
Overview of ACE

The [ADAPTIVE](#) Communication Environment (ACE) is a freely available, [open-source](#) object-oriented (OO) framework that implements many core [patterns](#) for concurrent communication software. ACE provides a rich set of reusable C++ [wrapper facades](#) and framework components that perform common communication software tasks across a range of OS platforms. The communication software tasks provided by ACE include [event demultiplexing and event handler dispatching](#), [signal handling](#), [service initialization](#), [interprocess communication](#), shared memory management, [message routing](#), [dynamic \(re\)configuration of distributed services](#), [concurrent execution and synchronization](#).

ACE is targeted for developers of high-performance and real-time communication services and applications. It simplifies the development of OO network applications and services that utilize interprocess communication, event demultiplexing, explicit dynamic linking, and concurrency. In addition, ACE automates system configuration and reconfiguration by dynamically linking services into applications at run-time and executing these services in one or more processes or threads.

ACE continues to improve and its [future](#) is bright. ACE is supported commercially by [multiple companies](#) using an [open-source](#) business model. In addition, many members of the [ACE development team](#) are currently working on building *The ACE ORB* ([TAO](#)).

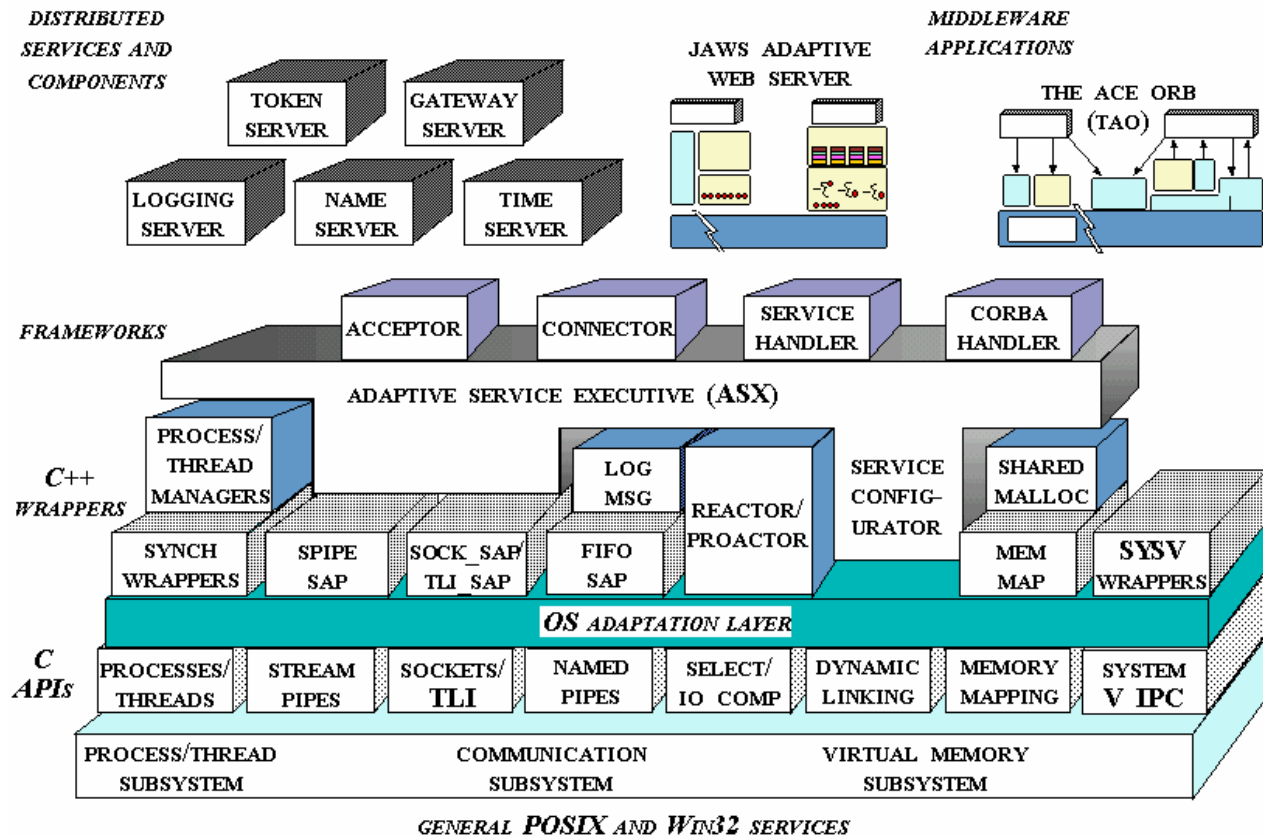
Benefits of Using ACE?

Some of the many benefits of using ACE include:

- **Increased portability** -- ACE components make it easy to write concurrent networked applications on one OS platform and quickly port them to [many other](#) OS platforms. Moreover, because ACE is [open source](#), [free](#) software, you never have to worry about getting locked into a particular operating system platform or compiler configuration.
 - **Increased software quality** -- ACE components are designed using many key [patterns](#) that increase key qualities, such as flexibility, extensibility, reusability, and modularity, of communication software.
 - **Increased efficiency and predictability** -- ACE is carefully designed to support a wide range of application quality of service (QoS) requirements, including low latency for delay-sensitive applications, high performance for bandwidth-intensive applications, and predictability for real-time applications.
 - **Easier transition to standard higher-level middleware** -- ACE provides the reusable components and patterns used in The ACE ORB ([TAO](#)), which is an open-source standard-compliant implementation of CORBA that's optimized for high-performance and real-time systems. Thus, ACE and TAO are designed to work well together in order to provide comprehensive middleware solutions.
-

The Structure and Functionality of ACE

The following diagram illustrates the key components in ACE and their hierarchical relationships:



The structure and participants of the layers in this diagram are described below.

The ACE OS Adapter Layer

This layer resides directly atop the native OS APIs that are written in C. It provides a [small footprint](#), "POSIX-like" OS adaptation layer that shields the other layers and components in ACE from platform-specific dependencies associated with the following OS APIs:

- **Concurrency and synchronization** -- ACE's adaptation layer encapsulates OS APIs for multi-threading, multi-processing, and synchronization.
- **Interprocess communication (IPC) and shared memory** -- ACE's adaptation layer encapsulates OS APIs for local and remote IPC and shared memory.
- **Event demultiplexing mechanisms** -- ACE's adaptation layer encapsulates OS APIs for synchronous and asynchronous demultiplexing I/O-based, timer-based, signal-based, and synchronization-based events.
- **Explicit dynamic linking** -- ACE's adaptation layer encapsulates OS APIs for explicit dynamic linking, which allows application services to be configured at installation-time or run-time.
- **File system mechanisms** -- ACE's adaptation layer encapsulates OS file system APIs for manipulating files and directories.

The portability of ACE's OS adaptation layer enables it to run on a many operating systems. ACE has been [ported](#) and tested on a wide range of OS platforms including [Windows](#) (i.e., WinNT 3.5.x, 4.x, 2000, Embedded NT, XP, Win95/98, and WinCE using MSVC++, Borland C++ Builder, and IBM's Visual Age on 32- and 64-bit Intel and Alpha platforms), [Mac OS X](#), most versions of UNIX (e.g., [Solaris](#) on SPARC and Intel, [SGI IRIX](#) 5.x and 6.x, DG/UX, [HP-UX](#) 10.x, and 11.x, [Tru64UNIX](#) 3.x and 4.x, [AIX](#) 3.x, 4.x, 5.x, DG/UX, UnixWare, [SCO](#), and freely available UNIX implementations, such as [Debian Linux](#) 2.x, [RedHat Linux](#) 5.2, 6.x, 7.x, 8.x, and 9.x, as well as the various Enterprise editions, [SUSE Linux](#) 8.1 and 9.2, [Timesys Linux](#), [FreeBSD](#), and [NetBSD](#)),

real-time operating systems (e.g., [LynxOS](#), [VxWorks](#), [ChorusOS](#), [QnX Neutrino](#), [RTEMS](#), [OS9](#), and [PSoS](#)), [OpenVMS](#), [MVS OpenEdition](#), and [CRAY UNICOS](#). A single [source tree](#) is used for all these platforms. There is also a [Java](#) version of ACE.

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

C++ Wrapper Facades for OS Interfaces

It is possible to program highly portable C++ applications directly atop ACE's OS adaptation layer. However, most ACE developers use the C++ wrapper facade layer shown in the figure above. The ACE C++ wrapper facades simplify application development by providing typesafe C++ interfaces that encapsulate and enhance the native OS concurrency, communication, memory management, event demultiplexing, dynamic linking, and file system APIs. Applications can combine and compose these wrappers by selectively inheriting, aggregating, and/or instantiating the following components:

- **Concurrency and synchronization components** -- ACE abstracts native OS multi-threading and multi-processing mechanisms like mutexes and semaphores to create higher-level OO [concurrency abstractions](#) like [Active Objects](#) and Polymorphic Futures.
- **IPC and filesystem components** -- The ACE C++ wrappers encapsulate local and/or remote [IPC](#) mechanisms, such as sockets, TLI, UNIX FIFOs and STREAM pipes, and Win32 Named Pipes. In addition, the ACE C++ wrappers encapsulate the OS filesystem APIs.
- **Memory management components** -- The ACE memory management components provide a flexible and extensible abstraction for managing dynamic allocation and deallocation of interprocess shared memory and intraprocess heap memory.

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

For instance, the use of C++ improves application robustness because the C++ wrappers are strongly typed. Therefore, compilers can detect type system violations at compile-time rather than at run-time. In contrast, it is not possible to detect typesystem violations for C-level OS APIs, such as sockets or filesystem I/O, until run-time.

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

Frameworks

ACE also contains a higher-level network programming framework that integrates and enhances the lower-level C++ wrapper facades. This framework supports the dynamic configuration of concurrent distributed services into applications. The framework portion of ACE contains the following components:

- **Event demultiplexing components** -- The ACE [Reactor](#) and [Proactor](#) are extensible, object-oriented demultiplexers that dispatch application-specific handlers in response to various types of I/O-based, timer-based, signal-based, and synchronization-based events.
- **Service initialization components** -- The ACE [Acceptor and Connector](#) components decouple the active and passive initialization roles, respectively, from application-specific tasks that communication services perform once initialization is complete.
- **Service configuration components** -- The ACE [Service Configurator](#) supports the configuration of applications whose services may be assembled dynamically at installation-time and/or run-time.
- **Hierarchically-layered stream components** -- The ACE [Streams components](#) simplify the development of communication software applications, such as user-level protocol stacks, that are composed of hierarchically-layered services.

- **ORB adapter components** -- ACE can be integrated seamlessly with single-threaded and multi-threaded [CORBA](#) implementations via its [ORB adapters](#).

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

Distributed Services and Components

In addition to its OS adaptation layer, C++ wrapper facades, and framework components, ACE provides a standard library of [distributed services](#) that are packaged as self-contained components. Although these service components are not strictly part of the ACE framework library, these service components play two roles in ACE:

1. **Factoring out reusable distributed application building blocks** -- These service components provide reusable implementations of common distributed application tasks such as naming, event routing, logging, time synchronization, and network locking.
2. **Demonstrating common use-cases of ACE components** -- The distributed services also demonstrate how ACE components like Reactors, Service Configurators, Acceptors and Connectors, Active Objects, and IPC wrappers can be used effectively to develop flexible, efficient, and reliable communication software.

Higher-level Distributed Computing Middleware Components

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

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

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

- Authentication, authorization, and data security.
- Service location and binding.
- Service registration and activation.
- Demultiplexing and dispatching in response to events.
- Implementing message framing atop bytestream-oriented communication protocols like TCP.
- Presentation conversion issues involving network byte-ordering and parameter marshaling.

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

1. **The ACE ORB (TAO)** -- [TAO](#) is a real-time implementation of CORBA built using the framework components and patterns provided by ACE. TAO contains the network interface, OS, communication protocol, and CORBA middleware components and features. TAO is based on the standard OMG CORBA reference model, with the enhancements designed to overcome the shortcomings of conventional ORBs for high-performance and real-time applications. TAO, like ACE, is freely available, [open source](#) software.
2. **JAWS** -- [JAWS](#) is a high-performance, adaptive Web server built using the framework components and patterns provided by ACE. JAWS is structured as a *framework of frameworks*. The overall JAWS framework contains the following components and frameworks: an Event Dispatcher, Concurrency Strategy, I/O Strategy, Protocol Pipeline, Protocol Handlers, and Cached Virtual Filesystem. Each framework is structured as a set of collaborating objects implemented by combining and extending components in ACE. JAW is also freely available, open-source software.

Back to the [ACE](#) home page.

Last modified 08:38:10 CDT 22 June 2011