

ACE: Desarrollo Multiplataforma de Aplicaciones en Red

Índice

Arquitectura.....	3
Capa de Adaptación al Sistema Operativo.....	3
Capa de envoltorios C++ de ACE o wrapper facades.....	3
Concurrencia y sincronización.....	5
Comunicación mediante Mecanismos IPC.....	5
Manejo de Memoria.....	5
Contadores de Tiempo o “timers”.....	6
Contenedores.....	6
Capa de Frameworks de ACE.....	6
Concepto de Patrón de Diseño.....	6
Concepto de Framework	7
Frameworks Reactor y Proactor.....	7
Configuración de Servicio.....	8
Control de Tareas y Procesos.....	9
Acceptor y Connector.....	10
Streams.....	11
Beneficios según la experiencia al usar ACE.....	11
¿Cómo aprender a utilizar ACE?.....	12

Introducción

ACE se corresponde con las siglas de ADAPTIVE Communication Environment, y es un marco de trabajo – *Framework* – orientado a objeto, de código abierto y disponible libre y gratuitamente, que implementa muchos de los patrones centrales para el desarrollo de software de comunicación.

ACE está implementado en C++ y se ha orientado desde sus principios a desarrolladores de aplicaciones y servicios de alto rendimiento en red y tiempo real. Para el equipo que lo comenzó, y que sigue desarrollando y aumentando su solidez y sus funcionalidades, ha sido siempre una obsesión la sencillez de uso, la reusabilidad de código entre aplicaciones y entre plataformas sin dejar de lado el rendimiento.

La curva de aprendizaje de un *Framework* siempre es inclinada en sus inicios, pero con ACE merece la pena el esfuerzo. Una vez adquiridos los conocimientos sobre cómo usar sus componentes, desarrollar un servicio de red como *ftp*, un *gateway* o un servicio de mensajes, puede ser cuestión de horas, si no minutos, y además sin estar atados a ninguna plataforma, sistema operativo o implementación del compilador.

Los comienzos de ACE se remontan a las actividades de investigación y desarrollo del doctorado del ahora profesor Douglas Schmidt en la Universidad de California en Irvine, alrededor de los patrones de diseño y la implementación y el análisis de técnicas orientadas a objeto que facilitarían el desarrollo de *Frameworks* para aplicaciones distribuidas en tiempo real de alto rendimiento.

ACE se creó para resolver dos frustraciones comunes en el día a día del desarrollo de aplicaciones distribuidas en red. La primera frustración, la necesidad de cambiar de un código C++ estructurado, reusable y encapsulado, a código en C para el acceso a los recursos de los diferentes sistemas operativos tales como procesos, hilos, *sockets*, memoria compartida, *DLLs*, y ficheros. En segundo lugar, el hecho demostrable de que se reusa poco el código ya hecho; tendemos a realizar desde cero el mismo código encargado del establecimiento de conexiones, de la demultiplexación síncrona de eventos, de la sincronización y las arquitecturas para manejar la concurrencia.

Introducción a ACE - Teoría

A estas alturas ACE es un proyecto de software muy consolidado, que tiene detrás al Grupo de Computación de Distribuida orientada a Objeto (DOC) de la Universidad de Washington. ACE se utiliza en numerosas aplicaciones comerciales, ha sido portado a docenas de plataformas, y cuenta, además de con soporte comercial, con una comunidad abierta muy activa e infinidad de artículos de investigación publicados en las revistas más importantes del sector.

Hoy en día no tiene sentido desarrollar software y atarse a un proveedor o a una plataforma concreta. Cualquier programador, jefe de proyecto o plan de negocio desearía que su software sea capaz de correr en un Linux, un Windows CE o una máquina Solaris siempre que nos evitemos los tediosos esfuerzos de mantenimiento cruzado o los problemas de compatibilidad. (Casi) todo software debe tener como uno de sus principales objetivos ser independiente de plataformas, sea hardware o software, y sistemas operativos, y no verse comprometido con ningún fabricante o proveedor en concreto. Algunas de las soluciones más comunes del mercado son:

- **Sun Java Virtual Machine (JVM)**, que proporciona una máquina virtual responsable de interpretar el código Java y traducirlo al lenguaje de la máquina real en la que reside. De esta manera las aplicaciones ven de forma transparente el sistema operativo subyacente.
- **Microsoft Common Lenguaje Runtime (CLR)**, que es la infraestructura sobre la que están contruidos los servicios Web de Microsoft .NET. Es similar a JVM
- **ADAPTIVE Communication Environment (ACE)**, disponible de forma gratuita, con acceso a su código y altamente portable entre plataformas. Es un entorno y un conjunto de herramientas escritas en C++, que separa el sistema operativo de la aplicación a través de la capa de Adaptación al Sistema Operativo.

Las principales diferencias entre ACE, JVM y CRL .NET son:

- ACE siempre genera un código compilado para la plataforma destino en lugar de un código intermedio que debe interpretarse después, eliminando un nivel de direccionamiento y optimizando el rendimiento en ejecución.
- ACE es Código Abierto, por lo que es posible adaptarlo o usar sólo un subconjunto en el que estemos interesados. Java en los últimos tiempos se ha unido también a la filosofía de código abierto.
- ACE es capaz de ejecutarse en más sistemas operativos y hardware que JVM y CLR.

Ejemplos de plataformas en las que corre ACE actualmente son:

- Para PC:
Todas los Windows de 32 y 64 bits, WinCE, RedHat, Debian, Suse...
- UNIX:
Solaris, AIX, SGI IRIX, HP-UX, FreeBSD, NtBSD,...
- Sistemas de Tiempo Real:
VxWorks , LynxOS, OS/9, Pharlap TNT, Chorus, QNX Neutrino, ...
- Grandes Sistemas:
OpenVMS, MVS OpenEdition, Tandem.

ACE consta a día de hoy con más de 320.000 líneas de código, con más de 500 clases, mantenidas por desarrolladores habituales y es usado por empresas y en proyectos comerciales y actuales.

Para más información:

<http://www.cs.wustl.edu/~schmidt/ACE-overview.html>

Arquitectura

ACE proporciona mucho más que una simple librería de clases y funciones. En primera instancia proporciona un API de acceso a los recursos de los sistemas operativos, un conjunto de clases que encapsulan dichas APIs, varios Frameworks e incluso servicios y aplicaciones distribuidas en red listos para ser usados directamente.

Para separar las diferentes áreas y reducir la complejidad, ACE está diseñado utilizando una arquitectura en capas, desde las más cercanas al sistema operativo hasta las de más alto nivel, como podemos ver en la *Ilustración 1*.

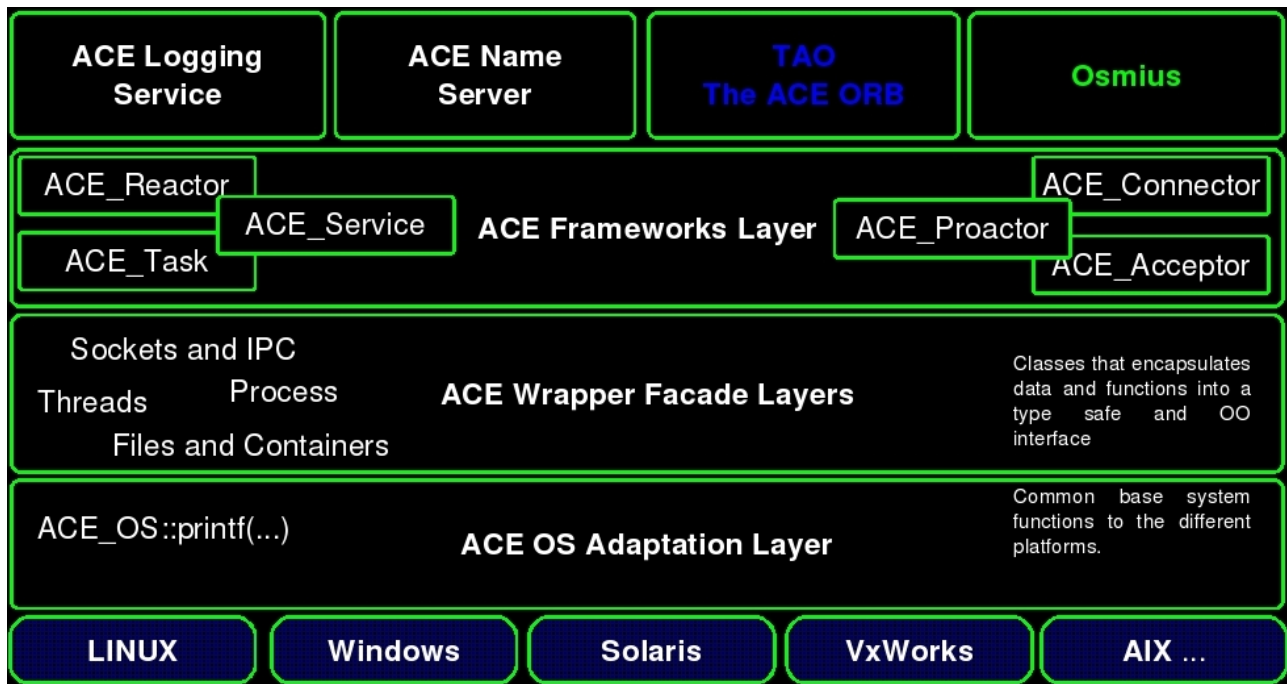


Ilustración 1: Arquitectura en Capas de ACE

Veamos las características principales de cada una de estas capas y las funcionalidades que nos ofrecen como programadores de aplicaciones en red.

Capa de Adaptación al Sistema Operativo.

Esta capa proporciona un conjunto de funciones para las operaciones y las llamadas al sistema operativo más comunes y se sitúa entre las APIs nativas de llamadas al sistema y el resto de ACE. Constituye aproximadamente un diez por ciento del total de ACE, y consiste en una clase llamada `ACE_OS` que alberga más de 500 métodos estáticos de C++. Son estos métodos los que encapsulan las APIs de sistema y nos ocultan los detalles específicos de cada plataforma, proporcionándonos un interfaz uniforme que podemos usar directamente como hacen las capas más altas de la arquitectura.

Mediante la Capa de Adaptación `ACE_OS`, nos aseguramos la portabilidad y la mantenibilidad de nuestro código. Son los desarrolladores de ACE los que bregan con las complejidades y las diferencias de los Sistemas Operativos, y no nosotros como desarrolladores de aplicaciones. Gracias a esta capa nuestro código basado en ACE pueden migrarse de una plataforma a otra sin esfuerzo, y es la responsable de que el conjunto de ACE esté disponible para tantos sistemas operativos diferentes.

Capa de envoltorios C++ de ACE o wrapper facades

Se pueden programar aplicaciones altamente portables utilizando la capa anterior, pero esta

Introducción a ACE - Teoría

capa simplifica el desarrollo de aplicaciones encapsulando y mejorando los servicios del Sistema Operativo con interfaces robustos usando los tipos del lenguaje C++.

Por definición un *Wrapper Facade* consiste en una o más clases que encapsulan funciones y datos dentro de un interfaz con comprobación de tipos y orientado a objeto. En realidad la funcionalidad proporcionada por esta capa es casi la misma que la de la capa de adaptación al sistema operativo. Donde reside su utilidad es en el empaquetado en clases C++ de las funciones aisladas estilo C de la capa anterior, de forma que se reduce significativamente el esfuerzo de aprender y utilizar ACE correctamente.

Esta capa envuelve a la anterior proporcionándonos, ya sí, un interfaz orientado a objeto. Además estos envoltorios han sido cuidadosamente diseñados para eliminar o minimizar la posible pérdida de rendimiento debida a la mejora en la facilidad de utilización y al control de tipos.

Para implementar los ACE wrapper facades se aplicaron los siguientes criterios de diseño:

- Mejorar la seguridad de los tipos de datos.
- Simplificar siempre hacia el caso más común.
- Utilizar jerarquías para mejorar la claridad y la capacidad de extenderse.
- Ocultar las diferencias entre plataformas en lo posible.
- Optimización encaminada hacia la eficiencia.

Podemos utilizar ACE en cualquiera de sus capas, pero es en ésta donde realmente comenzamos a beneficiarnos de las ventajas del conjunto, y los beneficios continúan según avanzamos en capacidad de abstracción. Al usar esta capa, las operaciones no permitidas con los tipos de datos, se detectan en tiempo de compilación en lugar de hacerlo en tiempo de ejecución como ocurría al utilizar APIs en C.

Veamos un ejemplo de uso de las *wrappers facades* y sus ventajas.

Problema: Las APIs de funciones C para llamadas al sistema contienen a menudo un montón de funciones que existen sólo para dar soporte a casos relativamente poco frecuentes. Por ejemplo, el API para manejo de *sockets* está preparado para funcionar con muchas familias diferentes de protocolos como TCP/IP, IPX/SPX, X.25, ISO OSI y *sockets* del dominio UNIX.

Para poder dar soporte a tal variedad de familias de protocolos, los diseñadores originales del API definieron funciones C separadas para

- 1.- Crear el descriptor del *socket* (*socket handle*).
- 2.- Enlazar el descriptor con un extremo de la comunicación (hacer el *bind*).
- 3.- Marcar un extremo de la comunicación como un *factory* en modo pasivo.
- 4.- Para aceptar una conexión y devolver un descriptor para los datos.

De esta forma si queremos crear e inicializar un *socket* de dominio internet en modo pasivo para recibir datos (algo tan común como ponernos a escuchar datos en un puerto) teníamos que hacer un montón de llamadas:

```
sockaddr_in  server_address;
int          address_len;
int          socket_handle;
int          network_handle;

// Inicializamos direccion...
memset (&server_address, 0, address_len);
address_len = sizeof(sockaddr_in);

// Inicializamos descriptor de socket...
s_handle = socket (PF_INET, SOCK_STREAM, 0);

// Definimos familia, puerto y tipo de conexiones que aceptamos.
server_address.sin_family      = AF_INET;
server_address.sin_port       = htons (999); // Puerto de escucha.
server_address.sin_addr.s_addr= INADDR_ANY;
```

Introducción a ACE - Teoría

```
// Enlazamos el descriptor del socket con la dirección de red.
bind (socket_handle, &server_address, address_len);

// Comenzamos a escuchar.
listen (socket_handle);

.....

// Aceptamos conexiones.
n_handle = accept (socket_handle, &server_address, &address_len);
```

Este código o uno muy similar se reescribía para cada aplicación basada en comunicaciones TCP/IP ya fuera para conexiones pasivas como la anterior o para las activas o iniciadas por nuestros procesos. ACE nos permite descongestionar y simplificar nuestro código con lo que la reutilización aumenta y la tentación de reescribir estas partes del código es mucho menor. *ACE_SOCK_Acceptor* en un *factory* para establecimiento pasivo de conexiones, cuyo método *open()* hace las llamadas oportunas a *socket()*, *bind()* y *listen()*, para crear el extremo de comunicación en modo pasivo. Así a una aplicación basada en ACE le bastaría con lo siguiente para conseguir lo que hemos visto anteriormente:

```
ACE_SOCK_Acceptor    acceptor;    // Extremo de comunicación pasivo.
ACE_SOCK_Streamstream;    // Canal de comunicación.

ACE_INET_Addr       server_address(999); // Dirección local en puerto 999.

acceptor.open      (server_address);
acceptor.accept    (stream);
```

Este código como vemos es mucho más sencillo e intuitivo. Además el constructor de *ACE_INET_Addr* se encarga de tareas que reducen la posibilidad de fallos y olvidos en la programación, ya que automáticamente inicializa la estructura interna *sockaddr_in* (*inet_addr_*) y convierte el número de puerto al ordenamiento de "bytes en red" (*htons()* -> "host to network server").

Veamos las diferentes áreas en las que se agrupan las *ACE wrapper facades*.

Concurrencia y sincronización:

Son clases que nos abstraen de las APIs del sistema operativo para programación multihilo y programación multiproceso. Además estos envoltorios encapsulan las funcionalidades primitivas de sincronización como semáforos, barreras, variables de condición y bloqueos de ficheros. Estas primitivas comparten interfaces de uso similares con lo que es relativamente fácil, e incluso directo, intercambiar unas por otras.

Comunicación mediante Mecanismos IPC:

ACE proporciona clases que encapsulan los mecanismos de comunicación entre procesos tales como *sockets BSD*, colas *UNIX FIFO*, *STREAM* o los "named" pipes de *Windows*. Además tenemos clases para manejo de colas de mensajes y *wrapper facades* para algunos tipos de colas de mensajes de tiempo real de sistemas operativos específicos.

Manejo de Memoria:

Esta capa incluye clases para reservar y liberar memoria de forma dinámica y también para la prereserva de memoria dinámica. Esta memoria se maneja localmente en ACE a través de las clases suministradas. El acceso de grano fino a la memoria es necesario en la mayoría de los sistemas empotrados de tiempo real. También hay clases para la gestión flexible de memoria compartida entre procesos.

Contadores de Tiempo o “*timers*”:

Disponemos de clases para programar y cancelar alarmas en el tiempo. Hay diferentes tipo de contadores que por debajo utilizan diferentes mecanismos (pilas o montones, ruedas de tiempos o listas ordenadas) para que podamos elegir en función del rendimiento deseado. Algo que ocurre con muchas clases y frameworks en ACE y que es altamente deseable es que independientemente de la implementación que elijamos los interfaces se mantienen iguales, lo que hace mucho más fácil y transparente el uso de una u otra implementación. También contamos con los *timers* de alta resolución que están disponibles en plataformas como VxWorks, Windows, AIX y Solaris.

Contenedores:

ACE también incluye varias clases tipo contenedor estilo *Standar Library STL* como *Map*, *Hash_Map*, *Set*, *List* y *Array*.

Manejo de Señales:

Tenemos clases envoltorio que encapsulan los diferentes interfaces que tienen los sistemas operativos para la gestión de señales (como un Control-C de usuario o un *kill* de sistema). Simplifican la instalación o cancelación de manejadores de señales y permiten instalar varios manejadores para la misma señal.

Sistema de Ficheros

Estas clases envuelven las diferentes funciones del API de gestión de los sistemas de ficheros, incluyendo la entrada/salida por fichero, E/S asíncrona, bloqueo de ficheros, *streams*, etc.

Hilos:

Programar con hilos puede ser un infierno sobretodo si tenemos que hacer un código portable entre diferentes plataformas. ACE proporciona clases para crear y controlar hilos de manera, uniforme y muy sencilla. Estas *wrapper facades* encapsulan las diferencias específicas de cada sistema operativo, y podemos usarlas para tener acceso a funcionalidades avanzadas como el disponer de almacenamiento específico para cada hilo o *thread*.

Capa de Frameworks de ACE

Los componentes de esta capa son el más alto nivel disponible como bloques para la construcción de aplicaciones. Estos componentes se basan en patrones de diseño específicos del dominio de software de comunicación distribuido. En general los frameworks de ACE facilitan el desarrollo de software de comunicación que puede ser actualizado y extendido en funcionalidad sin tener que modificar, recompilar, reenlazar e incluso sin volver a lanzar los procesos. Este alto grado de flexibilidad se consigue en ACE combinando las posibilidades que nos ofrece el lenguaje C++ como la herencia, la sobrecarga de métodos, el uso de plantillas y el enlace dinámico con la aplicación e implementación particular de patrones de diseño.

Antes de seguir repasemos por un momento los conceptos de Framework y Patrón de Diseño.

Concepto de Patrón de Diseño

Christopher Alexander, arquitecto de edificios reconocido como el originador de la idea de los patrones, explica que “cada patrón en una regla con tres integrantes que expresa la relación entre cierto contexto, un problema y una solución a dicho problema”.

Un desarrollador de middlewares, frameworks o aplicaciones específicas se tiene que enfrentar a retos relacionados con el diseño y la programación de asuntos tales como:

La persistencia de los datos, la organización de los datos, la gestión de las conexiones, inicialización de servicio y objetos, el manejo de los parámetros de configuración, la comunicación entre estructuras, **la concurrencia**, el control de acceso a los recursos, control del

Introducción a ACE - Teoría

flujo de programa, la gestión de errores y de información de *log*, bucles de gestión de eventos, etc.

La resolución de estos retos se presenta una y otra vez, e inicialmente el conocimiento necesario para resolverlos estaba en las mentes de algunos desarrolladores o, de forma implícita y poco apprehensible, en el código. Los patrones de diseño tratan de solventar este problema.

Concepto de Framework

Las características deseables para un producto software son variadas y muy necesarias. Un software debe ser extensible, modular, robusto y eficiente. Conseguir estas características es complejo, y los frameworks aportan la tecnología que trata de abordar esta complejidad incrementando los niveles de reusabilidad del código.

Un framework es una colección de componentes software que colaboran entre sí para proporcionar una arquitectura reusable para una determinada familia de aplicaciones.

Los componentes de un framework son clases (gestores de mensajes, manejadores de eventos, mapas de conexiones,...), jerarquías de clases, categorías de clases (familias de mecanismos de control) y objetos. Un componente es una unidad de encapsulación de servicios con uno o más interfaces a través de los cuales se proporciona acceso a los servicios que ofrece. También podemos definir un framework como un diseño reusable, de una parte o de un todo de un sistema, representado por un conjunto de clases abstractas y la forma en que interaccionan sus instancias.

Frameworks Reactor y Proactor

Muchas aplicaciones de comunicaciones se desarrollan como programas orientados a evento. Ocurren cosas, como un cliente que se quiere conecta o que otro envía un mensaje pidiendo información sobre el estado de la bolsa, a las que debo reaccionar con determinada acción.

Los tipos de evento más comunes incluyen la actividad en los mecanismos *IPC* para operaciones de Entrada/Salida, señales *POSIX*, gestión de señales en Windows y la expiración de contadores de tiempo.

El *framework Reactor* está orientado al tratamiento síncrono de eventos, mientras que Proactor lo está para el asíncrono. Nos centraremos sólo en el *framework Reactor* que es uno de los más utilizados en el software de comunicación.

El framework nos permite separar los conceptos de detección, demultiplexación y despacho de eventos, de su tratamiento. Implementa el patrón *Reactor*.

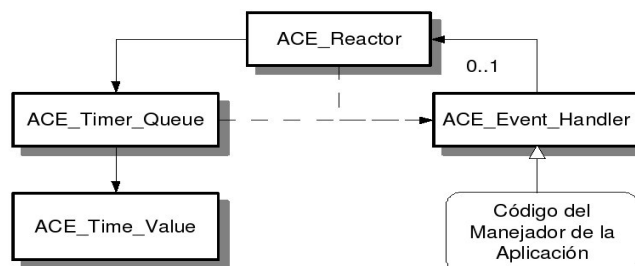


Ilustración 2: Framework Reactor

Se automatizan las siguientes tareas:

- Detección de eventos desde varios orígenes.
- Demultiplexación de eventos hacia los manejadores previamente registrados de esos

Introducción a ACE - Teoría

eventos.

- Despacho a los métodos gancho (*hook methods*) definidos por los manejadores para procesar los eventos como definen los requisitos de la aplicación concreta que se está implementando.
- Portabilidad

Los beneficios de uso del patrón *Reactor* son:

- Separación de Problemas.
- Modularidad, reusabilidad y facilidad de configuración.
- Control de Concurrencia.

La clase principal de este framework es *ACE_Reactor* e implementa el patrón *Facade* para definir el interfaz para las capacidades del framework:

- Centraliza el bucle de tratamiento de eventos en aplicaciones reactivas.
- Detecta los eventos a través del demultiplexor proporcionado por el SO y usado en la implementación concreta del *reactor*.
- Despacha a los métodos gancho (*hook methods*) de los manejadores de eventos para que procesen según el tratamiento definido por la aplicación.
- Garantiza que cualquier hilo pueda cambiar el conjunto de eventos o encole una llamada a un manejador, y que pueda esperar una respuesta a su petición de forma rápida.

Este framework ha sido diseñado para que sea posible extenderlo y se ha separado el interfaz de la implementación utilizando el patrón *Bridge*. Hay más de una decena de implementaciones diferentes del *Reactor* en ACE siendo las más comunes éstas:

- **ACE_Select_Reactor**

Utiliza la función síncrona de demultiplexación de eventos *select()* para detectar eventos de E/S y expiración de contadores de tiempo. Incorpora también señales POSIX.

- **ACE_TP_Reactor**

Usa el patrón *Leader/Follower* para extender el manejo de eventos de la implementación anterior usando un Grupo de Hilos (*Thread Pool Reactor*).

- **ACE_WFMO_Reactor**

Mediante la función de Windows *WaitForMultipleObjects()* para la demultiplexación de eventos detecta eventos de E/S, *timeouts*, y eventos de sincronización de Windows.

Es casi directo el realizar una nueva implementación del *Reactor* en nuestros programas reutilizando elementos e integrarla en este entorno.

Configuración de Servicio

Este framework (*ACE Service Configurator*) implementa el patrón *Configurator* y permite a las aplicaciones retrasar las decisiones de implementación y configuración de los servicios que ofrecen hasta más tarde dentro del ciclo de diseño de software, incluso en tiempo de instalación o de ejecución.

Tiene la habilidad de activar servicio de forma selectiva en tiempo de ejecución estén los servicios enlazados estáticamente o dinámicamente.

Gracias a que ACE tiene integrados los diferentes framework posibilita que los servicios diseñados usando este entorno puedan ser despachados por el framework ACE *Reactor*.

Introducción a ACE - Teoría

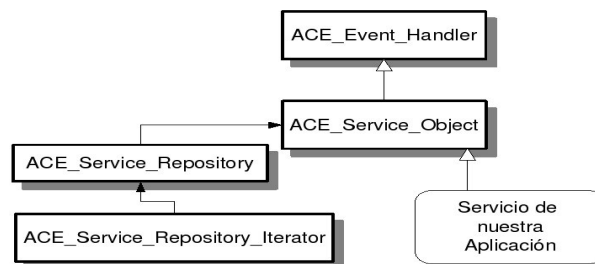


Ilustración 3: ACE_Service

Los beneficios de uso de ACE Service Configurator son:

- **Uniformidad**
Mismo interfaz de configuración y control de los servicios.
- **Administración Centralizada**
Agrupamos componentes y funcionalidad en una unidad administrativa lo que simplifica el desarrollo.
- **Modularidad y Reusabilidad**
Separamos la implementación de componentes de la forma en que dichos componentes se configuran en procesos.
- **Dinamismo en la Configuración y el Control**
Permitimos que los servicios puedan configurarse dinámicamente sin necesidad de recompilar o codificar.

Control de Tareas y Procesos

El framework *ACE Task* proporciona capacidades muy potentes en el ámbito de la concurrencia de procesos orientada a objetos, y nos permite lanzar varios hilos simultáneamente en el contexto de un objeto (*Active Object*).

También nos permite el paso de mensajes directo o través de colas entre objetos ejecutándose en hilos diferentes.

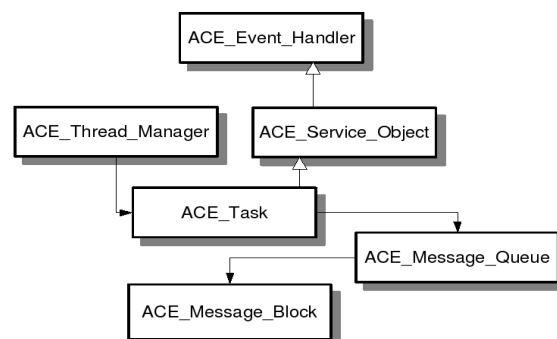


Ilustración 4: Framework ACE_Task

La clase *ACE_Task* contiene una cola de mensajes (*ACE_Message_Queue*) que se puede utilizar

Introducción a ACE - Teoría

para separar el flujo de información de su procesamiento y para enlazar hilos que ejecutan servicios productores y consumidores de mensajes concurrentemente (Patrón *Producer/Consumer*).

Los beneficios de usar `ACE_Task` son:

- Mejora de la seguridad en los tipos de datos
- Mejora de la concurrencia problemas de sincronización simplificados.
Se permite que los hilos ejecuten métodos asíncronos de forma simultánea. La sincronización se simplifica utilizando un planificador que evalúa las restricciones de sincronización.
- Aprovechamiento transparente del paralelismo disponible.
Varios métodos de objetos activos pueden ejecutarse en paralelo si lo permite el SO y el hardware subyacente.
- El Orden de ejecución puede diferir del orden de invocación.
Métodos llamados de forma síncrona pueden ejecutarse de forma síncrona de acuerdo a las restricciones de la aplicación. Podemos agrupar llamadas y enviarlas o procesarlas todas juntas para mejorar el rendimiento.

Acceptor y Connector

Este framework implementa el patrón *Acceptor/Connector*, que mejora la reusabilidad del software y su facilidad para extender su funcionalidad separando las tareas necesarias para la conexión e inicialización de pares servicios de red, del procesamiento que han de realizar una vez conectados e inicializados.

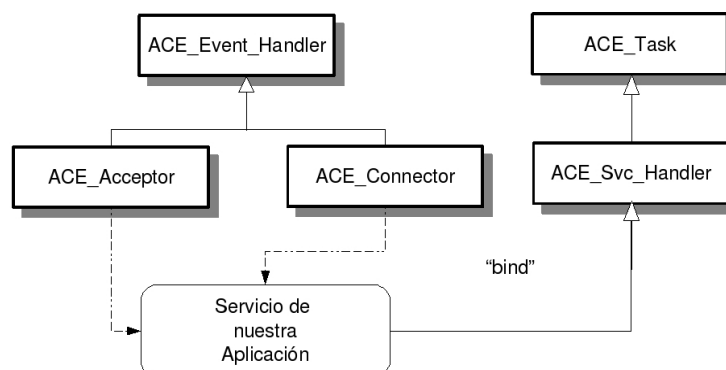


Ilustración 5: Framework Acceptor y Connector

Las clases principales son:

- **ACE_Svc_Handler**
Representar uno de los extremos de conexión de un servicio y contiene un mecanismo IPC que es utilizado para comunicarse con un compañero de conexión.
- **ACE_Acceptor**
Implementa un *factory* (creador de objetos) que espera de forma pasiva aceptando conexiones y cuando se produce una inicializa un `ACE_Svc_Handler` como respuesta a la petición activa de conexión.
- **ACE_Connector**
Este *factory* se conecta de forma activa a un compañero de conexión tipo *Acceptor* e

Introducción a ACE - Teoría

inicializa un *ACE_Svc_Handler* para comunicarse.

La clase ACE Acceptor nos ayuda a separar la conexión y la inicialización de un servicio, del tratamiento posterior de forma reusable para que los desarrolladores no tenga que reescribir este código repetidas veces.

Streams

Este framework se basa en el patrón *Pipes & Filters*, que simplifica el desarrollo de aplicaciones modulares en capas que pueden comunicarse de manera bidireccional entre los módulo de procesamiento.

Define una arquitectura para procesar una corriente de datos en la que cada paso de procesamiento está encapsulado en algún tipo de filtro o componente.

Los datos se pasan entre filtros adyacentes a través del mecanismo de comunicación definido , que puede ser desde canales IPC conectando procesos locales o remotos hasta simples punteros que referencian a objetos dentro del mismo proceso. Cada filtro puede añadir, modificar o eliminar los datos antes de pasarlos al siguiente filtro. En ocasiones los filtros carecen de información de estado, en cuyo caso el paso de los datos se hace entre los filtros sin ser almacenados.

Con este framework, ACE nos permite implementar esta estructura de forma sencilla y dinámica. Podemos añadir o quitar módulos de la cadena de tratamiento de los datos incluso en tiempo de ejecución basándonos en algunas capacidades de los frameworks anteriores. Podríamos añadir un módulo de depuración entre otros dos durante la ejecución del programa, en un hilo diferente, para ver los datos que se pasan entre ellos, y una vez depurarlo eliminarlo de la cadena.

El uso de los frameworks presentados se puede combinar dentro de la misma aplicación o servicios, y de hecho es lo que ocurre en la estructura de los programas medianamente grandes y orientados a comunicaciones en red.

Beneficios según la experiencia al usar ACE

Aumento de la Portabilidad:

ACE está preparado para generar aplicaciones en la gran mayoría de Sistemas Operativos. No tenemos que preocuparnos por quedarnos atrapados con un proveedor.

Aumento en la Calidad del Software:

Los componentes de ACE están diseñados utilizando muchos patrones de diseño. Los patrones de diseño son parejas de problema-solución probados y contrastados por muchos desarrollos y años de experiencia. Todo esto mejora cualidades como la flexibilidad, la modularidad, la reusabilidad y la solidez frente a errores del software.

Mejora de la Eficiencia:

ACE ha sido cuidadosamente diseñado para soportar un amplio espectro de requerimientos de aplicaciones en cuanto a su Calidad de Servicio (QoS). Esto incluye requisitos de bajas latencias para aplicaciones muy sensibles a retrasos en las comunicaciones en red, de alto rendimiento para aplicaciones que demandan un fuerte uso del ancho de banda, y las necesidades de predicción en aplicaciones de tiempo real.

ACE sigue evolucionando a día de hoy y sus objetivos a futuro son:

- Incremento de Solidez
- Nuevas Funcionalidades. Se añaden constantemente.
- Optimizaciones orientadas a rendimiento.
- Adición de nuevas plataformas. Sistemas Operativos Nuevos
- Mejora de Documentación.
- Mejorar la Integración. CORBA (TAO).

¿Cómo aprender a utilizar ACE?

Haberse leído este artículo puede ser un buen comienzo, pero lo recomendado es bajarse el código de la última versión beta (las "betas" de ACE son versiones extremadamente sólidas al contrario de lo que ocurre con otros productos software) de la página <http://download.dre.vanderbilt.edu> y seguir las instrucciones.

Dentro del árbol de los fuentes de ACE encontraremos varios plagados de ejemplos con los que comenzar ha hacer pruebas. Estos son los directorios "examples" y "tests", además de las carpetas por debajo de "apps" con aplicaciones ya hechas utilizando el entorno ACE.

La formación está más extendida en los EEUU pero en Europa existen, que seamos, empresas en Holanda y en Alemania capaces de dar cursos sobre ACE con toda dignidad y en España podéis ponerlos en contacto con la empresa Peopeware para estos asuntos.

Podemos utilizar los buscadores de artículos académicos para encontrar lo publicado sobre ACE, desde un punto más de investigación. Son recomendables estos dos que os citamos a continuación:

The ADAPTIVE Communication Environment An Object-Oriented Network Programming Toolkit for Developing Communication Software.

Douglas C. Schmidt - 1993
11th and 12th Sun user group conferences. California

An architectural overview of the ACE framework. A Case Study of Successful Cross-Platform Systems Software Reuse

Douglas C. Schmidt - 1999
USENIX

Además hay varios libros que abordan directamente la programación de aplicaciones en red mediante ACE. El primero de ellos está más orientado a la parte de envoltorios o "wrapper facade" y el segundo al uso de más alto nivel a través de los frameworks. El tercero puede valer como aproximación global a ACE aunque teniendo los dos primeros podría ser prescindible.

C++ Network Programming: Mastering Complexity Using ACE and Patterns

Autores: Douglas C. Schmidt y Stephen D. Huston
ISBN 0-201-60464-7 (2002)

C++ Network Programming: Systematic Reuse with ACE and Frameworks

Autores: Douglas C. Schmidt y Stephen D. Huston
ISBN 0-201-79525-6 (2003)

The ACE Programmer's Guide: Practical Design Patterns for Network and Systems Programming

Autores: Stephen D. Huston, James CE Johnson y Umar Syyid
ISBN 0-201-69971-0 (2003)