that address key networked application properties. These properties include service duration and structure, how networked services are identified, and the time at which they are bound together to form complete applications. These design dimensions are important in any networked application, and of particular importance to the ACE Service Configurator framework (Chapter 5). If you're already familiar with these design dimensions, however, you may want to skip ahead to Chapter 3, which begins the coverage of the ACE frameworks.

2.1 Service and Server Design Dimensions

When designing networked applications, it's important to recognize the difference between a service, which is a capability offered to clients, and a server, which is the mechanism by which the service is offered. The design decisions regarding services and servers are easily confused, but should be considered separately. This section covers the following service and server design dimensions:

-
- Short- versus long-duration services
-
-
- Internal versus external services
-
-
- Stateful versus stateless services
-
-
- Layered/modular versus monolithic services
-
-
- Single- versus multiservice servers
-
-
- One-shot versus standing servers
-

2.1.1 Short-Duration versus Long-Duration Services

The services offered by network servers can be classified as short duration or long duration. These time durations reflect how long a service holds system resources. The primary tradeoff in this design dimension involves holding system resources when they may be better used elsewhere versus the overhead of restarting a service when it's needed. In a networked application, this dimension is closely related to protocol selection because setup requirements for different protocols can vary significantly.

Short-duration services execute in brief, often fixed, amounts of time and usually handle a single request at a time. Examples of short-duration services include computing the current time of day, resolving the Ethernet number of an IP address, and retrieving a disk block from the cache of a network file server. To minimize the amount of time spent setting up a connection, short-duration services are often implemented using connectionless protocols, such as UDP/IP [Ste94].

Long-duration services run for extended, often variable, lengths of time and may handle numerous requests during their lifetime. Examples of long-duration services include transferring large software releases via FTP, downloading

2.2 Configuration Design Dimensions

This section covers the following configuration design dimensions:

-
- Static versus dynamic naming
-
-
- Static versus dynamic linking
-
-
- Static versus dynamic configuration
-

2.2.1 Static versus Dynamic Naming

Applications can be categorized according to whether their services are named statically or dynamically. The primary tradeoff in this dimension involves run-time efficiency versus flexibility.

Statically named services associate the name of a service with object code that exists at compile time and/or static link time. For example, INETD's internal services, such as ECHO and DAYTIME, are bound to statically named functions stored internally in the INETD program. A statically named service can be implemented in either static or dynamic libraries.

Dynamically named services defer the association of a service name with the object code that implements the service. Code therefore needn't be identified?or even be written, compiled, and linked?ntil an application begins executing the corresponding service at run time. A common example of dynamic naming is demonstrated by INETD's handling of TELNET, which is an external service. External services can be updated by modifying the inetd.conf configuration file and sending the SIGHUP signal to the INETD process. When INETD receives this signal, it rereads inetd.conf and dynamically rebinds the services it offers to their new executables.

2.2.2 Static versus Dynamic Linking

Applications can also be categorized according to whether their services are linked into a process address space statically or dynamically. The primary tradeoffs in this dimension involve extensibility, security, reliability, and efficiency.

Static linking creates a complete executable program by binding together all its object files at compile time and/or static link time, as shown in Figure 2.6 (1).

Figure 2.6. Static Linking versus Dynamic Linking

2.3 Summary

This chapter described two groups of design dimensions related to the successful development and deployment of networked applications. Service design dimensions affect the ways in which application services are structured, developed, and instantiated. Service configuration dimensions affect user or administrator abilities to vary the run-time placement and configuration of networked services after delivery and deployment.

Service design dimensions have a significant impact on how effectively applications use system and network resources. Efficient resource usage is closely linked to application response time, as well as to overall system performance and scalability. Performance is an important factor that's visible to end users. Though a coherent and modular design is less visible to end users, it's critical to a product's long-term success.

Good design simplifies maintenance and allows application functionality to evolve in response to market changes and competitive pressures without losing quality or performance. Fortunately, performance and modularity needn't be an either/or proposition. By carefully considering service design dimensions and applying ACE judiciously, you'll be able to create highly efficient and well-designed networked applications.

Even well-designed services and applications may need to adapt to a variety of deployment environments and user demands. Service configuration dimensions involve tradeoffs between design decisions associated with identifying a particular set of services and linking these services into the address space of one or more applications. To produce successful solutions, a networked application's flexibility must be weighed against its security, packaging, and complexity concerns.

When developing networked applications, the two sets of design dimensions in this chapter should be considered along with the dimensions described in Chapters 1 and 5 of C++NPv1. The ACE frameworks described in this book offer powerful tools to implement flexible and extensible designs with many combinations of tradeoffs and capabilities.

Chapter 3. The ACE Reactor Framework

CHAPTER SYNOPSIS

This chapter describes the design and use of the ACE Reactor framework. This framework implements the Reactor pattern [POSA2], which allows event-driven applications to react to events originating from a number of disparate sources, such as I/O handles, timers, and signals. Applications override framework-defined hook methods, which the framework then dispatch to process events. We show how to implement a logging server using a reactor that (1) detects and demultiplexes different types of connection and data events from various event sources and (2) then dispatches the events to application-defined handlers that process the events.

3.1 Overview

The ACE Reactor framework simplifies the development of event-driven programs, which characterize many networked applications. Common sources of events in these applications include activity on an IPC stream for I/O operations, POSIX signals, Windows handle signaling, and timer expirations. In this context, the ACE Reactor framework is responsible for

-
- Detecting the occurrence of events from various event sources
-
-
- Demultiplexing the events to their preregistered event handlers
-
-
- Dispatching to hook methods defined by the handlers to process the events in an application-defined manner
-

This chapter describes the following ACE Reactor framework classes that networked applications can use to detect the occurrence of events and then demultiplex and dispatch the events to their event handlers:

ACE Class	**Description**
ACE_Time_Value	Provides a portable, normalized representation of time and duration that uses C++ operator overloading to simplify time-related arithmetic and relational operations.
ACE_Event_Handler	An abstract class whose interface defines the hook methods that are the target of ACE_Reactor callbacks. Most application event handlers developed with ACE are descendants of ACE_Event_Handler.
ACE_Timer_Queue	An abstract class defining the capabilities and interface for a timer queue. ACE contains a variety of classes derived from ACE_Timer_Queue that provide flexible support for different timing requirements.
ACE_Reactor	Provides the interface for managing event handler registrations and executing the event loop that drives event detection, demultiplexing, and dispatching in the Reactor framework.

3.2 The ACE_Time_Value Class

Motivation

Different operating systems provide different functions and data to access and manipulate the time and date. For example, UNIX platforms define the timeval structure as follows:

```
struct timeval {
  long secs;
  long usecs;
};
```

Different date and time representations are used on other OS platforms, such as POSIX, Windows, and proprietary real-time operating systems. Time values are used in a number of situations, including timeout specifiers. As described in Sidebar 6 (page 45), ACE specifies timeouts in absolute time for some situations, such as the concurrency and synchronization wrapper facades in C++NPv1, and in relative time for other situations, such as the ACE_Reactor I/O timeouts and timer settings. The wide range of uses and different representations across platforms makes addressing these portability differences in each application unnecessarily tedious and costly, which is why the ACE Reactor framework provides the ACE_Time_Value class.

Class Capabilities

ACE_Time_Value applies the Wrapper Facade pattern [POSA2] and C++ operator overloading to simplify the use of portable time and duration related operations. This class provides the following capabilities:

-
- It provides a standardized representation of time that's portable across OS platforms.
-
-
- It can convert between different platform time representations, such as timespec_t and timeval on UNIX, and FILETIME and timeval on Windows.
-
-
- It uses operator overloading to simplify time-based comparisons by permitting standard C++ syntax for time-based arithmetic and relational expressions.
-
-
- Its constructors and methods normalize time quantities by converting the fields in a timeval structure into a canonical format that ensures accurate comparisons between ACE_Time_Value instances.
-
-
- It can represent either a duration, such as 5 seconds and 310,000 microseconds, or an absolute date and time, such as 2001-09-11-08.46.00. Note that some methods, such as operator*=(), are meaningless with absolute times.
-

3.3 The ACE_Event_Handler Class

Motivation

Networked applications are often based on a reactive model, in which they respond to various types of events, such as I/O activity, expired timers, or signals. Application-independent mechanisms that detect events and dispatch them to event-handling code should be reused across applications, while application-defined code that responds to the events should reside in the event handlers. To reduce coupling and increase reuse, a framework separates the reuseable mechanisms and provides the means to incorporate application-defined event handlers. This separation of concerns is the basis of the ACE Reactor framework's inversion of control. Its event detection and dispatching mechanisms control execution flow and invoke event-handling callback hook methods when there's application processing to perform.

Sidebar 8: The ACE_Get_Opt Class

ACE_Get_Opt is an iterator for parsing options from command-line arguments. Options passed in an optstring are preceded by '-' for short options or '--' for long options. ACE_Get_Opt can be used to parse argc/argv arguments, such as those passed as a program's main() command line or to an init() hook method. This class provides the following capabilities:

-
- A thin C++ wrapper facade for the standard POSIX getopt() function. Unlike getopt(), however, each instance of ACE_Get_Opt maintains its own state, so it can be used reentrantly. ACE_Get_Opt is also easier to use than getopt() since the optstring and argc/argv arguments are only passed once to its constructor, rather than to each iterator call.
-
-

- It can be told to start processing the command line at an arbitrary point specified by the skip_args parameter, which allows it to skip the program name when parsing a command line passed to main() or continue processing where it left off at a later time.
-
-

- It can regroup all the option arguments at the beginning of the command line, while maintaining their relative order, which simplifies option and nonoption argument processing. After all the options are scanned, it returns EOF and opt_ind() points to the first nonoption argument, so the program can continue processing the remaining arguments.
-
-

- Multiple argument ordering modes: PERMUTE_ARGS, REQUIRE_ORDER, and RETURN_IN_ORDER.
-
-

3.4 The ACE Timer Queue Classes

Motivation

Many networked applications perform activities periodically or must be notified when specified time periods have elapsed. For example, Web servers require watchdog timers that release resources if clients don't send an HTTP GET request shortly after they connect.

Native OS timer capabilities vary, but many platforms share the following problems:

-
- Limited number of timers. Many platforms allow applications to set a limited number of timers. For example, the POSIX alarm() and ualarm() system functions each reset a single "alarm clock" timer on each call. Managing multiple timer periods therefore often involves developing a timer queue mechanism that keeps track of the next scheduled expiration. Scheduling a new "earliest" timer can (re)set the alarm clock if necessary.
-
-
- Timer expiration raises a signal. For example, the alarm() system function raises the SIGALRM signal when the timer expires. Programming timer signals is hard because application actions are restricted in signal context. Applications can minimize processing in signal context on UNIX platforms by using the sleep() system function or using the sigsuspend() system function. These solutions are nonportable, however, and they block the calling thread, which can impede concurrency and complicate programming.
-

One way to avoid these problems is to manage timers in the normal course of event handling, as follows:

1.
1. Develop a timer queue mechanism that orders timers and associates each timer with an action to perform when a timer expires
1.
2.
2. Integrate the timer queue with the application's use of a synchronous event demultiplexer, such as select() or WaitForMultipleObjects(), to integrate the handling of timer expirations with other event processing.
2.

It's hard to develop this type of timer facility portably across OS platforms, however, due to the wide range of capabilities and restrictions. Moreover, this capability is often redeveloped for many projects due to tight coupling between the timer queue mechanism and the synchronous event demultiplexing mechanism. To alleviate the need for application developers to rewrite efficient, scalable, and portable time-driven dispatchers in an ad hoc manner, the ACE Reactor framework defines a family of reusable timer queue classes.

Class Capabilities

The ACE timer queue classes allow applications to register time-driven event handlers derived from

3.5 The ACE_Reactor Class

Motivation

Event-driven networked applications have historically been programmed using native OS mechanisms, such as the Socket API and the select() synchronous event demultiplexer. Applications developed this way, however, are not only nonportable, they are inflexible because they tightly couple low-level event detection, demultiplexing, and dispatching code together with application event processing code. Developers must therefore rewrite all this code for each new networked application, which is tedious, expensive, and error prone. It's also unnecessary because much of event detection, demultiplexing, and dispatching can be generalized and reused across many networked applications.

One way to address these problems is to combine skilled object-oriented design with networked application domain experience to produce a set of framework classes that separates application event handling code from the reusable event detection, demultiplexing, and dispatching code in the framework. Sections 3.2 through 3.4 laid the groundwork for this framework by describing reusable time value and timer queue classes, and by defining the interface between framework and application event processing code with the ACE_Event_Handler class. This section describes how the ACE_Reactor class at the heart of the ACE Reactor framework defines how applications can register for, and be notified about, events from multiple sources.

Class Capabilities

ACE_Reactor implements the Facade pattern [GoF] to define an interface that applications can use to access the various ACE Reactor framework features. This class provides the following capabilities:

-
- It centralizes event loop processing in a reactive application.
-
-
- It detects events via an event demultiplexer, such as select() or WaitForMultipleObjects(), provided by the OS and used by the reactor implementation.
-
-
- It demultiplexes events to event handlers when the event demultiplexer indicates the occurrence of the designated events.
-
-
- It dispatches the appropriate hook methods on registered event handlers to perform application-defined processing in response to the events.
-
-
- It ensures that any thread can change a Reactor's event set or queue a callback to an event handler and expect the Reactor to act on the request promptly.
-

3.6 Summary

This chapter showed how the ACE Reactor framework can simplify the development of concise, correct, portable, and efficient event-driven networked applications by encapsulating OS event demultiplexing mechanisms within an object-oriented C++ interface. Likewise, it showed how the ACE Reactor framework enhances reuse, improves portability, and enables the extensibility of event handlers by separating event detection, demultiplexing, and dispatching mechanisms from application-defined event processing policies.

Since reusable classes in the ACE Reactor framework perform the lower-level event detection, demultiplexing, and event handler dispatching, a relatively small amount of application-defined code must be written. For example, the logging service in Sections 3.3 and 3.4 is mostly concerned with application-defined processing activities, such as receiving client log records. Any applications that reuse the ACE_Reactor class described in Section 3.5 can therefore leverage the knowledge and experience of its skilled middleware developers, as well as its future enhancements and optimizations.

The ACE Reactor framework uses dynamic binding extensively since the dramatic improvements in clarity, extensibility, and modularity it provides usually compensate for any decrease in efficiency resulting from its indirect virtual table dispatching [HLS97]. The ACE Reactor framework is often used to develop networked applications, where the major sources of overhead result from caching, latency, network/host interface hardware, presentation-level formatting, dynamic memory allocation and copying, synchronization, and concurrency management. The additional indirection caused by dynamic binding is often insignificant by comparison [Koe92]. In addition, good C++ compilers can eliminate virtual method overhead completely via the use of "adjustor thunk" optimizations [Lip96].

One of the most powerful properties of the ACE Reactor framework design is its ability to enhance extensibility both above and below its public interface. The Reactor implementations provide a good example of how applying patterns can provide a set of classes that take advantage of unique platform capabilities, while maintaining the ability for networked applications to run unchanged across diverse computing platforms. The next chapter studies the techniques that ACE uses to achieve this flexibility.

Chapter 4. ACE Reactor Implementations

CHAPTER SYNOPSIS

This chapter describes the design and use of several implementations of the ACE_Reactor interface described in Chapter 3. These implementations support a diverse collection of operating system synchronous event demultiplexing mechanisms, including select(), WaitForMultipleObjects(), XtAppMainLoop(), and /dev/poll. We explain the motivations for, and capabilities provided by, the most common reactor implementations available in the ACE Reactor framework. We also illustrate how to use three different implementations of the ACE_Reactor to improve our logging server example. In addition, we show the range of concurrency models supported by these ACE_Reactor implementations.

4.1 Overview

The ACE Reactor framework discussed in Chapter 3 presents a real-world example of a framework that's designed for extensibility. The original ACE_Reactor implementation was based solely on the select() synchronous event demultiplexing mechanism. As application requirements and ACE platform support evolved, however, the internal design of the ACE Reactor framework changed to support new application needs and new OS platform capabilities. Fortunately, the ACE_Reactor interface has remained relatively consistent. This stability is important since it simultaneously helps to

-
- Ensure compatibility with applications written for previous ACE versions
-
-
- Allow each application to take advantage of new reactor capabilities as the need arises
-

This chapter focuses on the most common implementations of the ACE Reactor framework, which are listed in the following table:

ACE Class	**Description**
ACE_Select_Reactor	Uses the select() synchronous event demultiplexer function to detect I/O and timer events; incorporates orderly handling of POSIX signals.
ACE_TP_Reactor	Uses the Leader/Followers pattern [POSA2] to extend ACE_Select_Reactor event handling to a pool of threads.
ACE_WFMO_Reactor	Uses the Windows WaitForMultipleObjects() event demultiplexer function to detect socket I/O, timeouts, and Windows synchronization events.

ACE also offers other more specialized reactor implementations that are outlined in Section 4.5. The variety of reactor requirements motivating all these reactors grew out of the popular reactive model of networked application design, coupled with:

-
- The growing popularity and availability of multithreaded systems
-
-
- The addition of Windows to ACE's set of supported platforms

4.2 The ACE_Select_Reactor Class

Motivation

As discussed in Chapter 5 of C++NPv1, a reactive server responds to events from one or more sources. Ideally, response to events is quick enough so that all requests appear to be processed simultaneously, although event processing is usually handled by a single thread. A synchronous event demultiplexer is at the heart of each reactive server. This demultiplexer mechanism detects and reacts to events originating from a number of sources, making the events available to the server synchronously, as part of its normal execution path.

The select() function is the most common synchronous event demultiplexer. This system function waits for specified events to occur on a set of I/O handles. [1] When one or more of the I/O handles become active, or after a designated amount of time elapses, select() returns. Its return value indicates the number of handles that are active, that the caller-specified time elapsed before an event occurred, or an error occurred. The caller can then take appropriate action. Additional coverage of select() is available in Chapter 6 of C++NPv1 and in [Ste98].

[1] The Windows version of select() works only on socket handles.

Although select() is available on most OS platforms, programming to the native select() C API requires developers to wrestle with many low-level details, such as

-
- Setting and clearing fd_sets
-
-
- Detecting events and responding to signal interrupts
-
-
- Managing internal locks
-
-
- Demultiplexing events to associated event handlers
-
-
- Dispatching functions that process I/O, signal, and timer events
-

Chapter 7 of C++NPv1 discussed several wrapper facade classes that can be used to master many complexities associated with these low-level details. It's also useful, however, to use select() in environments where it's necessary to

-

4.3 The ACE_TP_Reactor Class

Motivation

Although the ACE_Select_Reactor is flexible, it's somewhat limited in multithreaded applications because only the owner thread can call its handle_events() method. ACE_Select_Reactor therefore serializes processing at the event demultiplexing layer, which may be overly restrictive and nonscalable for certain networked applications. One way to solve this problem is to spawn multiple threads and run the event loop of a separate instance of ACE_Select_Reactor in each of them. This design can be hard to program, however, since it requires developers to implement a proxy that partitions event handlers evenly between the reactors to divide the load evenly across threads. Often, a more effective way to address the limitations with ACE_Select_Reactor is to use the ACE Reactor framework's ACE_TP_Reactor class, where "TP" stands for "thread pool."

Class Capabilities

ACE_TP_Reactor is another implementation of the ACE_Reactor interface. This class implements the Leader/Followers architectural pattern [POSA2], which provides an efficient concurrency model where multiple threads take turns calling select() on sets of I/O handles to detect, demultiplex, dispatch, and process service requests that occur. In addition to supporting all the features of the ACE_Reactor interface, the ACE_TP_Reactor provides the following capabilities:

-
- It enables a pool of threads to call its handle_events() method, which can improve scalability by handling events on multiple handles concurrently. As a result, the ACE_TP_Reactor::owner() method is a no-op.
-
-
- It prevents multiple I/O events from being dispatched to the same event handler simultaneously in different threads. This constraint preserves the I/O dispatching behavior of ACE_Select_Reactor, alleviating the need to add synchronization locks to a handler's I/O processing.
-
-
- After a thread obtains a set of active handles from select(), the other reactor threads dispatch from that handle set instead of calling select() again.
-

Implementation overview. ACE_TP_Reactor is a descendant of ACE_Reactor_Impl, as shown in Figure 4.1 (page 89). It also serves as a concrete implementation of the ACE_Reactor interface, just like ACE_Select_Reactor. In fact, ACE_TP_Reactor derives from ACE_Select_Reactor and reuses much of its internal design.

Concurrency considerations. Multiple threads running an ACE_TP_Reactor event loop can process events concurrently on different handles. They can also dispatch timeout and I/O callback methods concurrently on the same event handler. The only serialization in the ACE_TP_Reactor occurs when I/O events occur concurrently on the same handle. In contrast, the ACE_Select_Reactor serializes all its dispatching to handlers whose handles are active in the handle set.

4.4 The ACE_WFMO_Reactor Class

Motivation

Although the select() function is available on most operating systems, it's not always the most efficient or most powerful event demultiplexer on any given OS platform. In particular, select() has the following limitations:

-
- On UNIX platforms, it only supports demultiplexing of I/O handles, such as files, terminal devices, FIFOs, and pipes. It does not portably support demultiplexing of synchronizers, threads, or System V Message Queues.
-
-
- On Windows, select() only supports demultiplexing of socket handles.
-
-
- It can only be called by one thread at a time for a particular set of I/O handles, which can degrade potential parallelism.
-

Windows defines the WaitForMultipleObjects() system function, described in Sidebar 24 (page 104), to alleviate these problems. This function works with many Windows handle types that can be signaled. Although it doesn't work directly with I/O handles, it can be used to demultiplex I/O-related events in two ways:

1.
1. Event handles used in overlapped I/O operations
1.
2.
2. Event handles associated with socket handles via WSAEventSelect()
2.

Moreover, multiple threads can call WaitForMultipleObjects() concurrently on the same set of handles, thereby enhancing potential parallelism.

WaitForMultipleObjects() is tricky to use correctly, however, for the following reasons [SS95a]:

-
- WaitForMultipleObjects() returns an index to the first handle array slot with a signaled handle. It does not, however, indicate the number of handles that are signaled, and there is no simple way to scan the handles and check which are. WaitForMultipleObjects() must therefore be invoked numerous times to find all signaled handles. In contrast, select() returns a set of active I/O handles and a count of how many are active.
-
-

4.5 Summary

This chapter described how the most common ACE_Reactor implementations are designed and illustrated some subtle nuances of their different capabilities. In addition to giving some guidelines on when to use each, our discussions emphasized two points:

1.

1. Different implementations of OS event demultiplexing mechanisms can present significant challenges, as well as important opportunities.
1.
2.

2. A well-designed framework can be extended to use OS capabilities effectively while isolating complex design issues in the framework, rather than in application code.
2.

An intelligently designed framework can significantly improve the portability and extensibility of both applications and the framework itself. The ACE Reactor framework implementations are good examples of how applying patterns, such as Wrapper Facade, Facade, and Bridge, and C++ features, such as inheritance and dynamic binding, can yield a high-quality, highly reusable framework with these elusive qualities. The ACE Reactor framework implementations described in this chapter encapsulate many complex capabilities, allowing networked application developers to focus on application-specific concerns.

One of the most powerful properties of the ACE Reactor framework design is its ability to enhance extensibility at the following variation points:

- Customized event handlers. It's straightforward to extend application functionality by inheriting from the ACE_Event_Handler class or one of its pre-defined ACE subclasses (such as ACE_Service_Object, ACE_Task,orACE_Svc_Handler) and selectively implementing the necessary virtual method(s). For example, Chapter 7 illustrates how event handlers in our client and server logging daemons can be customized transparently to support authentication.
- Customized ACE_Reactor implementations. It's straightforward to modify the underlying event demultiplexing mechanism of an ACE_Reactor without affecting existing application code. For example, porting a reactive logging server from a UNIX platform to a Windows platform requires no visible changes to application code. In contrast, porting a C implementation of the server from select() to WaitForMultipleObjects() is tedious and error-prone.

Over the previous decade, ACE's use in new environments has yielded new requirements for event-driven application support. For example, GUI integration is an important area due to new GUI toolkits and event loop requirements. The following new Reactor implementations were made easier due to the ACE Reactor framework's modular design:

Chapter 5. The ACE Service Configurator Framework

CHAPTER SYNOPSIS

This chapter describes the design and use of the ACE Service Configurator framework, which is an implementation of the Component Configurator pattern [POSA2]. This pattern helps increase application extensibility and flexibility by decoupling the behavior of services from the point of time when implementations of these services are configured into application processes. The chapter concludes by illustrating how the ACE Service Configurator framework can help to improve the extensibility of our networked logging server.

5.1 Overview

Section 2.2 described the naming and linking design dimensions that developers need to consider when configuring networked applications. An extensible strategy for addressing these design dimensions is to apply the Component Configurator design pattern [POSA2]. This pattern allows an application to reconfigure its services at run time without having to modify, recompile, or relink the program itself, or shut down and restart the application.

The ACE Service Configurator framework is a portable implementation of the Component Configurator pattern that allows applications to defer configuration and implementation decisions about their services until late in the design cycle?s late as installation time or even at run time. The ACE Service Configurator framework supports the ability to activate services selectively at run time regardless of whether they are

-
- Static services, which are linked statically into an application program
-
-
- Dynamic services, which are linked from one or more shared libraries (DLLs)
-

Each service can also be passed argc/argv-style arguments to set certain information at run time. Due to ACE's integrated framework design, services using the ACE Service Configurator framework can also be dispatched by the ACE Reactor framework.

This chapter examines the following ACE Service Configurator framework classes:

ACE Class	**Description**
ACE_Service_Object	Defines a uniform interface that the ACE Service Configurator framework uses to configure and control a service implementation. Control operations include initializing, suspending, resuming, and terminating a service.
ACE_Service_Repository	A central repository for all services managed using the ACE Service Configurator framework. It provides methods for locating, reporting on, and controlling all of an application's configured services.
ACE_Service_Repository_Iterator	A portable mechanism for iterating through all the services in a repository.

5.2 The ACE_Service_Object Class

Motivation

Service configuration and life cycle management involves the following aspects that we've alluded to briefly above:

-
- Initialization. A service must be initialized, which may involve creating one or more objects or invoking a factory method. Configuration parameters are passed to the service at this time.
-
-
- Execution control. Certain applications require the ability to suspend and resume services. Offering this capability therefore requires a mechanism by which a management application can locate the desired services and then contact the services to request or force the suspend/resume operation.
-
-
- Reporting. Mission-critical services often require the ability to respond to requests for information concerning their status and availability in a uniform way.
-
-
- Termination. Orderly shutdown processes are required to ensure that a service's resources are released properly, any necessary status information is updated, and that service shutdown is ordered properly to avoid improper service interactions.
-

Designing and implementing these capabilities in an ad hoc manner often produces tightly coupled data structures and classes, which are hard to evolve and reuse in future projects.

Moreover, if multiple projects or development groups undertake similar efforts, the primary benefits of service configuration will be lost because it's highly unlikely that multiple designs will interoperate at either the service or management level.

Since service configuration and management are largely application-independent they are good candidates to incorporate into a framework. Enforcing a uniform interface across all networked services makes it easier to configure and manage them consistently. In turn, this consistency simplifies application development and deployment by mitigating key challenges inherent in creating reusable administrative configuration tools. To provide a uniform interface between the ACE Service Configurator framework and the application-defined services, each service must be a descendant of a common base class called ACE_Service_Object.

Class Capabilities

ACE_Service_Object provides a uniform interface that allows service implementations to be configured and managed by the ACE Service Configurator framework. This class provides the following capabilities:

5.3 The ACE_Service_Repository Classes

Motivation

The ACE Service Configurator framework supports the configuration of both single-service and multiservice servers. Section 5.2 explained why the goals of initialization, execution control, reporting, and termination require application services to be based on a common framework class. For the framework to leverage the accessibility provided by ACE_Service_Object effectively, it must store service information in a well-known repository and be able to access and control these service objects individually or collectively.

Application services in multiservice servers also may require access to each other. To avoid tightly coupling these services, and to preserve the benefits of delayed configuration decisions, services should be able to locate each other at run time. Therefore, to satisfy the needs of the framework and applications without requiring developers to provide these capabilities in an ad hoc way, the ACE Service Configurator framework provides the ACE_Service_Repository and ACE_Service_Repository_Iterator classes.

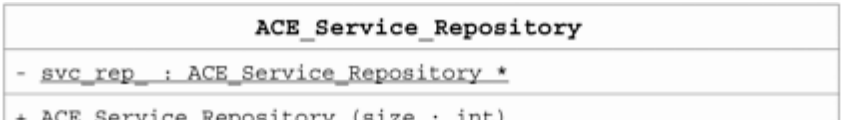
Class Capabilities

ACE_Service_Repository implements the Manager pattern [Som98] to control the life cycle of, and the access to, service objects configured by the ACE Service Configurator framework. This class provides the following capabilities:

-
- It keeps track of all service implementations that are configured into an application and maintains each service's status, such as whether it's active or suspended.
-
-
- It provides the mechanism by which the ACE Service Configurator framework inserts, manages, and removes services.
-
-
- It provides a convenient mechanism to terminate all services, in reverse order of their initialization.
-
-
- It allows an individual service to be located by its name.
-

The interface for ACE_Service_Repository is shown in Figure 5.4 (page 128) and its key methods are outlined in the following table:

Figure 5.4. The ACE_Service_Repository Class



5.4 The ACE_Service_Config Class

Motivation

Before a service can execute, it must be configured into an application's address space. One way to configure services into a networked application is to statically link the functionality provided by its various classes and functions into separate OS processes, and then manually instantiate or initialize them at run time. We used this approach in the logging server examples in Chapters 3 and 4 and throughout C++NPv1, where the logging server program runs in a process that handles log records from client applications. Although our use of the ACE Reactor framework in earlier chapters improved the networked logging server's modularity and portability, the following drawbacks arose from statically configuring the Reactor_Logging_Server class with its main() program:

-
- Service configuration decisions are made prematurely in the development cycle, which is undesirable if developers don't know the best way to collocate or distribute services in advance. Moreover, the "best" configuration may change as the computing context changes. For example, an application may write log records to a local file when it's running on a disconnected laptop computer. When the laptop is connected to a LAN, however, it may forward log records to a centralized logging server. Forcing networked applications to commit prematurely to a particular service configuration impedes their flexibility and can reduce their performance and functionality. It can also force costly redesign and reimplementation later in a project's life cycle.
-
-

- Modifying a service may affect other services adversely if the implementation of a service is coupled tightly with its initial configuration. To enhance reuse, for example, a logging server may initially reside in the same program as other services, such as a name service. If the other services change, however, for example if the name service lookup algorithm changes, all existing code in the server would require modification, recompilation, and static relinking. Moreover, terminating a running process to change some of its service code would also terminate the collocated logging service. This disruption in service may not be acceptable for highly available systems, such as telecommunication switches or customer care call centers [SS94].
-
-

- System performance may scale poorly since associating a separate process with each service ties up OS resources, such as I/O handles, virtual memory, and process table slots. This design is particularly wasteful if services are often idle. Moreover, processes can be inefficient for many short-lived communication tasks, such as asking a time service for the current time or resolving a host address request via the Domain Name Service (DNS).
-

To address the drawbacks of purely static configurations, the ACE Service Configurator framework defines the ACE_Service_Config class.

Class Capabilities

ACE_Service_Config implements the Facade pattern [GoF] to integrate other classes in the ACE Service Configurator framework and coordinate the activities necessary to manage the services in an application. This class provides the following capabilities:

5.5 Summary

Traditional software development techniques that statically link and configure a networked application's services together can be limiting. For example, the effort and time needed to rebuild an entire program for each new or modified service increases development and maintenance costs substantially. Moreover, making these changes in the field is inefficient and error prone, and can cause support costs to soar and customer satisfaction to plummet.

This chapter described the ACE Service Configurator framework, which implements the Component Configurator pattern [POSA2] to provide a portable way to statically and/or dynamically link services and then initiate, suspend, resume, and shut them down dynamically at run time. This framework helps to improve the extensibility of networked software by allowing applications to defer the selection of a particular service implementation until late in the software life cycle?s late as installation time or even run time. This flexibility yields the following important advantages:

-
- Applications can be composed and (re)configured at run time using mix-and-match, independently developed services.
-
-
- Developers can concentrate on a service's functionality and other key design dimensions without committing prematurely to a particular service configuration.
-
-
- Applications are composed of multiple services that are developed independently, so they don't require advanced global knowledge of each other, yet can still collaborate.
-

This chapter explained the origin and usage of each of the ACE Service Configurator framework's classes and helper macros. We also used these capabilities to separate parts of the previous chapter's logging servers into independently linkable and configurable services. The result was a networked logging service that can be configured and deployed in various ways. The extensibility afforded by the ACE Service Configurator framework allows operators and administrators to select the features and alternative implementation strategies that make the most sense in a particular context, as well as make localized decisions on how best to initialize and evolve them.

Chapter 6. The ACE Task Framework

CHAPTER SYNOPSIS

This chapter describes the design and use of the ACE Task framework. This framework helps to enhance the modularity and extensibility of concurrent object-oriented networked applications. The ACE Task framework forms the basis of common concurrency patterns, such as Active Object and Half-Sync/Half-Async [POSA2]. After discussing the motivation and usage of the framework's classes, we apply the ACE Task framework to enhance the concurrency and scalability of our networked logging service.

6.1 Overview

The ACE Task framework provides powerful and extensible object-oriented concurrency capabilities that can spawn threads in the context of an object, as well as transfer and queue messages between objects executing in separate threads. This framework can be applied to implement key concurrency patterns [POSA2], such as:

-
- The Active Object pattern, which decouples the thread that invokes a method from the thread that executes the method. This pattern enhances concurrency and simplifies synchronized access to objects executing in the context of one or more threads.
-
-
- The Half-Sync/Half-Async pattern, which decouples asynchronous and synchronous processing in concurrent systems to simplify programming without unduly reducing performance. This pattern introduces three layers: one for asynchronous (or reactive) processing, one for synchronous service processing, and a queueing layer that mediates communication between the asynchronous/reactive and synchronous layers.
-

This chapter shows how these patterns, and the ACE Task framework that reifies them, can be applied to develop concurrent object-oriented applications at a higher level of abstraction than existing C operating system APIs and C++ wrapper facades. The ACE Task framework consists of the following classes that networked applications can use to spawn and manage threads and pass messages between one or more threads within a process:

ACE Class	**Description**
ACE_Message_Block	Implements the Composite pattern [GoF] to enable efficient manipulation of fixed- and variable-sized messages
ACE_Message_Queue	Provides an intraprocess message queue that enables applications to pass and buffer messages between threads in a process
ACE_Thread_Manager	Allows applications to portably create and manage the lifetime, synchronization, and properties of one or more threads
ACE_Task	Allows applications to create passive or active objects that decouple different units of processing; use messages to communicate requests, responses, data, and control information; and can queue and process messages sequentially or concurrently

6.2 The ACE_Message_Queue Class

Motivation

As discussed in Section 2.1.4 on page 27, networked applications whose services are layered/modular are often composed of a set of collaborating tasks within a process. To simplify interfaces and design, minimize maintenance costs, and maximize reuse, these tasks should have the following properties:

-
- Low intertask coupling, that is, separate task objects should have minimal dependencies on each other's data and methods.
-
-
- High intratask cohesion, that is, the methods and data in a task should focus on a related set of functionality.
-

To achieve these properties, tasks often communicate by passing messages via a generic method, such as push() or put(), rather than calling specific statically typed methods directly. Messages can represent work requests, work results, or other types of data to process. They can also represent control requests that direct tasks to alter their processing, for example, to shut down or reconfigure themselves.

When producer and consumer tasks are collocated in the same process, tasks often exchange messages via an intraprocess message queue. In this design, producer task(s) insert messages into a synchronized message queue serviced by consumer task(s) that remove and process the messages. If the queue is full, producers can either block or wait a bounded amount of time to insert their messages. Likewise, if the queue is empty, consumers can either block or wait a bounded amount of time to remove messages.

Although some operating systems supply intraprocess message queues natively, this capability isn't available on all platforms. Moreover, when it is offered, it's often either highly platform specific, such as VxWorks message queues, and/or inefficient, tedious, and error prone to use, such as System V IPC message queues [Ste99]. Examples of how to create wrapper facade classes to handle these problems appear in C++NPv1. Wrapper facades could encapsulate the spectrum of available intraprocess message queue mechanisms behind a common interface, emulating missing capabilities where needed. ACE takes a different approach, however, for the following reasons:

-
- To avoid unnecessary complexity. Native message queueing mechanisms, where they exist, can be hard to program correctly since they use low-level C APIs. They can also impose constraints on system administration that reflect poorly on a product's operational procedures, which can increase product support costs. For example, System V IPC message queues can persist after a program finishes execution if not cleaned up properly. These remnants may prevent an application from restarting, or contribute to resource leaks and often require a system administrator's intervention to repair the system manually. Likewise, System V IPC message queues offer interprocess queueing that incurs more overhead than the intraprocess queueing that's the target use case for many networked applications.
-
-

6.3 The ACE_Task_Class

Motivation

The ACE_Message_Queue class described in Section 6.2 can be used to

-
- Decouple the flow of information from its processing
-
-
- Link threads that execute producer/consumer services concurrently
-

To use a producer/consumer concurrency model effectively in an object-oriented program, however, each thread should be associated with the message queue and any other service-related information. To preserve modularity and cohesion, and to reduce coupling, it's therefore best to encapsulate an ACE_Message_Queue with its associated data and methods into one class whose service threads can access it directly.

Thread-spawning capabilities provided by popular OS platforms are based on each spawned thread invoking a C-style function call. The ACE_Thread_Manager wrapper facade class described in Chapter 9 of C++NPv1 implements portable multithreading capabilities. However, programmers must still pass a C-style function to its spawn() and spawn_n() methods. Providing a spawned thread with access to a C++ object requires a bridge to the C++ object environment. The CLD_Handler::open() method (page 172) illustrated this technique. Since implementing this technique manually for each class is repetitive, it's a good candidate for reuse. The ACE Task framework therefore defines ACE_Task to encapsulate a class's messaging capability and provide a portable way for thread(s) to execute in the context of an object.

Class Capabilities

ACE_Task is the basis of ACE's object-oriented concurrency framework. It provides the following capabilities:

-
- It uses an instance of ACE_Message_Queue from Section 6.2 to separate data and requests from their processing.
-
-
- It uses the ACE_Thread_Manager class to activate the task so it runs as an active object [POSA2] that processes its queued messages in one or more threads of control. Since each thread runs a designated class method, they can access all of the task's data members directly.
-
-
- It inherits from ACE_Service_Object, so its instances can be configured dynamically via the ACE Service Configurator framework from Chapter 5.
-

6.4 Summary

The ACE Task framework allows developers to create and configure concurrent networked applications using powerful and extensible object-oriented designs. This framework provides the ACE_Task class that integrates multithreading with object-oriented programming and queueing. The queueing mechanism in ACE_Task uses the ACE_Message_Queue class to transfer messages between tasks efficiently. Since ACE_Task derives from the ACE_Service_Object in Section 5.2, it's easy to design services that can be configured dynamically to run as active objects and be dispatched by the ACE Reactor framework.

This chapter illustrated how the ACE Reactor framework can be combined with the ACE Task framework to implement variants of the Half-Sync/Half-Async pattern [POSA2]. The ACE Task framework classes can also be combined with the ACE_Future, ACE_Method_Request, and ACE_Activation_List classes to implement the Active Object pattern [POSA2]. A subset of the ACE_Message_Queue implementation is presented in Chapter 10 of C++NPv1.

Chapter 7. The ACE Acceptor-Connector Framework

CHAPTER SYNOPSIS

This chapter describes the design and use of the ACE Acceptor-Connector framework. This framework implements the Acceptor-Connector pattern [POSA2], which decouples the connection and initialization of cooperating peer services in a networked application from the processing they perform after being connected and initialized. The Acceptor-Connector framework allows applications to configure key properties of their connection topologies independently from the services they provide. We illustrate how this framework can be combined with the ACE Reactor and Task frameworks and applied to enhance the reusability, extensibility, security, and scalability of our networked logging service.

7.1 Overview

Many networked applications, such as e-mail, remote file backups, and Web services, use connection-oriented services containing classes that play the following roles:

-
- The connection role determines how an application establishes connections.
-
-
- The communication role determines whether an application plays the role of a client, a server, or both client and server in a peer-to-peer configuration.
-

Networked applications that communicate via connection-oriented protocols (e.g., TCP/IP) are typified by the following asymmetric connection roles between clients and servers:

-
- Servers often wait passively to accept connections by listening on a designated TCP port.
-
-
- Clients often initiate connections actively by connecting to a server's listening port.
-

Even in peer-to-peer applications, where applications play both client and server roles, connections must be initiated actively by one peer and accepted passively by the other. To enhance reuse and extensibility, networked applications should be designed to easily change connection and communication roles to support different requirements and environments.

The ACE Acceptor-Connector framework implements the Acceptor-Connector design pattern [POSA2], which enhances software reuse and extensibility by decoupling the activities required to connect and initialize cooperating peer services in a networked application from the processing they perform once they're connected and initialized. This chapter describes the following ACE Acceptor-Connector framework classes that networked applications can use to establish connections and initialize peer services:

ACE Class	**Description**
ACE_Svc_Handler	Represents the local end of a connected service and contains an IPC endpoint used to communicate with a connected peer.
ACE_Acceptor	This factory waits passively to accept a connection and then initializes an ACE_Svc_Handler in response to an

7.2 The ACE_Svc_Handler Class

Motivation

Chapter 2 defined a service as a set of functionality offered to a client by a server. A service handler is the portion of a networked application that either implements or accesses (or both, in the case of a peer-to-peer arrangement) a service. Connection-oriented networked applications require at least two communicating service handlers?ne for each end of every connection. Incidentally, applications using multicast or broadcast communication may have multiple service handlers. Although these connectionless communication protocols don't cleanly fit the Acceptor-Connector model, the ACE_Svc_Handler class is often a good choice for implementing a service handler and should be considered.

When designing the service handlers involved in a service, developers should also take into account the communication design dimensions discussed in Chapter 1 of C++NPv1. In general, the application functionality defined by a service handler can be decoupled from the following design aspects:

-
- How the service handler was connected (actively or passively) and initialized
-
-
- The protocols used to connect, authenticate, and exchange messages between two service handlers
-
-
- The network programming API used to access the OS IPC mechanisms
-

In general, connection/authentication protocols and service initialization strategies change less frequently than the service handler functionality implemented by an application. To separate these concerns and allow developers to focus on the functionality of their service handlers, the ACE Acceptor-Connector framework defines the ACE_Svc_Handler class.

Class Capabilities

ACE_Svc_Handler is the basis of ACE's synchronous and reactive data transfer and service processing mechanisms. This class provides the following capabilities:

-
- It provides the basis for initializing and implementing a service in a synchronous and/or reactive networked application, acting as the target of the ACE_Connector and ACE_Acceptor connection factories.
-
-
- It provides an IPC endpoint used by a service handler to communicate with its peer service handler(s). The type of this IPC endpoint can be parameterized with many of ACE's IPC wrapper facade classes, thereby separating lower-level communication mechanisms from application-level service processing policies.

7.3 The ACE_Acceptor Class

Motivation

Many connection-oriented server applications tightly couple their connection establishment and service initialization code in ways that make it hard to reuse existing code. For example, if you examine the Logging_Acceptor (page 58), Logging_Acceptor_Ex (page 67), Logging_Acceptor_WFMO (page 112), CLD_Acceptor (page 176), and TP_Logging_Acceptor (page 193) classes, you'll see that the handle_input() method was rewritten for each logging handler, even though the structure and behavior of the code was nearly identical. The ACE Acceptor-Connector framework defines the ACE_Acceptor class so that application developers needn't rewrite this code repeatedly.

Class Capabilities

ACE_Acceptor is a factory that implements the Acceptor role in the Acceptor-Connector pattern [POSA2]. This class provides the following capabilities:

-
- It decouples the passive connection establishment and service initialization logic from the processing performed by a service handler after it's connected and initialized.
-
-
- It provides a passive-mode IPC endpoint used to listen for and accept connections from peers. The type of this IPC endpoint can be parameterized with many of ACE's IPC wrapper facade classes, thereby separating lower-level connection mechanisms from application-level service initialization policies.
-
-
- It automates the steps necessary to connect the IPC endpoint passively and create/activate its associated service handler.
-
-
- Since ACE_Acceptor is derived from ACE_Service_Object, it inherits the event-handling and configuration capabilities described in Chapters 3 and 5.
-

The interface for ACE_Acceptor is shown in Figure 7.4. As shown in the figure, this class template is parameterized by:

Figure 7.4. The ACE_Acceptor Class

7.4 The ACE_Connector Class

Motivation

Section 7.3 focused on how to decouple the functionality of service handlers from the steps required to passively connect and initialize them. It's equally useful to decouple the functionality of service handlers from the steps required to actively connect and initialize them. Moreover, networked applications that communicate with a large number of peers may need to actively establish many connections concurrently, handling completions as they occur. To consolidate these capabilities into a flexible, extensible, and reusable abstraction, the ACE Acceptor-Connector framework defines the ACE_Connector class.

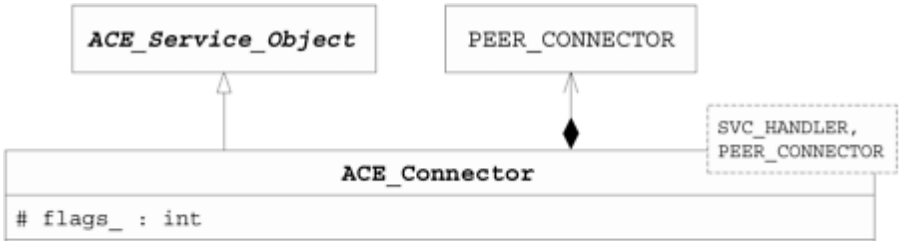
Class Capabilities

ACE_Connector is a factory class that implements the Connector role in the Acceptor-Connector pattern [POSA2]. This class provides the following capabilities:

-
- It decouples the active connection establishment and service initialization logic from the processing performed by a service handler after it's connected and initialized.
-
-
- It provides an IPC factory that can actively establish connections with a peer acceptor either synchronously or reactively. The type of this IPC endpoint can be parameterized with many of ACE's IPC wrapper facade classes, thereby separating lower-level connection mechanisms from application-level service initialization policies.
-
-
- It automates the steps necessary to connect the IPC endpoint actively as well as to create and activate its associated service handler.
-
-
- Since ACE_Connector derives from ACE_Service_Object it inherits all the event handling and dynamic configuration capabilities described in Chapters 3 and 5, respectively.
-

The interface for ACE_Connector is shown in Figure 7.7 (page 230). This template class is parameterized by:

Figure 7.7. The ACE_Connector Class



7.5 Summary

The ACE Acceptor-Connector framework decouples a service's connection and initialization strategies from its service handling strategy. This separation of concerns allows each set of strategies to evolve independently and promotes a modular design of networked applications. The ACE Acceptor-Connector framework factors connection and initialization strategies into the ACE_Acceptor and ACE_Connector class templates, and the service handling strategy into the ACE_Svc_Handler class template.

The ACE Reactor, Service Configurator, and Task frameworks described in earlier chapters use class inheritance and virtual methods as their primary extensibility mechanisms. The ACE Acceptor-Connector framework uses these mechanisms as well, primarily as the means to configure different strategies for connection establishment, communication, concurrency, and service behavior. Unlike the frameworks in previous chapters, however, classes in the ACE Acceptor-Connector framework share an inherent relationship in networked application services, so the use of parameterized types plays a more significant role here. To allow and enforce the relationships between the strategies, both ACE_Acceptor and ACE_Connector include an ACE_Svc_Handler-derived class in their template arguments to act as the target of the connection factory.

This chapter defined and illustrated the communication and connection roles that networked application services play, as well as the passive and active connection modes that connection-oriented services use. Although the ACE Socket wrapper facades described in Chapter 3 of C++NPv1 assist with mastering the problems associated with C operating system APIs, this chapter illustrated how the design of the ACE Acceptor-Connector framework encourages modular separation of roles leading to highly extensible and maintainable designs. The examples showed how easy it is to define an application's service handlers by defining class(es) derived from ACE_Svc_Handler and adding the service-specific behavior in the hook and callback methods inherited from ACE_Svc_Handler, ACE_Task, and ACE_Event_Handler. Although the ACE Acceptor-Connector framework encapsulates the most common use-cases for service establishment, this chapter showed how the framework uses the Template Method pattern [GoF] to allow application developers to customize the behavior of each service establishment step to match the requirements, environment, and resources of specific networked applications.

Chapter 8. The ACE Proactor Framework

CHAPTER SYNOPSIS

This chapter outlines the asynchronous I/O mechanisms available on today's popular OS platforms and then describes the design and use of the ACE Proactor framework. This framework implements the Proactor pattern [POSA2], which allows event-driven applications to efficiently demultiplex and dispatch service requests triggered by the completion of asynchronous I/O operations. This chapter shows how to enhance our client logging daemon to use a proactive model that (1) initiates I/O operations, (2) demultiplexes I/O completion events, and (3) dispatches those completion events to application-defined completion handlers that process the results of asynchronous I/O operations.

8.1 Overview

Chapter 3 described the ACE Reactor framework, which is most often used with a reactive I/O model. An application based on this model registers event handler objects that are notified by a reactor when it's possible to perform one or more desired I/O operations, such as receiving data on a socket, with a high likelihood of immediate completion. I/O operations are often performed in a single thread, driven by the reactor's event dispatching loop. Although reactive I/O is a common programming model, each thread can execute only one I/O operation at a time. The sequential nature of the I/O operations can be a bottleneck since applications that transfer large amounts of data on multiple endpoints can't use the parallelism available from the OS and/or multiple CPUs or network interfaces.

One way to alleviate the bottlenecks of reactive I/O is to use synchronous I/O in conjunction with a multithreading model, such as the thread pool model in Chapter 6 or the thread-per-connection model in Chapter 7. Multithreading can help parallelize an application's I/O operations and may improve performance. However, adding multiple threads to a design requires appropriate synchronization mechanisms to avoid concurrency hazards, such as race conditions and deadlocks [Tan92]. These additional considerations require expertise in concurrency and synchronization techniques. They also add complexity to both design and code, increasing the risk of subtle defects. Moreover, multithreading can incur non-trivial time/space overhead due to the resources needed to allocate run-time stacks, perform context switches [SS95b], and move data between CPU caches [SKT96].

A proactive I/O model is often a more scalable way to alleviate reactive I/O bottlenecks without introducing the complexity and overhead of synchronous I/O and multithreading. This model allows an application to execute I/O operations via the following two phases:

1.

1. The application can initiate one or more asynchronous I/O operations on multiple I/O handles in parallel without having to wait until they complete.
1.
2.

2. As each operation completes, the OS notifies an application-defined completion handler that then processes the results from the completed I/O operation.
2.

The two phases of the proactive I/O model are essentially the inverse of those in the reactive I/O model, in which an application

1.

1. Uses an event demultiplexer to determine when an I/O operation is possible, and likely to complete immediately, and then
1.
2.

2. Performs the operation synchronously
2.

In addition to improving application scalability via asynchrony, the proactive I/O model can offer other benefits, depending on the platform's implementation of asynchronous I/O. For example, if multiple asynchronous I/O operations can be initiated simultaneously and each operation carries extended information, such as file positions for

8.2 The Asynchronous I/O Factory Classes

Motivation

The proactive I/O model is generally harder to program than reactive and synchronous I/O models because

-
- I/O initiation and completion are distinct activities that must be handled separately.
-
-
- Multiple I/O operations can be initiated simultaneously, which requires more recordkeeping.
-
-
- There's no guaranteed completion order when multiple I/O operations complete simultaneously.
-
-
- In a multithreaded service a completion handler may execute in a thread other than the one that initiated the I/O operation.
-

The proactive I/O model therefore requires a factory to initiate asynchronous I/O operations. Since multiple I/O operations can execute simultaneously and complete in any order, the proactive model also requires an explicit binding between each asynchronous operation, its parameters (such as the I/O handle, data buffer, and buffer size), and the completion handler that will process the results of the operation.

In theory, designing classes to generate asynchronous I/O operations and bind them to their completion handlers should be relatively straightforward. In practice, however, the design is complicated by the fact that asynchronous I/O is implemented in different ways across today's popular OS platforms. Two common examples include:

-
- Windows. The Windows ReadFile() and WriteFile() system functions can either perform synchronous I/O or initiate an overlapped I/O operation.
-
-
- POSIX. The POSIX aio_read() and aio_write() functions initiate asynchronous read and write operations, respectively. These functions are separate from the read() and write() (and Sockets recv() and send()) functions that are used in ACE's IPC wrapper facade classes (see Chapter 3 in C++NPv1).
-

Each platform's asynchronous I/O facility also includes its own mechanism for binding an I/O operation with its parameters, such as buffer pointer and transfer size. For example, POSIX AIO provides an AIO control block (aiocb), whereas Windows provides the OVERLAPPED structure and a completion key argument to the I/O completion port facility. Sidebar 54 discusses other challenges with OS asynchronous I/O mechanisms.

8.3 The ACE_Handler Class

Motivation

A chief differentiator between the proactive and reactive I/O models is that proactive I/O initiation and completion are distinct steps that occur separately. Moreover, these two steps may occur in different threads of control. Using separate classes for the initiation and completion processing avoids unnecessarily coupling the two. Section 8.2 described the ACE_Asynch_Read_Stream and ACE_Asynch_Write_Stream classes used to initiate asynchronous I/O operations; this section focuses on I/O completion handling.

Completion events signify that a previously initiated I/O operation has finished. To process the result of the I/O operation correctly and efficiently, a completion handler must know all of the arguments specified for the I/O operation, in addition to the result. Together, this information includes

- What type of operation was initiated
- Whether or not the operation completed successfully
- The error code, if the operation failed
- The I/O handle that identifies the communication endpoint
- The memory address for the transfer
- The requested and actual number of bytes transferred

Asynchronous I/O completion processing requires more information than is available to callback methods in the ACE Reactor framework. The ACE_Event_Handler class presented in Section 3.3 is therefore not suitable for use in the ACE Proactor framework. Since completion handling also depends on the asynchronous I/O mechanism offered by the underlying OS platform, it has the same portability issues discussed in Section 8.2. Addressing these issues in each application is unnecessarily tedious and costly, which is why the ACE Proactor framework provides the ACE_Handler class.

Class Capabilities

ACE_Handler is the base class of all asynchronous completion handlers in the ACE Proactor framework. This class

8.4 The Proactive Acceptor-Connector Classes

Motivation

TCP/IP connection establishment is a two-step process:

1.

1. The application either binds a listening socket to a port and listens, or learns of a listening application and initiates an active connection request.
1.
2.

2. The connect operation completes after OS-mediated TCP protocol exchanges open the new connection.
2.

This two-step process is often performed using either a reactive or synchronous I/O model, as shown in Chapter 3 of C++NPv1 and in Chapter 7 of this book. However, the initiate/complete protocol of TCP connection establishment lends itself well to the proactive model. Networked applications that benefit from asynchronous I/O can therefore also benefit from asynchronous connection establishment capabilities.

OS support for asynchronous connection establishment varies. For example, Windows supports asynchronous connection establishment, whereas POSIX.4 AIO does not. It's possible, however, to emulate asynchronous connection establishment where it doesn't exist by using other OS mechanisms, such as multithreading (Sidebar 57 on page 283 discusses the ACE Proactor framework's emulation for POSIX). Since redesigning and rewriting code to encapsulate or emulate asynchronous connection establishment for each project or platform is tedious and error prone, the ACE Proactor framework provides the ACE_Asynch_Acceptor, ACE_Asynch_Connector, and ACE_Service_Handler classes.

Class Capabilities

ACE_Asynch_Acceptor is another implementation of the acceptor role in the Acceptor-Connector pattern [POSA2]. This class provides the following capabilities:

-

- It initiates asynchronous passive connection establishment.
-
-

- It acts as a factory, creating a new service handler for each accepted connection.
-
-

- It can cancel a previously initiated asynchronous accept() operation.
-
-

- It provides a hook method to obtain the peer's address when the new connection is established.

8.5 The ACE_Proactor Class

Motivation

Asynchronous I/O operations are handled in two steps: initiation and completion. Since multiple steps and classes are involved, there must be a way to demultiplex the completion events and efficiently associate each completion event with the operation that completed and the completion handler that will process the result. The diversity of OS asynchronous I/O facilities plays a deeper role here than in the reactive I/O model because

-
- Platforms have different ways to receive completion notifications. For example, Windows uses I/O completion ports or events, whereas POSIX.4 AIO uses real-time signals or the aio_suspend() system function to wait for a completion.
-
-
- Platforms use different data structures to maintain state information for asynchronous I/O operations. For example, Windows uses the OVERLAPPED structure, whereas POSIX.4 AIO uses struct aiocb.
-

Thus, the chain of knowledge concerning platform-specific mechanisms and data structures runs from initiation operations through dispatching and into completion handling. In addition to being complicated and hard to reimplement continually, it's easy to tightly couple proactive I/O designs. To resolve these issues and provide a portable and flexible completion event demultiplexing and dispatching facility, the ACE Proactor framework defines the ACE_Proactor class.

Class Capabilities

ACE_Proactor implements the Facade pattern [GoF] to define an interface that applications can use to access the various ACE Proactor framework features portably and flexibly. This class provides the following capabilities:

-
- It centralizes event loop processing in a proactive application.
-
-
- It dispatches timer expirations to their associated ACE_Handler objects.
-
-
- It demultiplexes completion events to completion handlers and dispatches the appropriate hook methods on completion handlers that then perform application-defined processing in response to the completion events.
-
-
- It can decouple the thread(s) performing completion event detection, demultiplexing, and dispatching from thread(s) initiating asynchronous operations.
-

8.6 Summary

This chapter explored the concept of proactive I/O and outlined how the proactive model differs from the reactive model. It also showed how the proactive I/O model can be used to overcome the performance limitations of the reactive I/O model without incurring certain liabilities associated with the use of multithreaded synchronous I/O. However, the proactive I/O model presents several challenges:

-
- Design challenges. The multistep nature of this model increases the likelihood of overly coupling the I/O mechanisms that initiate asynchronous operations with the processing of the completions of operations.
-
-
- Portability challenges. There are highly divergent standards and implementations for asynchronous I/O offered by today's computing platforms.
-

The Proactor pattern [POSA2] defines a set of roles and relationships to help simplify applications that use proactive I/O. The ACE Proactor framework implements the Proactor pattern across a range of operating systems that support asynchronous I/O. The ACE Proactor framework provides a set of classes that simplify networked application use of asynchronous I/O capabilities across all platforms that offer it. This chapter discussed each class in the framework, covering their motivations and capabilities. It showed an implementation of the client logging daemon that uses the proactive I/O model for all of its network operations. This version of the client logging daemon works portably on all ACE platforms that offer asynchronous I/O mechanisms.

Chapter 9. The ACE Streams Framework

CHAPTER SYNOPSIS

This chapter describes the design and use of the ACE Streams framework. This framework implements the Pipes and Filters pattern [POSA1], which is an architectural pattern that provides a structure for systems that process streams of data. We illustrate how the ACE Streams framework can be used to develop a utility program that formats and prints files of log records stored by our logging servers.

9.1 Overview

The Pipes and Filters architectural pattern is a common way of organizing layered/modular applications [SG96]. This pattern defines an architecture for processing a stream of data in which each processing step is encapsulated in some type of filter component. Data is passed between adjacent filters via a communication mechanism, which can range from IPC channels connecting local or remote processes to simple pointers that reference objects within the same process. Each filter can add, modify, or remove data before passing it along to the next filter. Filters are often stateless, in which case data passing through the filter are transformed and passed along to the next filter without being stored.

Common examples of the Pipes and Filters pattern include

-
- The UNIX pipe IPC mechanism [Ste92] used by UNIX shells to create unidirectional pipelines
-
-
- System V STREAMS [Rit84], which provides a framework for integrating bidirectional protocols into the UNIX kernel
-

The ACE Streams framework is based on the Pipes and Filters pattern. This framework simplifies the development of layered/modular applications that can communicate via bidirectional processing modules. This chapter describes the following classes in the ACE Streams framework:

ACE Class	**Description**
ACE_Task	A cohesive unit of application-defined functionality that uses messages to communicate requests, responses, data, and control information and can queue and process messages sequentially or concurrently.
ACE_Module	A distinct bidirectional processing layer in an application that contains two ACE_Task objects?ne for "reading" and one for "writing"
ACE_Stream	Contains an ordered list of interconnected ACE_Module objects that can be used to configure and execute layered application-defined services

The most important relationships between classes in the ACE Streams framework are shown in Figure 9.1. These classes play the following roles in accordance with the Pipes and Filters pattern [POSA1]:

9.2 The ACE_Module Class

Motivation

Many networked applications can be modeled as an ordered series of processing layers that are related hierarchically and that exchange messages between adjacent layers. For example, kernel-level [Rit84, Rag93] and user-level [SS95b, HJE95] protocol stacks, call center managers [SS94], and other families of networked applications can benefit from a message-passing design based on a layered/modular service architecture. As discussed in Section 2.1.4, each layer can handle a self-contained portion (such as input or output, event analysis, event filtering, or service processing) of a service or networked application.

The ACE_Task class provides a reuseable component that can easily be used to separate processing into stages and pass data between them. Since ACE_Task objects are independent, however, additional structure is required to order ACE_Task objects into bidirectional "reader-writer" pairs that can be assembled and managed as a unit. Redeveloping this structure in multiple projects is tedious and unnecessary because the structure is fundamentally application independent. To avoid this redundant development effort, therefore, the ACE Streams framework defines the ACE_Module class.

Class Capabilities

ACE_Module defines a distinct layer of application-defined functionality. This class provides the following capabilities:

-
- Each ACE_Module is a bidirectional application-defined processing layer containing a pair of reader and writer tasks that derive from ACE_Task. Layered designs can be expressed easily using ACE_Module, which simplifies development, training, and evolution.
-
-
- The ACE Service Configurator framework supports dynamic construction of ACE_Module objects that can be configured into an ACE_Stream at run time. Layered designs based on ACE_Module are therefore highly extensible.
-
-
- The reader and writer ACE_Task objects contained in an ACE_Module collaborate with adjacent ACE_Task objects by passing messages via a public hook method, which promotes loose coupling and simplifies reconfiguration.
-
-
- The objects composed into an ACE_Module can be varied and replaced independently, which lowers maintenance and enhancement costs.
-

The interface for ACE_Module is shown in Figure 9.2 and its key methods are shown in the following table:

9.3 The ACE_Stream Class

Motivation

The ACE_Module class described in Section 9.2 can be used to decompose a networked application into a series of interconnected, functionally distinct layers. Each module implements a different layer of application-defined functionality, such as a reader, formatter, separator, and writer of log records. ACE_Module provides methods to transfer messages between sibling tasks within a module, as well as between modules. It does not, however, provide a facility to connect or rearrange modules in a particular order. To enable developers to build and manage a series of hierarchically related module layers as a single object, the ACE Streams framework defines the ACE_Stream class.

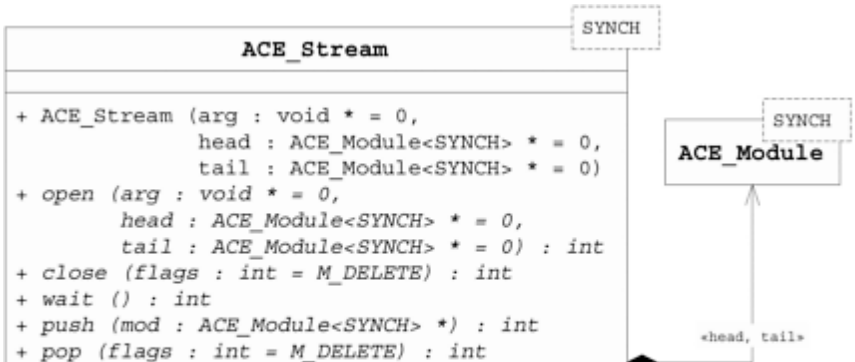
Class Capabilities

ACE_Stream implements the Pipes and Filters pattern [POSA1] to enable developers to configure and execute hierarchically related services by customizing reusable application-independent framework classes. This class provides the following capabilities:

-
- It provides methods to dynamically add, replace, and remove ACE_Module objects to form various stream configurations.
-
-
- It provides methods to send/receive messages to/from an ACE_Stream.
-
-
- It provides a mechanism to connect two ACE_Stream streams together.
-
-
- It provides a way to shut down all modules in a stream and wait for them all to stop.
-

The interface for ACE_Stream is shown in Figure 9.4. Since this class exports many features of the ACE Streams framework, we group its method descriptions into the three categories described below.

Figure 9.4. The ACE_Stream Class



9.4 Summary

The ACE Streams framework is an implementation of the Pipes and Filters pattern that employs object-oriented design techniques, the ACE Task framework, and C++ language features. The ACE Streams framework makes it easy to incorporate new or modified functionality into an ACE_Stream without modifying the application-independent framework classes. For example, incorporating a new layer of service functionality into an ACE_Stream involves the following steps:

1.

1. Inheriting from the ACE_Task interface and overriding the open(), close(), put(), and svc() methods in the ACE_Task subclass to implement application-defined functionality.
1.
2.

2. Allocating a new ACE_Module that contains one or two instances of the application-defined ACE_Tasks, one for the reader-side and one for the writer-side.
2.
3.

3. Inserting the ACE_Module into an ACE_Stream object. Multiple ACE_Modules can be inserted into an ACE_Stream to form an ordered series of hierarchically related processing capabilities.
3.

The ACE Streams framework enables developers to create layered, modular networked applications that are easily extended, tuned, maintained and configured. Moreover, the synergy between the ACE Task, Service Configurator, and Streams frameworks allows a wide range of designs and configurations that can be extended and modified to suit countless design situations, run-time environments, and OS platforms.

Glossary

Acceptor-Connector Pattern

A design pattern that decouples the connection and initialization of cooperating peer services in a networked system from the processing they perform once connected and initialized.

Active Connection Establishment

The connection role played by a peer application that initiates a connection to a remote peer (compare with Passive Connection Establishment).

Active Object

An object that implements the Active Object pattern. Such objects generally execute service requests in a thread separate from the caller's (compare with Passive Object).

Active Object Pattern

A design pattern that decouples method execution from method invocation in order to enhance concurrency and simplify synchronized access to objects that reside in their own threads of control.

Architectural Pattern

A pattern that expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.

Aspects

A property of a program, such as memory management, synchronization, or fault tolerance, that cross-cuts module boundaries.

Asynchronous Completion Token (ACT)

A developer-supplied value associated with an asynchronous operation. It is used to communicate information related to the operation to the operation's completion handler.

Asynchronous I/O

A mechanism for sending or receiving data in which an I/O operation is initiated but the caller does not block waiting for the operation to complete.

Barrier Synchronization

Bibliography

[Ale01] Andrei Alexandrescu. Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley, Boston, 2001.

[All02] Paul Allen. *Model Driven Architecture*. Component Development Strategies, 12(1), January 2002.

[Aus99] Matthew H. Austern. Generic Programming and the STL: Using and Extending the C++ Standard. Addison-Wesley, Reading, MA, 1999.

[BA90] M. Ben-Ari. Principles of Concurrent and Distributed Programming. Prentice Hall International Series in Computer Science, 1990.

[Bay02] John Bay. *Recent Advances in the Design of Distributed Embedded Systems*. In Proceedings of Proceedings of SPIE, Volume 47: Battlespace Digitization and Network Centric Warfare, April 2002.

[Bec00] Kent Beck. Extreme Programming Explained: Embrace Change. Addison-Wesley, Boston, 2000.

[Ber95] Steve Berczuk. *A Pattern for Separating Assembly and Processing*. In James O. Coplien and Douglas C. Schmidt, editors, Pattern Languages of Program Design. Addison-Wesley, Reading, MA, 1995.

[BHLM94] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. *Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems*. International Journal of Computer Simulation, Special Issue on Simulation Software Development Component Development Strategies, >4, April 1994.

[Bja00] Bjarne Stroustrup. The C++ Programming Language, Special Edition. Addison-Wesley, Boston, 2000.

[BL88] Ronald E. Barkley and T. Paul Lee. *A Heap-based Callout Implementation to Meet Real-time Needs.* In Proceedings of the USENIX Summer Conference, pages 213?22. USENIX Association, June 1988.

[Bla91] U. Black. OSI: A Model for Computer Communications Standards. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[BM98] Gaurav Banga and Jeffrey C. Mogul. *Scalable Kernel Performance for Internet Servers under Realistic Loads*. In Proceedings of the USENIX 1998 Annual Technical Conference, New Orleans, LA, June 1998. USENIX.

[Boo94] Grady Booch. Object Oriented Analysis and Design with Applications, 2nd Edition. Benjamin/Cummings, Redwood City, CA, 1994.