ADAPTIVE Communication Environment

Douglas C. Schmidt
schmidt@cs.wustl.edu
http://www.cs.wustl.edu/_schmidt/
Department of Computer Science
Washington University
St. Louis, MO 63130, (314) 935-7538

译者： Smile， yfang76@hotmail.com

ACE **是一个面向对象的工具开发包，它实现了通信软件的基本设计模式。** ACE **面向在** UNIX **和** Win32 **平台上开发高性能通信服务的开发人员。它简化了面向对象的网络应用程序和服务的开发，这些程序和服务用到了进程间通信，事件分离，直接动态链接和并发机制。**ACE **通过在运行时动态链接服务到应用程序和在一个或多个进程或线程中执行这些服务自动完成系统配置和重新配置。**

1 介绍：

1.1 问题：分布式软件危机

健壮性和高性能分布式计算系统正在稳定增加。这些系统包括：全球个人通信系统，网络管理平台，在线金融分析系统和实时电子航空系统等。对于需要通过连接和交互来提高协作性而言，分布式计算是一个很有前途的技术：通过并行处理了提高性能，通过复制来提高可靠性和可用性；通过模块化设计来提高可伸缩性和可移植性：通过动态配置和重新配置可以提高扩充性。

尽管分布式计算提供了许多潜在的好处，开发通信系统仍是昂贵的并且容易导致错误的。为减少软件代价和提高软件质量，面向对象的程序设计语言，组件和框架得到了广泛的吹捧。剥去吹捧的成分，面向对象的基本好处是强调模块化和可扩展性，在固定的接口内封装了实现细节，增强了软件可重用性。

开发人员在确定的领域应用面向对象技术已经取得了成功。例如 MFC GUI 框架和 OCX 组件。尽管这些技术有它们的局限性，它们仍较好的解释了重用通用框架和组件带来的效率的提高。

软件开发人员在更复杂的领域，如通信，医疗图像，航空电子和在线交易过程中缺少标准的中间件组件。结果，开发人员必须从最底层开始构建、验证和维护软件系统，这样的开发既费时又费力，在全球竞争的环境下是不允许的。联系到工业生产，这种情况下就产生了“分布式软件危机”，即用于分布计算的硬件和网络越来越小，越来越快，越来越便宜，而分布式软件却越来越大，越来越慢，开发和维护成本越来越高。

构建分布式软件的挑战来源于分布式系统固有的和次要的复杂性。固有的复杂性来源于开发分布式系统的基本挑战，主要是探测和恢复网络及主机的失败，减小通信延迟的冲击，以及通过网络确定和优化服务组件以及工作负载的分配。

次要的复杂性来源于开发通信软件的工具和技术的限制。例如，许多标准的网络机制（例如 sockets 和 TLI）和可重用组件库（如 X windows 和 Sun RPC）缺少类型安全、可移植性和可扩展的 API。同样，通用网络编程接口，例如 sockets 和 TLI 使用弱类型的整数句柄，这会导致潜在的允许错误。

另一个复杂性来源于广泛使用的算法分解，这会导致非扩展和不可重用的软件系统。尽管 OO 技术中经常构建图形用户界面，分布式软件典型的开发会应用算法分解技术。这个问题的加剧在于流行的网络编程书中的例子都是基于面向算法的设计和实现技术的。

缺少可扩展性和可重用性是大部分复杂分布式软件的问题。可扩展性主要是为确保服务和特性能够随着时间推移能够进行改变和增强。可重用性主要是平衡专业开发人员的知识域，以避免重新开发和验证通用的解决办法。

1.2 解决办法：面向对象的设计模式和框架

由于面向对象的设计模式和框架能够帮助减轻昂贵的分布式软件概念和抽象核心的重新发现和重新生成，被人们所看好。模式提供了一种对设计知识的封装方法，此方法提供了标准分布式软件开发的一种解决。例如，模式可用于描述微体系结构的 recurring （例如 Reactor 和 Active Object），微体系结构是通用对象结构的抽象，在分布式通信软件的构建过程中被证明是有用的。然而，抽象的模式文档不能直接生成可重用的代码。因此，加强对模式的研究以及框架的生成和使用是必需的。

框架为应用程序提供了可重用的软件构件，通过集成抽象类集和定义这些类的实例协作的标准方法。框架实例化设计模式 families 以帮助开发人员避免昂贵的通用分布式软件构件的重新生成。结果产生了“准完全”的应用程序骨架，这个骨架可以通过从框架中可重用的构建块组件继承和实例化进行定制。由于框架与关键的分布式编程任务是紧密结合的（例如服务初始化，错误处理，流控制，事件分离，并发控制），可重用的范围比使用传统函数库或 OO 类库大的多。

本论文如下组织：第2部分给出了ACE的结构和函数；第3部分描述了ACE C++包组件和高级ACE框架组件和模式的详细内容；第4部分给出了几个构建于ACE的网络应用程序的实现；第5部分是结论。

2 ACE 概述

为了解释OO模式和框架是如何被应用到分布式软件的，本文研究了ACE。ACE是一个免费的OO工具包，它包含一个丰富的集合，此集合中有可重用的wrappers、类、和可在很多操作系统平台上交互的通用的网络编程的框架。ACE提供的任务包括：

- 事件分离和事件处理程序的调度
- 连接的建立和服务的初始化
- 交互通信和共享内存管理
- 动态配置和分布式通信服务
- 并发/并行和同步
- 高级分布式服务组件（例如名字服务，事件服务，日志服务等）

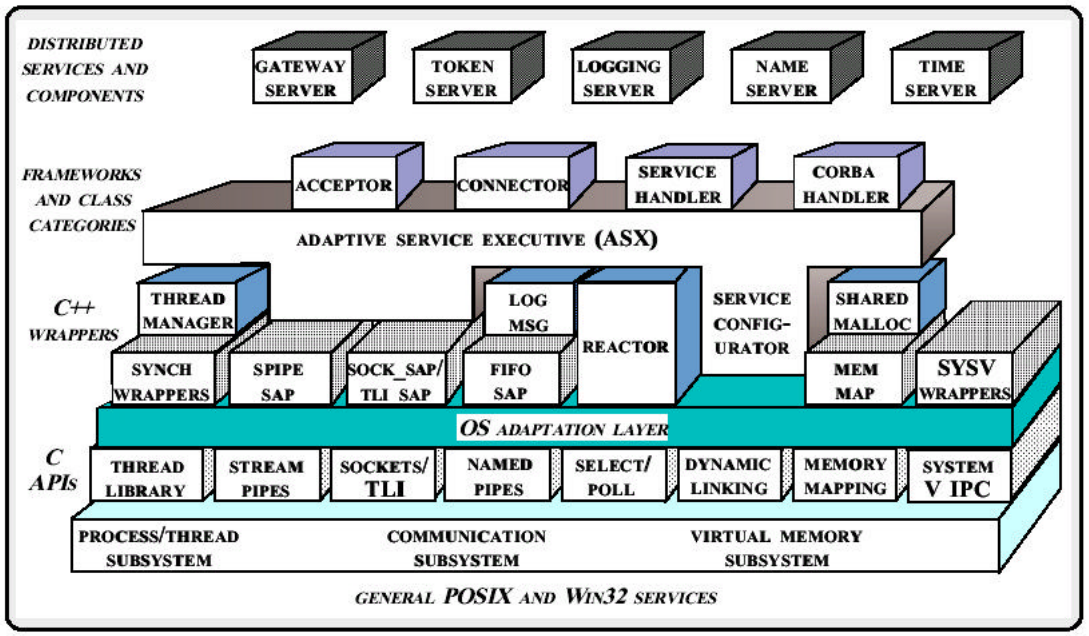

Figure 1: Components in the ADAPTIVE Communication Environment

ACE工具包是用层次体系结构设计的。图1说明了ACE组件的垂直和水平关系。ACE的底层是封装了现存操作系统网络编程机制的的面向对象的wrappers。高层扩展了底层的wrappers，以提供面向对象的框架和组件，这些框架和组件覆盖了面向应用程序的网络任务和服务的较大范围。本部分的剩下内容给出了ACE 的类的结构和功能的大致介绍。第3部分是ACE网络编程特性和组件的进一步介绍。

本文中，ACE组件是通过 Booch 表示来解释的。实心矩形表示类别，它结合了若干相关的类为一个通用的名字空间；实心云朵表示对象；嵌套表示对象间的组织关系；无向边表示两个对象间存在某种类型的连接。虚线云朵表示多个类；有向边指示类之间的继承关系；一个无向边以及在一端的一个小圆指

示两个类之间的组成或使用关系。“A”记在一个三角形里表示一个抽象类。一个抽象类不能被直接实例化，必须先继承到子类。一个抽象类的子类在任何类的对象实例化之前必须提供它的所有抽象方法的定义。

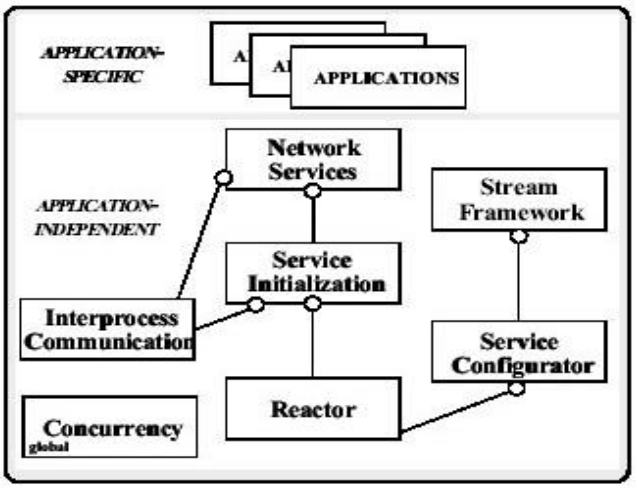

Figure 2: The Class Categories in ACE

2.1 ACE 操作系统适配层

ACE源码包含超过85,000行C++语句。大约9,000行代码属于操作系统适配层。该层将ACE的高层和与平台相关的非独立性屏蔽起来。它包括以下操作系统机制：

- 多线程和同步
- 进程间通信
- 事件分离
- 直接动态链接
- 内存映射文件和共享内存

2.2 ACE OO Wrappers

在操作系统适配层之上的是面向对象的wrappers，这些wrappers封装和加强了并发性，进程间通信（IPC），和现代操作系统的虚拟内存管理机制。应用程序可以通过有选择的继承、聚合和/或实例化下列ACE包的分类来结合以及组织这些组件：

- IPC SAP－封装了本地和/或远地的服务访问点（IPC SAP机制例如sockets，TLI，UNIX FIFOs和STREAM 管道，以及Win32命名管道）
- 访问初始化—ACE提供了CONNECTOR和ACCEPTOR组件的集合，在初始化完成后,这些组件从通信访问执行的任务中分别分离主动和被动初始化角色。
- 并发机制—ACE抽象了底层操作系统的多线程和多进程机制（例如互斥和信号量）以生成高级的面向对象并发抽象（例如主动对象）
- 内存管理机制—ACE内存管理组件为管理共享内存和本地内存的动态

定位和内存单元分配提供了一个灵活的可扩展的抽象。

- CORBA集成——ACE可以和CORBA产品集成起来（例如单线程和多线程 Orbix）。

OO wrappers 的使用通过类型安全的面向对象接口封装操作系统的通信、并发和虚拟内存机制提高了应用程序的健壮性。这样就避免了应用程序直接访问底层操作系统库，而底层操作系统库是用弱类型的C接口实现的。因此，OO语言的编译器，例如C++和Java可以在编译时检测到类型系统冲突，而不是在运行时。C++版的ACE使用了内联函数扩展以消除性能代价，否则wrapper层附加的类型安全和抽象会导致性能下降。

2.3 ACE 框架

ACE包含一个高级的网络编程框架，它集成并加强了低层OS wrappers。这个框架支持由应用程序服务组成的动态配置和并发网络后台程序。ACE的框架部分包含下列分类：

- Reactor—ACE Reactor提供了可扩展的、面向对象的多路输出，根据不同事件的类型进行处理器调度（如基于I/O的事件、基于定时器的事件、信号和同步的事件等）；
- Service Configurator—ACE Service Configurator 支持在安装或运行时配置服务的应用程序结构；
- Streams—ACE streams 组件简化了包含一个或多个层次相关的服务的并发通信软件的开发（例如协议栈）。

2.4 ACE 网络服务组件

除了wrappers和框架，ACE提供了一个标准的网络服务组件库。这些组件在ACE中有两个角色：

1. 它们解释了如何利用ACE IPC wrappers，Reactor，Service Configurator，service Initialization, Concurrency, Memory Management, and Streams components;
2. 它们提供了通用分布式系统任务的可重用组件，例如日志，命名，时间同步等。

当和OO语言特性以及设计模式相结合时，可重用的ACE组件方便了通信服务的开发，另外还方便了那些要在不修改，不重新编译，不重新链接的情况下可以升级的应用程序的开发。

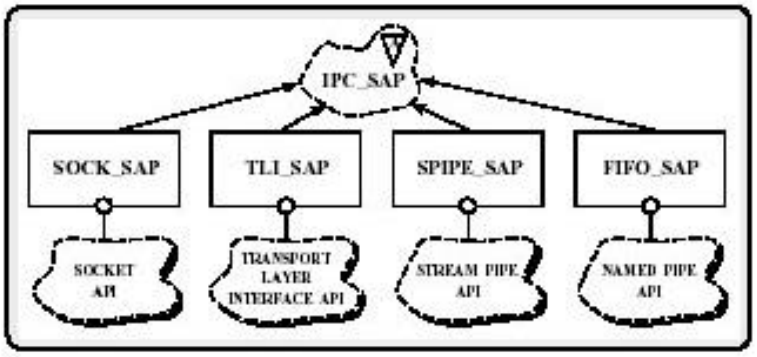

Figure 3: IPC_SAP Class Category Relationships

3 ACE 组件详细介绍

3.1 IPC_SAP: 本地和远程 IPC 机制

ACE提供了一些根部为IPC SAP基类的类别，IPC SAP封装了标准的基于I/O处理的操作系统本地和远程IPC机制，可以提供面向连接和无连接的服务。如图3所示，这些类包括SOCK SAP，TLI SAP，SPIPE SAP和FIFO SAP。

每个类别组织成一个继承层次。每个子类提供一个本地或远程通信机制的子集的定义好的接口。并且在一个层次的子类包含一个特定通信抽象的所有功能（例如Internet域或UNIX域协议族）。使用类而不是单独的函数可以从以下几个方面简化网络编程：

- 将应用程序与易导致错误的细节屏蔽起来——例如，图3中的ACE_Addr类层次通过一个类型安全的OO接口支持几个不同的网络地址格式，而不是直接使用基于C的易错的struct sockaddr 数据结构。
- 将几个操作结合起来组成一个单独的操作——例如，SOCK Acceptor结构执行不同的socket 系统调用（例如socket，bind和listen），这些调用都需要生成一个被动模式的服务器端点。
- 在应用程序中将IPC机制参数化——类通过它需要的IPC机制类型构成了应用程序参数化的基础。这帮助提高了可移植性。
- 增强代码共享——基于继承的层次分解增加了通用代码的数量，这些通用代码是在不同的IPC机制之间共享的（例如底层的面向OS设备控制系统调用的OO接口，如fcntl和ioctl）。

下面的部分讨论IPC SAP中的每个类别。

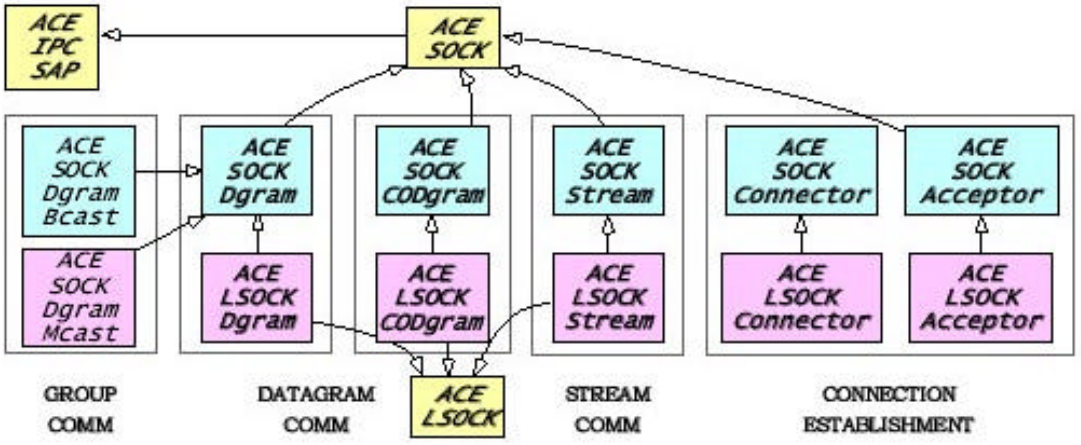

Figure 4: The SOCK SAP Class Categories

3.1.1 SOCK SAP

SOCK SAP 类别提供给应用程序一个· 面向Interner域和UNIX域协议族的接口。应用程序可以通过继承或实例化图4中适当的SOCK SAP子类来访问底层Internet 域或UNIX域socket类型的函数。ACE_SOCK*子类封装了Internet域的函数，而ACE_LSOCK*子类封装了UNIX域的函数。如图4所示，这些子类可以进一步分解到（1）*Dgram 组件中（该组件提供不可靠，无连接的，面向消息函数），或分解到*ACE_STREAM组件中（该组件提供了可靠的，面向连接和字节留的函数）（2）分解到ACE_*ACCEPTOR组件中（该组件提供了主要由服务器端使用的连接建立函数）或*STREAM组件中（该组件提供由客户和服务器端共同使用的双向字节流数据传输函数）。

使用OO wrappers封装socket接口能够在下列方面提供帮助（1）在编译时检测许多微小的应用程序类型系统冲突，（2）方便了独立于平台的传输层接口，提高应用程序可移植性，（3）大大减少了在底层网络编程细节上所花费的应用程序代码数量和精力。为了解释后点，下面给出一个简单的使用ACE_SOCK_Dgram_Bcast类的程序，它向所有的监听LAN中特定端口的服务器发送一个广播消息：

```
int    main (int argc, char *argv[])
{
ACE_SOCK_Dgram_Bcast b_sap (sap_any);
char *msg;
unsigned short b_port;
msg = argc > 1 ? argv[1] : "hello world\n";
b_port = argc > 2 ? atoi (argv[2]) : 12345;
if (b_sap.send (msg, strlen (msg),
b_port) == -1)
perror ("can't send broadcast"), exit (1);
exit (0);
}
```

建议将此简洁的例子和C源代码中直接使用socket接口实现广播的几十行程序进行比较。

3.1.2 TLI SAP

TLI SAP 类别 提供了面向System V TLI的OO接口。TLI SAP继承层次和SOCK SAP几乎是一样的。主要的不同点在于TLI和TLI SAP没有定义面向UNIX域协议族的接口。此外，TLI目前不适应于win32系统。

通过将C++的特性与tirdwr（STREAMS模块读/写兼容性）结合，开发需要在编译时参数化的应用程序变得相对直接。例如，下列代码解释了C++模板如何被应用到IPC机制的参数化中的。这段代码是从4.1部分的分布式日志工具中提取出来的。下面的代码中，从ACE_Event_Handler继承的一个子类由特定类型的传输接口和相应协议地址类进行参数化。

```
template <class PEER_STREAM, class ADDR>
class Logging_Handler : public ACE_Event_Handler
{
public:
Logging_Handler (void);
virtual ~Logging_Handler (void);
virtual int handle_input (ACE_HANDLE);
virtual ACE_HANDLE get_handle (void) const
{
   return this->peer_stream_.get_handle ();
}
protected:
PEER_STREAM peer_stream_;
};
```

根据特定的底层操作系统平台，日志应用程序可以实例化 Client Handler**类以使用**SOCK SAP**或**TLI SAP**，如下所示：**

```
/* Logging application */
class Logging_Handler
#if defined (MT_SAFE_SOCKETS)
: public Logging_Handler<ACE_SOCK_Stream, ACE_INET_Addr>
#else
: public Logging_Handler<ACE_TLI_Stream, ACE_INET_Addr>
#endif /* MT_SAFE_SOCKETS */
{
/* ... */
};
```

这个基于模板的方法提供了更多的灵活性，此方法在开发需要跨操作系统平台的应用程序时很有用。尤其是，根据传输接口对应用程序参数化在跨越不同的 SunOS**平台时是必需的，因为**SunOS5.2**不是**thread-safe**而且**TLI**在**SunOS4.x**中实现时有一些缺陷。**

TLI SAP**也将应用程序与许多特殊的**TLI**接口屏蔽起来。例如，通过**C++**缺省参数的使用，调用**accept**的标准方法对基于**TLI SAP**和基于**SOCK SAP**的应用程序在语法上是相同的。**

3.1.3 SPIPE SAP

SPIPE SAP**类别为高性能本地**IPC**提供了一个**OO wrapper**接口。在**Win32**平台，**SPIPE SAP **类别是在** Named Pipes**的上层实现的。**Win32**的**Named Pipes**机制主要用于在同一台机器的进程之间高效传输数据。它比本地**IPC**的**socket **效率高的多。**

在UNIX**平台，**SPIPE SAP **类别是用包装好的**STREAM **管道和**connld**实现的。**SunOS 5.x**提供了**fattach**系统调用，该调用在**UNIX**文件系统的一个指定位置包装了一个管道句柄。服务器应用程序的生成可以通过将**connld STREAM**模块**

推到管道包装好的一端来完成。当客户程序运行，随后在同一台主机上服务器打开与mounted管道相关的文件名，客户和服务器各接收到一个I/O句柄，此句柄指定了一个唯一的，非多元的，双向的通信信道。

SPIPE SAP继承层次和SOCK SAP、TLI SAP类似。它提供了类似SOCK SAP ACE_LSOCK*类相似的功能。然而，在SunOS 5.x平台上SPIPE SAP比ACE_LSOCK* 接口复杂的多，因为它允许STREAM模块被“推”到SPIPE SAP端点或从端点“弹”出。SPIPE SAP也支持双向字节流的分发和对运行在同一主机的进程或线程之间面向消息的数据分配优先权。

3.1.4 FIFO SAP

FIFO SAP 类别封装了UNIX命名管道机制。与STREAM管道不同的是，命名管道仅提供一个从一个或多个发送者到一个接收者的单向的数据通道。并且，不同发送者都置于同一通信信道。因此，在每个消息中必需清楚的包含标识，使接收端能够确定是哪个发送端发出了此消息。

命名管道在SunOS 5.x中基于STREAMS的实现同时提供了面向对象和面向字节流的数据分发语义。而有些平台（如SunOS 4.x）只提供面向字节流的命名管道。因此，除非发送的总是固定长度的消息，在SunOS4.x每个通过命名管道发送的消息必需通过某种字节计数或特殊的结束符号来区分，这样接收端才能从通信信道字节流中提取消息。为了避免这个限制，ACE FIFO SAP实现中包含logic，能够仿效SunOS5.x中的面向消息的语义。

3.1.5 其它通信机制

除了封装基于句柄的I/O通信机制外，ACE也为内存映射文件和SystemV UNIX IPC机制提供了OO wrappers.

● 内存映射文件：ACE_Mem_Map类为Win32和UNIX中的其它内存映射文件机制（例如系统调用的mmap族）提供了一个OO接口。这些系统调用使用底层的OS虚拟内存工具以将文件映射到进程的地址空间。映射文件的内容可以通过指针直接访问。一个指针接口通常比通过标准的I/O系统读/写调用访问数据块要方便和高效的多。此外，内存映射文件的内容也可以在两个或更多进程之间共享。

现存的Win32和UNIX内存映射文件接口结构有些复杂。例如，开发者必须手工执行许多bookkeeping细节（打开一个文件，确定文件长度，执行多重映射等）。相反，ACE_Mem_Map OO wrapper 提供了一个接口，该接口使用缺省值和多个含有几个类型指定变量的构造器以简化典型的内存映射文件使用模式。

下列程序给出了一个例子，该例使用ACE_Mem_Map OO wrappers映射通过指令行指定的文件并反向打印文件行。

```
static void
putline (const char *s)
{
while (putchar (*s++) != ’\n’)
continue;
```

```
}
int
main (int argc, char *argv[])
{
char *filename = argv[1];
char *file_p;
Mem_Map mmap (filename);
if (mmap (file_p) != -1)
{
size_t size = mmap.size () - 1;
if (file_p[size] == '\0')
file_p[size] = '\n';
while (--size >= 0)
if (file_p[size] == '\n')
putline (file_p + size + 1);
putline (file_p);
return 0;
}
else
return 1;
}
```

● System V IPC **机制：**SunOS UNIX**提供了一套共享内存，同步和消息传递机制，即通常所说的“**System V IPC**”。这些机制提供的大部分功能已被更新版的**SunOS UNIX**工具包含（例如**mmap**，线程同步和**STREAM**管道原语）。不过，某些应用程序（如数据库引擎）仍可以从**System V IPC**机制中获得好处（例如消息队列的端到端特性，信号量的操作原子性，**UNIX**系统平台广泛应用的**System V IPC**机制）。但正确使用**System V IPC**机制有些困难，因为它们的接口很通用而文档却直到最近才多起来。**

ACE System V IPC wrapper **接口将许多不必要的细节屏蔽起来。例如，**ACE OO wrapper**的**System V IPC **信号量比标准的**wait **和** signal **更简单好用，如下列生产者/消费者的典型代码段中一段所示：**

```
typedef ACE_SV_Semaphore_Simple SEMA;
SEMA prod (1, SEMA::CREATE, 1);
SEMA cons (2, SEMA::CREATE, 0);
void producer (void)
{
for (;;) {
prod.wait ();
// produce resource...
cons.signal ();
}
}
void consumer (void)
```

```
{
for (;;) {
cons.wait ();
/ / consume resource...
prod.signal ();
}
}
```

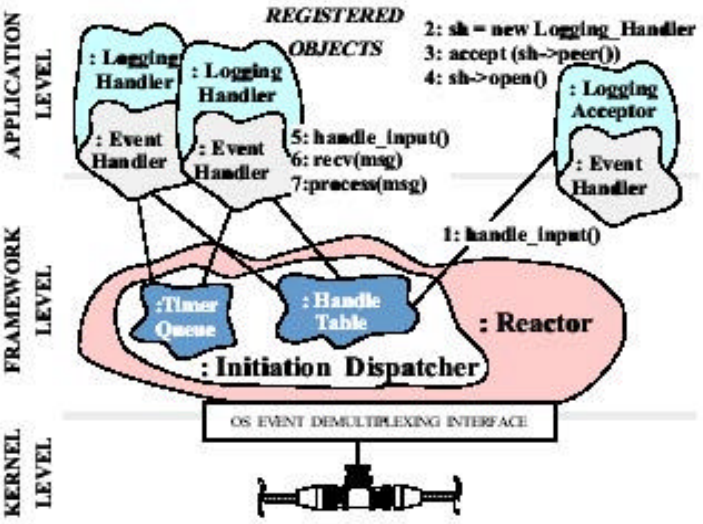

Figure 5: Software Architecture of the WWW Server

3.2 Reactor：事件分离和事件处理器的调度

通信软件分离和处理许多不同类型的事件（例如基于定时器的，基于I/O的，基于信号的以及同步的事件）。例如，一个WWW服务器内部通常使用事件循环来监测一个著名的Internet端口（80）。这个端口与一个应用程序处理器相关联，这个处理器监听连接到80端口的客户。当有客户连接时，WWW服务器接收这个连接并生成一个事件处理器为HTTP请求服务。例如，如果Netscape浏览器发送一个GET请求，WWW服务器把请求内容返回给浏览器。

为了使事件驱动处理变成自动的，ACE提供了事件分离和事件处理调度框架——ACE_Reactor。Reactor在一个可移植和可扩展的OO wrapper内封装了UNIX和Windows NT的事件分离机制（例如选择和轮询）。这些操作系统事件分离的系统调用同时在一个或多个I/O处理中检测不同类型的输入输出事件。

为了方便应用程序移植性，ACE_Reactor为不同的事件分离机制提供了相同的接口。此外ACE_Reactor封装了互斥机制，在多线程事件处理环境中，该机制在正确和高效执行回调类型的事件调度时是必需的。

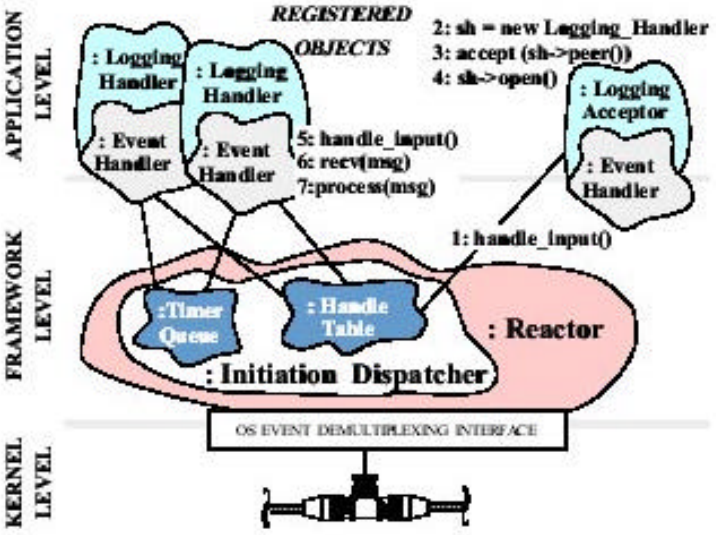

Figure 6: The ACE_Reactor Class Category Components

ACE_Reactor**中的对象结构如图**6**所示。其中的对象负责（**1**）事件分离（**2**）调度预先注册的事件处理器的适当方法给这些事件的进程。如图**6**所示，所有的事件句柄对象都是从**ACE_Event_Handler**抽象基类派生来的。该类说明了一个接口，此接口被**ACE_Reactor**用来根据到达的事件调度特定方法。**

ACE_Reactor**使用在**Event Handler**接口中声明的虚拟方法来集成基于**I/O**处理的事件分离、基于定时器的事件分离和基于信号量的事件分离。基于**I/O**处理的事件通过**handle_input, handle_output**和**handle_exceptions**方法进行调度；基于定时的事件通过**handle_timeout **方法进行调度；基于信号量的事件则通过**handle_signal**方法进行调度。**

ACE_Event_Handler**的子类（如**4.1**描述的**Logging_Handler**）可以通过定义新的方法和数据成员增加基类的接口。此外，**ACE_Event_Handler**接口中的虚拟方法可以由子类有选择的重载以实现特定的应用程序功能。例如，**4.2**中的**PBX**监视服务器程序子类通过继承和实例化**SOCK SAP**或**TLI SAP **传输接口类与客户端进行通信。子类定义了**ACE_Event_Handler**基类中的虚拟方法后，应用程序可以实例化结果处理器对象。**

下例实现了一个简单的程序，该程序在两个使用双向通信信道的进程之间来回交换消息。该例子解释了服务是如何从ACE_Event_Handler**继承的。它也描述了**ACE_Reactor**是如何被用于分离和调度基于**I/O**，基于信号量和基于定时的事件。其中的**Ping-Pong **类继承了**ACE_Event_Handler**的接口，并实现了它的函数。**

```
class Ping_Pong : public ACE_Event_Handler
{
public:
Ping_Pong (char *b)
: len (min (strlen (b) + 1, BUFSIZ)) {
strncpy (this->buf, b, BUFSIZ);
```

```
}
virtual int handle_input (ACE_HANDLE handle) {
return read (handle, this->buf, BUFSIZ);
}
virtual int handle_output (ACE_HANDLE handle) {
return write (handle, this->buf, this->len);
}
virtual int handle_signal (int signum) {
this->finished = 1;
}
virtual int handle_timeout (const Time_Value &,
const void *) {
this->finished = 1;
}
bool done (void) {
return this->finished == 1;
}
private:
sig_atomic_t finished;
char buf[BUFSIZ];
size_t len;
};
```

双向通信信道使用SVR4 UNIX STREAM**管道创建：**

```
static int
init_handles (ACE_HANDLE handles[])
{
if (pipe (handles) == -1)
LM_ERROR ((LOG_ERROR, "%p\n%a", "pipe", 1));
// Enable message-oriented mode instead of
// bytestream mode.
int arg = RMSGN;
if (ioctl (handles[0], I_SRDOPT, arg) == -1
|| ioctl (handles[1], I_SRDOPT, arg) == -1)
return -1;
}
```

主程序开始时先打开正确的通信信道，随后程序分出一个子进程，实例化一个 Ping_Pong**事件处理程序对象，叫做**callback**，接着用**ACE_Reactor**的一个**instance**为**I/O**事件、信号事件或定时事件注册**callback**对象，然后进入一个事件循环，如下：**

```
int main (int argc, char *argv[])
{
ACE_HANDLE handles[2];
```

```
ACE_Reactor reactor;
init_handles (handles);
pid_t pid = fork ();
Ping_Pong callback (argv[1]);
// Register I/O-based event handler
reactor.register_handler (handles[pid == 0],&callback,
                          ACE_Event_Handler::READ_MASK
                          | ACE_Event_Handler::WRITE_MASK);
// Register signal-based event handler
reactor.register_handler (SIGINT, &callback);
// Register timer-based event handler
reactor.schedule_timer (&callback, 0, 10);
/* Main event loop (run in each process) */
while (callback.done () == false)
reactor.handle_events ();
return 0;
}
```

基于定时器的和基于信号量的Callback**事件处理器和表存储在**ACE_Reactor**内部。同样**ACE_Reactor**也在一个内部表中存储着正确的处理方法，当**register_handler**方法被调用来登记基于**I/O**的事件处理器时。随后应用程序通过调用**ACE_Reactor::handle_events **方法执行主程序事件循环，这个处理被作为参数传递给底层**OS I/O **分离系统调用。（如选择或轮询）。**

由于输入，输出，信号量和定时事件与callback**对象的预登记事件处理器关联发生在运行时，**ACE_Reactor**自动检测这些事件和调度正确的事件处理对象。**Callback**对象的调度方法负责执行应用程序相关的函数（如写消息到通信信道，从通信信道读消息或设置信号触发程序的终止）。这些组件间的协作通过图**7**中的对象交互图描述。**

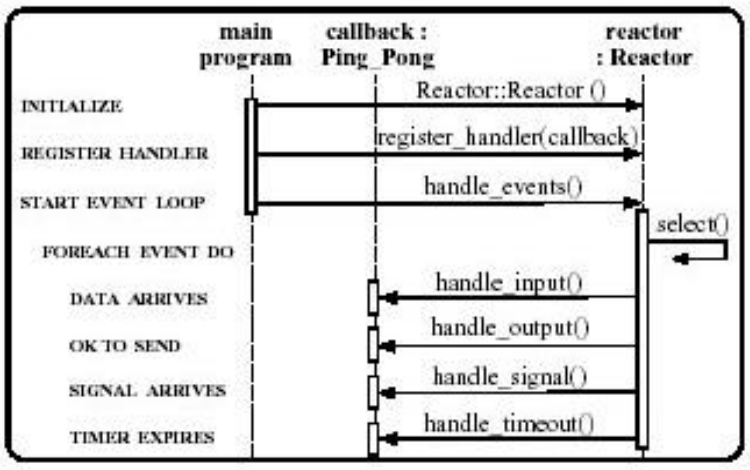

Figure 7: ACE Reactor Interaction Diagram

3.3 Concurrency：多线程和同步机制

ACE Concurrency 类别 包含OO wrappers（如ACE_Mutex,ACE_Condition，ACE_Semaphore和ACE_RW_Mutex）封装了相应的Solaris和Posix Pthreads 多线程和同步机制。这些wrappers自动完成同步对象的初始化，这些同步对象作为类中的fields定义，也简化了典型的线程和同步机制的使用模式。例如，下列代码解释了在如何在共享资源管理类中使用ACE 面向 SunOS mutex_t和cond_t 同步机制的wrappers：

```
class Resource_Manager
{
public:
Resource_Manager (u_int initial_resources)
: resource_add_ (this->lock),
resources_ (initial_resources) {}
int acquire_resources (u_int amount_wanted)
{
this->lock_.acquire ();
while (this->resources_ < amount_wanted) {
this->waiting_++;
// Block until resources are released.
this->resource_add_.wait ();
}
this->resources_ -= amount_wanted;
this->lock_.release ();
}
int release_resources (u_int amount_released)
{
this->lock_.acquire ();
this->resources_ += amount_released;
if (this->waiting_ == 1) {
this->waiting_ = 0;
this->resource_add_.signal ()
}
else if (this->waiting_ > 1) {
this->waiting_ = 0;
this->resource_add_.broadcast ();
}
this->lock_.release ();
}
// ...
private:
ACE_Mutex lock_;
ACE_Condition<ACE_Mutex> resource_add_;
```

```
u_int resources_;
u_int waiting_;
// ...
};
```

注意ACE_Condition**对象** resource_add**的构造器是如何将**ACE_Mutex**对象和**Condition**对象绑定到一起的。这样简化了**ACE_Condition::wait **调用接口，与底层**SunOS cond_t cond_wait**接口相比。**

尽管ACE_Mutex wrappers **为同步多线程控制提供了一个相对优雅的方法，它们潜在的仍是易错的，因为可能会忘记调用**release **方法。为了提高应用程序健壮性，**ACE**同步机制方便了**C++**类构造和析构的语义。为确保**ACE_Mutex**锁定可以自动获得和释放，**ACE**提供了一个辅助类，叫做**ACE_Guard**，定义如下：**

```
template <class MUTEX>
class ACE_Guard
{
public:
ACE_Guard (MUTEX &m): lock (m) {
this->lock_.acquire ();
}
~ACE_Guard (void) {
this->lock_.release ();
}
private:
MUTEX &lock_;
}
```

ACE_Guard**类的对象定义了一块需要**ACE_Mutex**的代码段，然后在退出代码段时自动释放**ACE_Mutex**。**

注意ACE_Guard**类定义成一个模板，通过互斥机制参数化。共有几种类型的互斥语义，每种类型的互斥共享一个通用接口（即**acquire/release**），但拥有不同的串行化和执行属性。**ACE**支持的两种类型的互斥是非递归和递归锁定。**

- **非递归锁：非递归锁提供了高效的互斥**form**，该**form**定义了一个临界区（**critical section**），某一时刻只能有一个线程执行。这就是非递归调用，线程拥有一个锁，如果锁没有释放就不能重新获得这个锁。否则就会立即发生死锁。**SunOS5.x**支持了非递归锁，利用它的**mutex_t, rwlock_t**和**sema_t**类型。**ASX**框架提供了**Mutex，RW_Mutex**和**Semaphore**包，分别封装了这些语义。**
- **递归锁： 递归锁允许**acquire**方法嵌套调用，只要拥有锁的的线程重新请求该锁。递归锁在回调驱动事件调度框架中很有用（例如**Reactor**），其中**event_loop**框架执行回调以预先登记用户定义对象。由于用户定义对象可能会随后通过方法入口点重新进入调度框架，递归锁必需用来防止在回调时发生死锁。**

下列C++**模板类实现了**Solaris**线程和**POSIX Pthread**中的同步递归锁语义，这两者原本不支持递归锁语义：**

```
template <class MUTEX>
```

```
class ACE_Recursive_Thread_Mutex
{
public:
// Initialize a recursive mutex.
ACE_Recursive_Thread_Mutex (void);
// Implicitly release a recursive mutex.
~ACE_Recursive_Thread_Mutex (void);
// Acquire a recursive mutex.
int acquire (void) const;
// Conditionally acquire a recursive mutex.
int tryacquire (void) const;
// Releases a recursive mutex.
int release (void) const;
private:
ACE_Mutex nesting_mutex_;
ACE_Condition<ACE_Mutex> mutex_available_;
thread_t owner_id_;
int nesting_level_;
};
```

注意，该类的接口与ACE**中其它的锁定机制是一致的。**

下面的代码解释了ACE_Guard**和**ACE_Recursive_Thread_Mutex**如何被用于**callback**：**

```
int
Callback::dispatch (const Event_Handler *eh,
Event *event)
{
// Constructor acquires the lock on entry.
ACE_Guard<ACE_Recursive_Thread_Mutex<ACE_Mutex> >
m (this->lock_);
eh->handle_event (event);
// Destructor of Guard releases the lock on exit.
}
```

这些代码保证了注册一个Event_Handler对象的执行程序是一个临界区。本例也解释了C++ idiom的使用。同样，类析构自动解锁对象，当mon对象超出范围时。并且，对mon的析构将自动调用以释放Mutex锁，而不考虑是从if/else语句的哪个分支返回的。除此之外，如果在register_handler函数体的处理过程中出现C++异常，锁会自动释放。ACE_Recursive_Thread_Mutex用于确保由dispatch方法调度的特定的应用程序的handle_event回调函数重新进入Callback对象时不会导致死锁。

ACE也提供了一个ACE_Thread_Manager类，此类包含一个用于管理一组线程的机制集合，这组线程合作完成一组动作。例如，ACE_Thread_Manager**类提供了允许任意数量的参与线程自动挂起和重新启动的机制（如 suspend_all和**

resume_all）。

3.4 Service Configurator：直接动态链接机制

静态链接技术通过在编译和/或静态链接时将所有对象文件绑定到一起，组成一个复杂的可执行程序。动态链接正相反，它允许在运行最初的程序调用时刻或在后来的任何时刻增加和/或减少对象文件到进程的地址空间。SunOS4.x和5.x同时支持间接和直接动态调用：

？间接动态链接(Implicit dynamic linking) 用于实现共享对象文件，也叫做共享库。共享对象文件减少了主存储器和附属存储器的使用，因为在内存或磁盘上只有共享对象代码的一个拷贝存在，不管正在执行这段代码的进程数目是多少。此外，地址解析和重定位操作会推迟到一个动态链接函数第一次涉及到时。这种“lazy evaluation”策略减少了在进程开始时的链接编辑开销。

？直接动态链接(Explicit dynamic linking) 提供了一个接口，此接口允许应用程序获得、使用和/或移动在共享对象文件中定义的符号在运行时的地址绑定。直接动态链接机制在增强通信软件的功能性和灵活性方面非常有意义，因为服务可以在运行时插入和/或删除而不需要终止和/或重新启动整个应用程序。SunOS5.x通过 dlopen/dlsym/dlclose 例程支持直接动态链接，win32 则通过 LoadLibrary/GetProcAddress 例程来支持。

ACE提供了Service Configurator 类来将SunOS的直接动态链接机制封装在一个类集和继承层次中。Service Configurator 支撑其它的ACE组件以扩充传统的后台程序配置和控制框架（例如listen，inetd和Windows NT Service Control Manager），提供了对下列功能的自动支持（1）多个服务的通信软件的静态和动态配置，（2）监测I/O动作的一组通信端口（3）调度由监测端口到达的消息给正确的应用程序服务程序。本部分剩余的内容讨论Service Configurator类别的基本组件。

3.4.1 ACE_Service_Object 继承层次

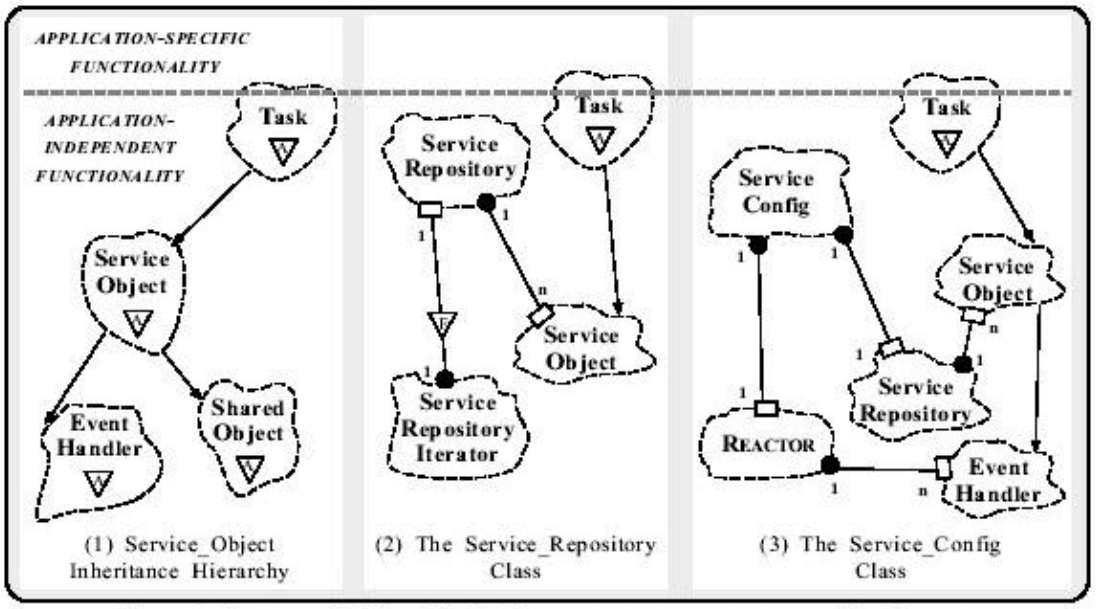

Figure 8: Component Relationships for the `Service Configurator` Class Category

在Service Configurator 中的配置基本单元是service。Service 可以是简单的（例如返回当前时间）或极其复杂的（例如分布式实时路由）。为了提供一个定义、配置和使用通信软件的一致环境，所有的应用程序service都是从ACE_Service_Object继承层次得到的。（图8(1)）

ACE_Service_Object 类是一个类型相关的多级层次的焦点。这个类型层次的抽象类提供的标准接口可以有选择的实现，以便访问独立于应用程序的Service Configurator机制。这些机制提供了透明动态链接，事件处理器的注册，事件分离和服务调度。通过将处理器对象中应用程序相关的部分从底层独立于应用程序的Service Configurator机制中分离，明显减少了从运行时的应用程序中插入和移动services所要做的工作。

ACE_Service_Object 继承层次由ACE_Event_Handler 和ACE_Shared_Object 抽象基类和ACE_Service_Objec t派生类组成。ACE_Event_Handler类在3.2中已经说明。其它类的动作下面列出：

● ACE_Shared_Object抽象基类：该基类将一个动态链接服务处理对象的接口指定到应用程序的地址空间。ACE_Shared_Object 抽象基类输出三个抽象方法：init，fini,和info。这些方法在Service Configurator提供的独立于应用程序的组件和应用程序使用这些组件的相关函数之间强加了一个contract。通过使用抽象方法，Service Configurator 确保服务处理程序提供配置相关信息。这信息随后由Service Configurator 在运行时用于自动链接，初始化，确认和断开服务程序。

ACE_Shared_Object 基类的定义独立于ACE_Event_Handler 类以清楚地区分两者涉及的正交集。例如，一些应用程序（如编译器或文本编辑器）可以从动态链接方法中得到益处，尽管它们可能不需要通信端口事件分离。相反，其它的一些应用程序（如ftp服务器）可能需要事件分离而不需要动态链接。通过将这些接口分成两个基类，应用程序可以选择Service Configurator 机制的子集就不会导致不必要的存储开销。

● ACE_Service_Object 抽象派生类：通常，安装和运行复杂的分布式系统需要动态链接，事件分离和服务调度以自动进行应用程序服务的动态配置和重新配置。因此，Service Configurator 定义了ACE_Service_Object 类，该类将ACE_Event_Handler和ACE_Shared_Object抽象基类的接口联结起来。产生的派生类提供了一个接口供程序员作为实现和配置服务到Service Configurator的基础使用。

在开发期间，应用程序指定的子类ACE_Service_Object 必需实现由ACE_Service_Object指定的suspend 和resume 抽象方法。这些方法作为外部事件的响应由Service Configurator 自动调用。一个应用程序开发人员可以控制个对象的动作以在不完全移动和断开的情况下挂起一个服务，重新开始一个先前挂起的服务也一样。

此外，应用程序指定的子类也必需实现下列四个抽象方法：init，fini，info，和get_handle，这些方法从ACE_Service_Object 继承来的。Init 方法作为在运行初始化时服务程序的入口点，负责在动态链接到一个ACE_Service_Object 实例时执行应用程序指定的初始化。同样，fini方法在ACE_Service_Object 断开链接并从应用程序移走时由Service Configurator 自动调用，主要执行释放动态定位资源的结束操作（如内存，I/O句柄或同步锁的释放）。Info方法将服务地址分配信息用可读字符串产生一个简明报告并将服务功能文档化。客户可以查询应用程序检索这些信息并用它和在此应用程序运行的特定服务联系。最后，get_handle方法由Reactor 用于从一个服务处理器对象中提取底层的I/O处理器，该I/O处理器指示了一个可以用来接受客户连接或客户数据的传输点。

● 应用程序相关的具体派生类

Service Object是一个抽象类，因为它的接口包含从Event Handler 和Shared Object 抽象基类继承来的方法。因此，程序开发人员必需给出具体的子类 （1）定义以上6个抽象方法（2）实现必需的应用程序指定功能。要完成后一个任务，子类通常定义由Service Object接口导出的虚方法。例如，handle_input 方法经常用于接收来自客户的连接请求和数据。

图8(1)中描述的ACE_Acceptor 类是一个独立于应用程序的子类，它接收连接请求作为一个分布式日志工具的一部分。该类在4.1的例子里进一步描述。

3.4.2 ACE_Service_Repository 类

Service Configurator类同时支持单个服务和多个服务通信软件。因此，为了简化运行时的管理，经常需要单独和/或集体控制和协调包含应用程序当前的活动服务ACE_Service_Objects对象。ACE_Service_Repository 是一个对象管理器，负责协调本地和远程的服务请求，这些请求使用到基于Service Configurator 的应用程序提供的服务。该对象管理器内的一个查询结构绑定了服务名称（用ASCII字符表示）和ACE_Service_Object 实例（用对象代码表示）。一个服务名称指定唯一一个存储在库中的ACE_Service_Object 对象实例。

ACE_Service_Repository 的每个入口都包含一个指针指向应用程序派生类（图8(2)）的ACE_Service_Object 部分。这使得Service Configurator 能够静态或

动态的从应用程序装载、挂起、重启或卸载ACE_Service_Object。对于动态链接ACE_Service_Object，此库也维护着一个指向底层共享对象文件的句柄。当服务不再需要时，该句柄从一个运行时的应用程序断开链接并且卸载ACE_Service_Object对象。

一个iterator 类也被用于和ACE_Service_Repository 关联。该类用于访问库中的每一个ACE_Service_Object 对象而数据封装不需要不适当的折中改变。

3.4.3 ACE_Service_Config 类

ACE_Service_Config 类是Service Configurator 框架内的统一组件。如图8(3)所示，此类集成了其它的Service Configurator 框架组件（如ACE_Service_Repository 和ACE_Reactor）以自动完成并发，多服务通信软件的静态和/或动态配置。

```
<svc-config-entries> ::=
        svc-config-entries svc-config-entry
        | NULL
<svc-config-entry> ::= <dynamic> | <static>
         | <suspend> | <resume> | <remove>
         | <stream> | <remote>
<dynamic> ::= DYNAMIC <svc-location>
                    [ <parameters-opt> ]
<static> ::= STATIC <svc-name>
                  [ <parameters-opt> ]
<suspend> ::= SUSPEND <svc-name>
<resume> ::= RESUME <svc-name>
<remove> ::= REMOVE <svc-name>
<stream> ::= STREAM <stream_ops>
             '{' <module-list> '}'
<stream_ops> ::= <dynamic> | <static>
<remote> ::= STRING '{' <svc-config-entry> '}'
<module-list> ::= <module-list> <module>
                | NULL
<module> ::= <dynamic> | <static>
          | <suspend> | <resume> | <remove>
<svc-location> ::= <svc-name> <type>
                   <svc-initializer> <status>
<type> ::= SERVICE_OBJECT '*' | MODULE '*'
           STREAM '*' | NULL
<svc-initializer> ::= <object-name>
                    | <function-name>
<object-name> ::= PATHNAME ':' IDENT
<function-name> ::= PATHNAME ':' IDENT '(' ')'
<status> ::= ACTIVE | INACTIVE | NULL
<parameters-opt> ::= STRING | NULL
```

Figure 9: EBNF Format for a Service Config Entry

Symbol	Description
dynamic	Dynamically link and enable a service
static	Enable a statically linked service
remove	Completely remove a service
suspend	Suspend service without removing it
resume	Resume a previously suspended service
stream	Configure a Stream into a daemon

Table 1: Service Config Directives

ACE_Service_Config 类使用一个配置文件（svc.conf）引导配置和重新配置动作。每个应用程序必需关联到各自不同的svc.conf配置文件。同样，一些应用程序集可以用一个单独的svc.conf 配置文件说明。图9说明了svc.conf 文件中的基本语法元素，该图使用的是EBNF）。文件中的每个服务配置入口用service config directive 开始，service config directive指定了要执行的配置动作。表1总结了有效的服务配置directive。

每个服务配置入口包含有为每个动态链接服务指示共享对象文件所在地址的属性，和运行时初始化服务所需的参数。通过将服务属性和初始化参数统一到一个单独的配置文件，应用程序服务的安装和运行明显简化。svc.conf 文件协助的从应用程序的服务动作解偶应用程序结构。这个解偶合过程也允许框架提供的“ lazy” 配置和重新配置机制，该机制在svc.conf文件中基于应用程序相关属性和参数。

图10描述了一个状态透明图解释了Service Configurator类中在服务配置，执行和重新配置阶段为响应事件的发生而被调用的方法。例如，当CONFIGURE和RECONFIGURE 事件发生时，ACE_Service_Config类的process_directives 方法被调用来访问svc.conf 文件。当一个新的应用程序实例第一次初始化时，这个文件被首先访问。当应用程序在接收到一个触发重新配置的外部事件时（如UNIX SIGHUP 信号），再次访问该文件。

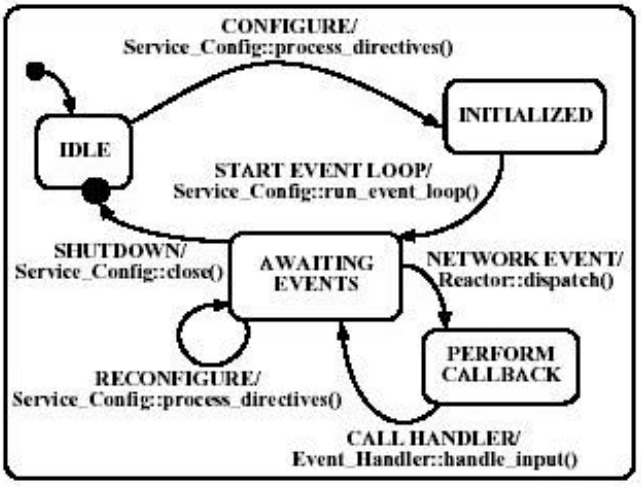

Figure 10: State Transition Diagram for Service Configuration, Execution, and Reconfiguration

3.5 Stream：各层次服务集成

Stream 类是ACE的一个主要焦点。该类包含ASX（ADAPTIVE Service eXecutive）框架，此框架集成了底层OO wrapper 组件（如IPC SAP）和高层的类（如Reactor 和Service Configurator）。ASX框架可以使集成各层次的通信软件的开发简化，尤其是用户级通信协议栈和网络服务。ASX框架的设计用于提高应用程序相关的服务的模块化，可扩展性，可重用性和可移植性，同时这些应用程序构建的基础——底层OS的并发、IPC、直接动态链接和分离机制的模块化，可扩展性，可移植性，可重用性也可提高。

ASX框架为通信软件的开发人员提供了以下两点便利：

1. ASX具体化，封装并实现了开发通信软件时通用的设计模式。设计模式可以通过针对在大规模系统开发中的困难来增强软件的质量。这些困难包括开发人员知识体系的交流；安置新的设计范例或体系风格；解决非函数功能如可重用性，可移植性和可扩展性；以及避免只有通过经验才能获得的开发陷阱和缺陷。

2. ASX严格区分关键的开发核心——ACE将通信软件开发分成两个不同的类：（1）独立于应用程序的核心，这些对大多数或全部通信软件是通用的（如端口的监测；消息缓冲，排队和分离；服务调度；本地/远程交互通信；并发控制；和应用程序配置，安装和运行管理（2）应用程序相关核心，这些是依赖于一个单独的应用程序的。通过重用ACE提供的OO wrappers和框架，程序开发人员可以不需要花时间重新构造通用的任务。而是关注关键的高层的功能需求和设计。

3.5.1 基本的 ASX 特性

ASX框架提高了通信软件的灵活性，这是通过将应用程序相关的处理策略和下列配置相关开发活动和机制进行分离获得的：

- 与每个应用程序处理器关联的服务类型和数目：ASX框架允许应用程序统一一个或多个服务到一个单独的管理单元。这种多服务配置通信软件方法有助于以下两点（1）通过自动执行通用的服务初始化简化开发和重用代码，（2）通过按需求产生服务处理程序减少OS资源的消费（例如进程表slots），（3）允许应用程序服务在不需要修改现存源代码或终止一个正在执行的调度进程的情况下更新应用程序（例如inetd superserver）（4）通过一个统一的配置管理操作集合来加强对网络服务的管理。
- 服务配置到应用程序的时间点：ASX框架支撑service Configurator（3.4.1 中描述）以提供一个扩展的面向对象的接口使对OS 直接 动态调用机制的使用自动进行。动态调用通过允许内部服务在应用程序第一次执行或运行时配置增强了通信软件的extensibility。这个特性使应用程序服务可以在不需要修改、重新编译、重新链接或重新启动活动服务的情况下动态重新配置服务。在ASX框架中，动态或静态配置可以基于单个服务进行选择。并且，这个选择可以推迟到应用程序开始执行时。
- 执行代理的类型：在ASX框架中，运行时，服务的执行可以通过几个不

同类型的进程和线程代理来完成。通过从调用服务的执行代理中分离服务功能，ASX框架为程序开发人员增加了应用程序并发配置可选范围。

有效的应用程序并发配置通常依赖于特定的服务需求和平台特性。例如，基于进程的配置适于实现时间较长的以进程所有者为安全机制的服务（例如ftp和telnet）。在这种情况下，每个服务（或每个服务的活动实例）可以被映射到一个单独的进程并在多进程处理器平台下并行执行。然而，其它的配置在别的环境中可能更合适。例如，在单线程中实现协同操作服务通常更简单有效，因为它们经常涉及通用数据结构，每个服务必须在同一个进程的一个单独线程执行以减少调度、内容转换和同步开销。

● 层次相关的服务联结到应用程序的顺序：复杂的服务可能由用消息传递进行通信连接的一系列独立服务对象组成。这些对象可以随意组合到一起满足应用程序需求和增强组件重用性。

● 基于I/O处理和基于定时器的时间分离机制：这些机制用于调度到达的连接请求和数据给一个预先注册的应用程序指定的处理程序。ASX框架使用Reactor类别通过一个扩展的类型安全的面向对象接口集成了基于I/O的，基于定时的，和基于信号的事件。

● 底层IPC机制：应用程序服务可以使用3.1节描述的IPC SAP机制与本地或远程系统加入的通信实体交换数据。与弱类型“基于句柄”的socket和TLI接口不同的是，IPC SAP包允许应用程序通过类型安全，可移植的接口访问底层OS IPC机制。

ASX框架综合了几个模块化通信框架的概念，包括System V的STREAMS，x-kernel 和Conduit framework [41] from the Choices object-oriented operating system（关于这些通信框架和其它框架的研究见参考文献〔42〕）。这些框架都包含通过互连的“building－clock”协议和访问组件来支持通信子系统的灵活配置。通常，这些框架都鼓励通过从周围的框架基础中分离出处理函数来开发标准的通信相关组件（例如消息管理，定时事件调度，分离和assorted 协议功能）。如下所述，ASX框架包含有从底层进程体系中进一步分离处理函数的特性。

与STREAMS不同的是，配置到ASX框架应用程序服务在用户空间执行而不是在内核空间执行。在用户空间开发通用目的的通信程序而不是在OS内核空间开发有下列好处：

● 访问通用的OS特性：在客户空间运行的应用程序可以访问OS机制的全部范围（例如动态链接，内存映射文件，多线程，大的虚拟地址空间，交互通信机制，文件系统和数据库）。相反，驻留内核的组件通常只能限制在内核机制的一个有限集合。尽管有些idioms已经克服了某些限制（例如，维护一个用户级的执行文件I/O的后台程序的就是一个驻留内核的组件），但这些工作区是笨拙的，不可移植的。

● 增强了开发环境：高级的程序开发工具可以用于在用户空间开发应用程序。相反，在OS内核开发网络服务是一个很复杂的困难任务，因为在内核编程环境中调试工具是低级的和subtle timing 交互。在这样受限制的情况下，程序员很难有效工作。

● 提高了系统健壮性：在用户级进程或线程产生的异常情况（如撤销空指针，或除以0等）只会影响offending进程或线程。然而，在OS内核产生的异常可

能会导致整个操作系统出故障或崩溃。并且，在每次崩溃后重新启动操作系统是很烦琐的。

● 移植性：在不同的操作系统平台（甚至是同一平台的不同产品）移植内核级的驱动程序都会比在平台之间移植用户级应用程序复杂的多。SunOS5.x DDI/DKI API 是为了避免在UNIX平台上的移植问题而使用的，但它并没有解决在其它平台上（如OS/2，Windows NT，等）运行的应用程序的移植问题。

在操作系统内核实现分别式服务的基本原因是提高性能。例如，一个驻留在内核的通信协议栈可以减少调度，内容转换和保护。然而，对许多通信软件而言，由在用户空间开发程序提供的日益增长的灵活性，简单性，健壮性和可移植性是弥补潜在的性能不足的关键。

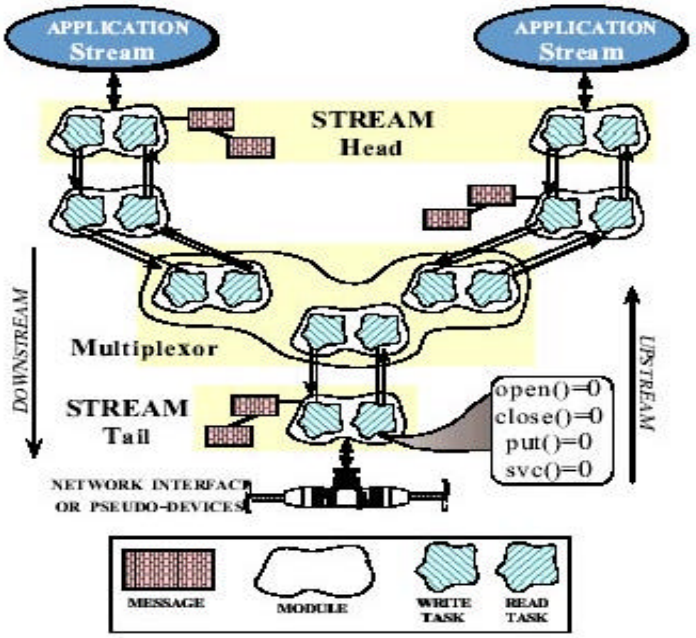

Figure 11: Components in the ASX Framework

3.5.2 Stream 类组件

在Stream类中的组件负责协调一个或多个ACE_Stream对象的配置和执行。ACE_Stream是和用户应用程序合作的一个对象，两者合作用于配置和执行ASX框架中的应用程序相关服务。如图3.5所示，ACE_Stream包含一些互连的ACE_Module对象，这些对象可以通过程序员安装时连接或应用程序运行时连接。ACE_Module 用来将应用程序体系分解成一些互连的，不同功能的层次。每个层实现一组相关的服务功能（如点到点的传输服务或表示层格式服务）。每个ACE_Module包含一对ACE_Task 对象，它们将一个层次分成读和写处理功能。

OO语言特性（如类，集成，动态绑定和参数化类型）使程序员可以合并应用程序功能到一个Stream中而不需要修改独立于应用程序的ASX框架组件。例如，增加一个服务层到Stream包括（1）从缺省的ACE_Task接口继承，有选择的

重载子类中的几个虚方法以提高应用程序需要的功能（2）分配一个新的ACE_Module包含应用程序指定的ACE_Task子类的两个实例（一个读，一个写）（3）将ACE_Modle 插入到Stream。相邻的互联的ACE_Task中的服务通过交换有类型的消息进行合作，这些消息是通过一个消息传递接口进行交互的。

为了避免重新开发类似的终端，许多在Stream 类中的类与System V STREAMS框架中的类似组件相对应。然而，在这两个框架中用来支持扩展性和并发性的技术有很大差别。例如，向ASX Stream类增加应用程序指定的功能是通过继承几个定义好的接口来执行的并由现存的框架组件实现的。使用继承性来增加新的函数比System V STREAMS中用指针指向函数提供了更好的类型安全。ASX Stream类也使用了一些不同的并发控制策略以减少死锁的可能性并且简化了Stream内的Tasks之间的流量控制。ASX Stream类完全重新涉及和实现了在System V STREAMS中使用的co-routine-based，“weightless”服务处理机制。这些ASX的变化是为了更有效的利用共享内存的多处理器平台下的多个PE。

本部分的剩下内容详细讨论了Stream 类的基本构件（例如ACE_Stream 类，ACE_Module 类，ACE_Task 类：

● ACE_Stream 类：ACE_Stream 类定义了Stream 的应用程序接口。一个ACE_Stream对象包含与一个或多个层次相关服务的栈，这些层次相关服务为应用程序提供一个双向get/put 风格的接口以通过互连的组成一个特定Stream的Modules 发送和接收数据和控制消息。ACE_Stream类也实现了push/pop风格的接口，允许应用程序在运行时通过插入和移动下列ACE_Module类的对象来配置Stream：

● ACE_Module 类：ACE_Module 类定义了一个应用程序相关功能的单独层次。Stream 使用相互连接的一些ACE_Module 对象组成的。一个Stream中的ACE_Module 对象是松耦合的，通过传递有类型的消息合作。每个ACE_Module 对象包含一对指向应用程序指定的ACE_Task类的子类对象的指针。

如图3.5所示，两个缺省的ACE_Module对象（ACE_Stream_Head 和ACE_Stream_Tail）在Stream打开时自动安装。这两个ACE_Module 解释了预定义ASX框架控制消息和数据消息在运行时通过Stream 循环流动。ACE_Stream_Head 类在应用程序和Stream 之间提供了一个消息缓冲接口。ACE_Stream_Tail 类把从网络或一台伪设备到达的消息转换成规范的内部消息格式以被Stream 中的上层组件处理。同样，对于发出的消息，它将内部格式消息转换成网络消息。

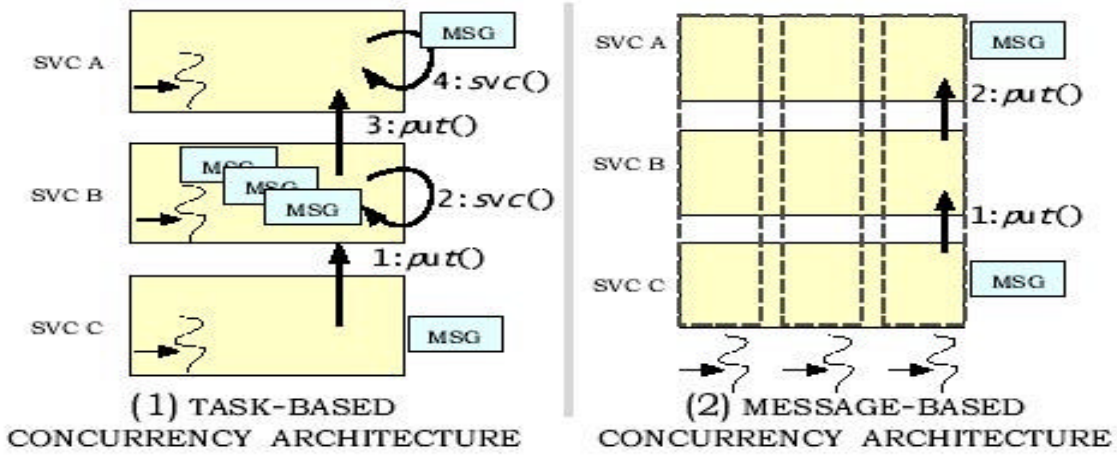

Figure 12: Alternative Methods for Invoking put and svc Methods

● ACE_Task 抽象类：ACE_Task 类是ACE用来产生用户定义的处理应用程序消息的主动对象和被动对象的核心机制。ACE Task 可以执行下列动作：

- 可以被动态链接；
- 可以作为I/O操作的一个分离点；
- 可以和多线程控制关联起来（即变成所谓的“主动对象”）；
- 可以为随后的处理存储消息在队列中；
- 可以执行用户定义的服务。

ACE_Task抽象类定义了一个通过派生类继承和实现的接口以提供应用程序相关功能。ACE_Task是一个抽象类因为它的接口定义了下面描述的抽象方法（open,close,put,svc）。将ACE_Task 作为抽象类定义增强了重用性，通过从那些继承自ACE_Stream提供的独立于应用程序的组件的应用程序相关的子类中分离出这些独立组件。同样，抽象方法的使用允许编译程序确保Task子类提供以下功能：

● 初始化方法和结束方法——从ACE_Task派生的子类必需实现open和close方法，这两个方法执行应用程序相关的ACE_Task 的初始化和结束动作。这些动作用于分配和释放资源，例如连接控制块，I/O处理器和同步锁。

ACE_Task 的定义和使用可以单独进行或者和ACE_Module 一起进行。当和ACE_Modules 同时使用时，它们是成对存储的：一个ACE_Task 子类进行对上传给ACE_Module 层的消息进行读处理，另一个则对下传给Module 层的的消息进行写处理。模Module 的ACE_Task读端和写端的Open 和 close 方法由ASX框架自动调用，当ACE_Module被插入或从Stream移走时。

● 应用程序相关的处理方法—除了open和close，ACE_Task 的子类还必需定义put 和svc方法。这些方法执行应用程序相关的消息处理功能。例如，当消息到达Stream的头部或尾部，就被escorted 通过一些互相连接的ACE_Task，作为调用每个ACE_Task的put 和/或svc方法的结果。

当位于Stream某一层的ACE_Task 传递一个消息给另一层的相邻的ACE_Task 时就调用put 方法。Put方法与它的调用者同步运行，即，它从调用put方法的Task 中借来控制线程。该控制线程引起从应用程序进程“upstream”

或从处理I/O 设备中断的线程池中“downstream”，或者由事件调度机制（如定时驱动调用队列用于触发一个面向连接的传输协议ACE_Module）调度到Stream内。如果ACE_Task作为一个被动对象执行（即总是从调用者借用控制线程），ACE_Task::put方法就是到达ACE_Task 的入口点，并且作为ACE_Task执行动作的内容。相反，如果ACE_Task 作为主动对象执行，ACE_Task::svc方法就执行应用程序相关的处理，与其它的ACE_Task 异步进行。与put 方法不同的是，svc 方法不直接从一个相邻的ACE_Task 调用。而是，通过一个与它的ACE_Task 关联的单独线程调用。该线程为svc 方法提供了一个执行内容和线程控制。该方法在事件循环中运行，同时等待到达ACE_Message_Queue 的消息。

在put 或 svc 方法的实现内部，一条消息会通过put_next ACE_Task方法向前传送到一个相邻的ACE_Task 。Put_next 调用驻留在相邻层的下一个ACE_Task的put 方法。这个put调用会从调用者借用控制线程并立即处理消息（即，图12(1) 中解释的同步进程）。相反，put 方法会将消息排到队列中并推迟处理给svc 方法处理，此时svc方法正在另一个控制线程执行。（即，图12(2) 中说明的异步处理方法）。如〔6〕中讨论的，选择特定的处理方法对性能和编程难度有很重要影响。

● 消息队列机制— 处理open,close,put 和 svc 抽象方法接口之外，每个ACE_Task 也包含一个ACE_Message_Queue。 ACE_Message_Queue 是一个ACE标准组件，用于在ACE_Task 之间传输信息，此外，当ACE_Task 作为一个主动对象执行时，它的ACE_Message_Queue 被用于缓冲一个数据消息序列并为随后的svc 方法中的处理控制消息。当消息到达时，svc 方法从队列中取出消息并执行ACE_Task 子类的应用程序相关处理任务。

在ACE_Message_Queue中会出现两种类型的消息：简单的和复合的。一个简单消息仅包含一个ACE_Message_Block 而一个复合消息包含多个连接在一起的ACE_Message_Block。复合消息通常由控制块和一个或多个数据块组成。一个控制块包含bookkeeping 信息（例如目的地址和长度字段），数据块在包含消息的实际内容。在Task 之间传递ACE_Message_Block 的开销是很小的，因为是通过传递指针而不是拷贝数据来完成的。

ACE_Message_Queue 包含一对高低water mark 变量，用于在一个Stream 内的相邻层次间实现层次间的流控制。高water mark 指示在ACE_Message_Queue流控制之前将要缓冲的消息的字节数。低water mark 指示前一个流控制ACE_Task 的level不是full。

4 ACE 实例

ACE 组件目前在几项研究和商业环境中使用，以增强配置的灵活性和通信软件组件的可重用性。为了解释ASX框架是如何在实际中应用的，本部分内容研究了当前采用了ACE的两个商业应用软件的结构：一个分布式日志工具和一个电信交换设备的分布式监视系统。

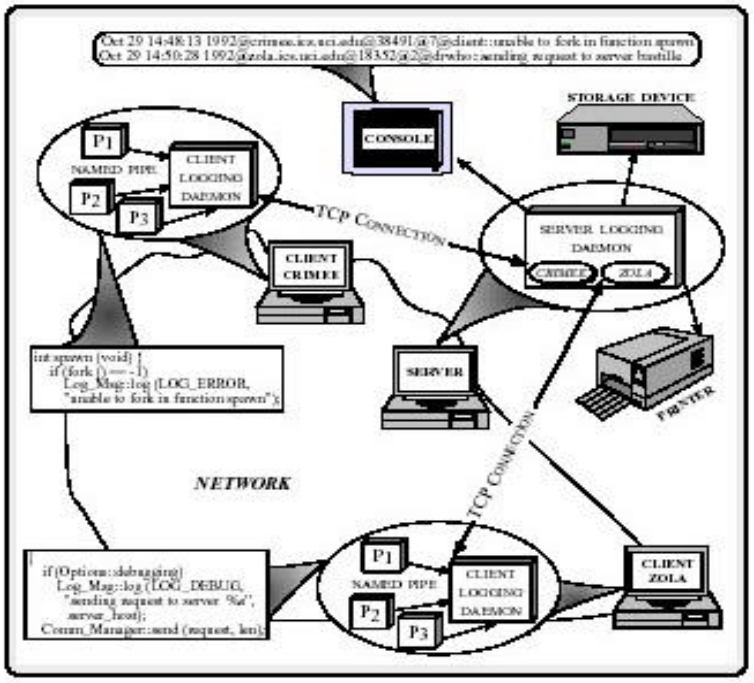

Figure 13: The Distributed Logging Facility

4.1 分布式日志服务的例子

调试分布式通信软件是一项挑战性的工作，因为诊断输出出现在不同的窗口和/或不同的远程主机系统。因此ACE提供了一个分布式日志工具简化调试和运行时的跟踪。这个工具现在被用于一个商业在线交易处理系统以为高速网络环境下一些工作站和多处理器数据库服务器提供日志服务。

如图13所示，分布式日志工具允许应用程序运行在多个客户主机上来发送日志记录到一个服务器日志后台程序，该后台程序运行在一个指定的服务器主机上。本部分内容集中于服务器后台程序部分的配置和结构，该结构和配置是基于Service Configurator和ACE_Reactor类别的。分布式日志工具的完整设计和实现在参考文献〔10〕中介绍。

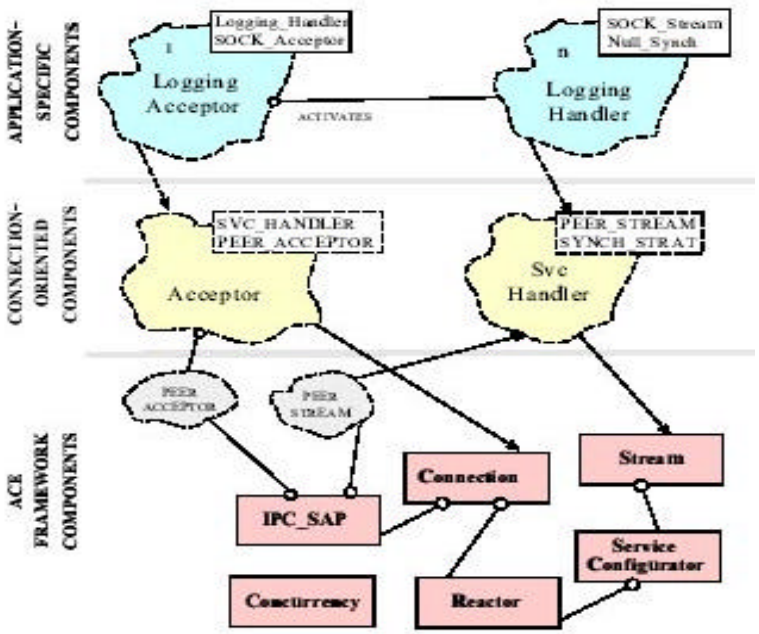

Figure 14: Class Components in the Server Logging Daemon

4.1.1 服务器日志后台程序组件

服务器日志后台程序组件是一个并发、多服务后台程序处理同时来自一个或多个主机的日志记录。服务器日志后台程序的面向对象设计可以分解成为几个模块化组件（如图14）执行定义好的任务。应用程序相关组件（Logging_Acceptor **和** Logging Handler**）负责处理从客户接收到的日志记录。面向连接的应用程序组件（**Acceptor**和**Client_Handler**）负责从客户端接收连接请求和数据。最后，独立于应用程序的**ASX**框架组件（**ACE_Reactor**，**Service Configurator **和** IPC SAP**类）负责执行**IPC**，直接动态链接，事件分类，服务调度和并发控制。**

Logging_Handler**子类是一个参数类型，负责处理从客户主机发送到服务器日志后台程序的日志记录。它的通信机制可以用**SOCK SAP **或** TLI SAP**包实例化，如下：**

```
class Logging_Handler : public Client_Handler <
#if defined (MT_SAFE_SOCKETS)
ACE_SOCK_Stream,
#else
ACE_TLI_Stream,
#endif /* MT_SAFE_SOCKETS */
ACE_INET_Addr>
{
/* ... */
};
```

Logging_Handler类继承自Event Handler（间接通过Client_Handler）而不是Service Object，因为它不是动态链接到服务器日志后台程序。

当日志记录从与一个特定Logging_Handler对象相关的客户主机到达时，ACE_Reactor自动调度该对象的handle_input方法。该方法在一个或多个输出设备上格式化并且显示这些记录（例如打印机，永久存储器，和控制台设备，如图13）。

Logging_Acceptor子类也是一个参数化类型，负责接收从客户主机到达的连接。

```
class Logging_Acceptor :
public Client_Acceptor<Logging_Handler,
#if defined (MT_SAFE_SOCKETS)
ACE_SOCK_Acceptor,
#else
ACE_TLI_Acceptor,
#endif /* MT_SAFE_SOCKETS */
ACE_INET_Addr>
{
/* ... */
};
```

由于Logging_Acceptor类继承自ACE_Service_Object（间接来自ACE_Acceptor基类），它可以动态链接到服务器后台日志程序并且通过服务器日志后台程序的svc.conf 配置文件进行处理。同样，由于Logging_Acceptor间接继承自ACE_Event_Handler接口，它的handle_input方法将自动被ACE_Reactor调用当连接请求到达时。当一个连接请求到达，Logging_Acceptor子类定位Logging_Handler对象并注册对象。

通过分离连接建立功能和日志记录接收功能到两个分开的类层次中，分布式日志工具的模块化，重用性和可配置性得到了显著增强（如图14所示）。这种分离允许ACE_Acceptor类可被其它类型的面向对象的服务重用。尤其是，提供完全不同的处理功能时，只有ACE_Client_Handler部分需要重新实现。此外，参数化类型的使用消除了对特定IPC机制的依赖。

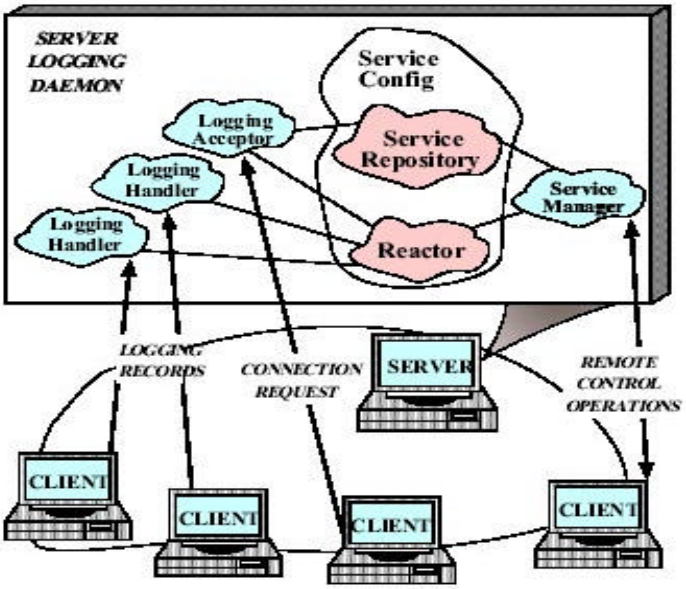

Figure 15: ACE Components in the Distributed Logging Facility

4.1.2 服务器日志后台程序配置

ASX框架使用Service Configurator 使日志服务可以动态和静态配置到服务器日志后台程序。动态配置服务可以在运行时插入，修改，或移动，以此提高服务灵活性和扩展性。下面的svc.conf文件入口点用来动态配置日志服务到日志服务后台程序：

```
dynamic Logger Service_Object *
        ./Logger.so:_alloc() "-p 7001"
```

<svc-name> 标识 Logger 指定的服务名，用于安装和运行时，以确认相应的ACE_Service_Repository 内的 Service Object对象。Service Object是_alloc方法的返回类型，位于共享对象文件，用路径名./Logger.so来指示。Service Configurator框架定位和动态链接共享对象文件到后台日志程序的地址空间。服务定位也指定从Service Object继承的应用程序相关对象的名字。在这种情况下，_alloc函数被用于动态定位一个新的Logging_Acceptor对象。该行剩下的内容（“ -p 7001 ”）表示一个应用程序相关的配置参数集。这些参数用以argc/argv形式的命令行参数传递到init方法。Logging_Acceptor类的init方法将“ -p 7001 ”作为端口号解释，服务器后台日志程序监听该端口等待客户的连接请求。

静态配置服务对一个后台程序始终是可行的当该程序第一次执行时。例如，Service Manager是一个标准的Service Configurator框架组件，客户用它来获得一个活动的后台服务的列表。Svc.conf中的下列入口用于在初始化阶段静态配置Service Manager服务到服务器后台日志程序：

static ACE_Svc_Manager “ -p 911 ”

为了静态直接工作，实现ACE_Svc_Manager服务的对象代码必须静态链接到

主后台驱动执行程序。此外，在动态配置前，ACE_Svc_Manager对象必须插入到Service Repository（这由ACE_Service_Config构造函数自动完成）。由于这些限制，一个静态配置服务如果不先从Service Repository移出的话就不可以在运行时重新配置。服务器日志后台程序的主驱动程序用下列代码实现：

```
int
main (int argc, char *argv[])
{
ACE_Service_Config loggerd;
// Configure server logging daemon.
if (loggerd.open (argc, argv) == -1)
return -1;
// Perform logging service.
loggerd.run_reactor_event_loop ();
return 0;
}
```

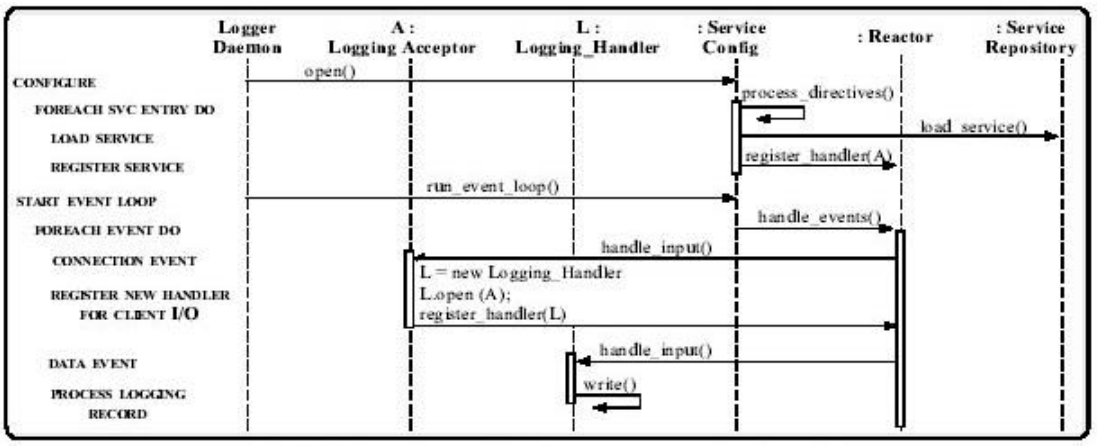

Figure 16: Interaction Diagram for the Server Logging Daemon

图16描述了在不同的框架和应用程序相关对象直接的运行时的交互和合作以提供日志服务。后台配置在ACE_Service_Config::open方法中执行。该方法consult下面的svc.conf文件，该文件指定了要配置到后台程序的服务：

```
static ACE_Svc_Manager -p 911
dynamic Logger Service_Object *
./Logger.so:_alloc() "-p 7001"
```

每个进入svc.conf文件的服务配置通过插入指定的ACE_Service_Object到ACE_Service_Repository和注册服务对象Handler 的ACE_Event_Handler部分来处理。

当所有的配置活动都完成后，主驱动程序调用ACE_Service_Config的run_reactor_event_loop方法。该方法进入一个事件循环连续调用ACE_Reactor::handle_events服务调度方法。如图10所示，该调度函数阻塞等待事件的发生（例如连接请求或I/O事件）。当这些事件发生时，ACE_Reactor

自动调度预先注册的事件处理程序来执行指定的应用程序相关服务。ASX框架也响应在运行时触发后台配置的外部事件。上面列出的动态配置步骤在一个正在执行的基于ASX的后台程序接收到一个预先指定的外部事件（如UNIX SIGHUP信号）时运行。依赖于svc.conf文件的更新内容可以添加服务，挂起服务，重新启动服务或将服务从后台程序移除。

ASX框架的动态配置机制使开发人员可以修改服务器日志后台程序的功能或调整性能而不需要重新开发和重新安装。例如，调试一个不完善的日志服务的实现只需要动态重装一个功能上等价的服务，包含附加的手段以隔离错误源。（？？？）注意，这个重装过程可以不需要修改，不需要重新编译和链接或重新启动服务器后台日志程序。

4.2 分布式 PBX 监视系统

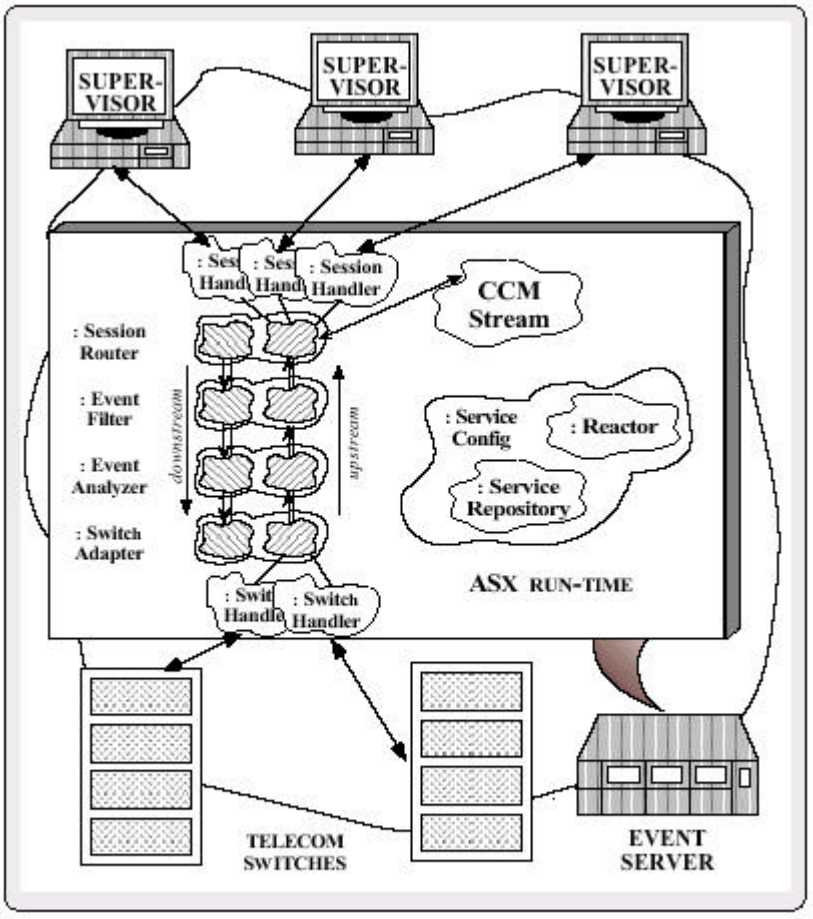

Figure 17: ASX Components for the PBX Application

图17解释了一个用ASX框架组件实现的专用分组交换机（PBX）电信交换监视系统客户机/服务器结构。在这个分布式电信系统中，服务器接收和处理一个或

多个通过高速通信链接到服务器的PBX产生的状态信息。该服务器传送和转发状态信息到客户端系统，端系统用图形显示该信息给端用户。端用户通常是一个检查人员，他使用PBX状态信息监视人员的工作和预报资源定位以满足客户需要。

附属于服务器的PBX设备由Device_Adapter ACE_Module来控制。该ACE_Module将服务器的其它部分从PBX相关的通信特性中屏蔽起来。Device_Adapter ACE_Module的读部分维护一个Device_Handler对象的集合（每个PBX对应一个对象），对象负责解析和传输到达的设备事件到一个规范的独立于PBX的消息对象，该消息对象是建立在一个灵活的消息管理类之上的，在参考文献〔6〕中介绍了该消息管理类。

在初始化之后，到达的规范的消息对象被传递给Event_Analyzer ACE_Module的读端。该Module实现了服务器端应用程序相关的功能。一个在Event Analyzer内部维护的地址表被用于决定哪个客户接收哪个消息。在Event_Analyzer决定了正确的目标之后，消息对象被转发到Muticast_Router ACE_Module的读端。

Muticast_Router ACE_Module是一个可重用组件，它屏蔽了应用程序相关服务器代码的其它部分和客户机/服务器交互部分以及通信协议的选择部分。客户通过建立一个与Multicast_Router ACE_Module的连接同意接收服务器的事件。Multicast_Router ACE_Module的写端接收客户的连接请求，并生成一个单独的Client_Handler对象来管理每个客户连接。该Client_Handler对象处理所有与其关联的客户机和服务器的随后的数据传输和控制操作。一旦客户机连接上服务器，它就表示要监视的PBX事件的类型。从这点上说，当一个Multicast_Router的读端从Event_Analyzer接收到一个消息对象时，它自动广播消息给所有同意接收封装在消息对象内的特定类型事件的客户机。

ACE_Service_Config对象被服务器用来控制Stream ACE_Module组件的初始化和终止，该组件可在安装时静态配置或在运行时动态配置。ACE_Service_Config对象包含一个ACE_Reactor实例的事件分离程序用来调度到达的客户机消息到正确的Client_Handler或Device_Handler事件处理程序。从客户机到达的控制消息沿Stream的写端向下传输，起始于Multicast_Router，通过Stream ACE_Module的写端到达Device_Adapter，Device_Adapter发送消息到正确的PBX设备。同样，Reactor监测到从PBX设备到达的事件并向上调度它们到Device_Adatpter ACE_Module。

4.3 Server Configuration

组成PBX服务器的ACE_Modules 可以在任何时候配置到服务器。ASX框架通过svc.conf配置脚本驱动直接动态链接提供了这种灵活度。下面的配置脚本指示了哪个服务将被动态链接到服务器的地址空间：

```
stream Server_Stream dynamic
   STREAM * /svcs/Server_Stream.so : _alloc()
{
   dynamic Device_Adapter
```

```
Module * /svcs/DA.so:_alloc() "-p 2001"
dynamic Event_Analyzer
Module * /svcs/EA.so:_alloc()
dynamic Multicast_Router
Module * /svcs/MR.so:_alloc() "-p 2010"
}
```

这个配置脚本指示了层次相关的服务的动态链接和推到Server Stream的顺序。在应用程序初始化期间，Service Config类解析这个配置脚本并执行每个入口点的指示。

Server Stream由三个动态配置到服务器的ACE_Module组成（Device_Adapter，Event_Analyzer和Multicast_Router）。指示的共享对象文件动态链接到服务器。一个Module对象随后通过调用_alloc函数从共享对象库提取出来。如下所示，如果需要（例如，安装一个ACE_Modules的更新版本），这些Modules可以随后更新和重新链接而不用完全终止PBX服务器的执行。

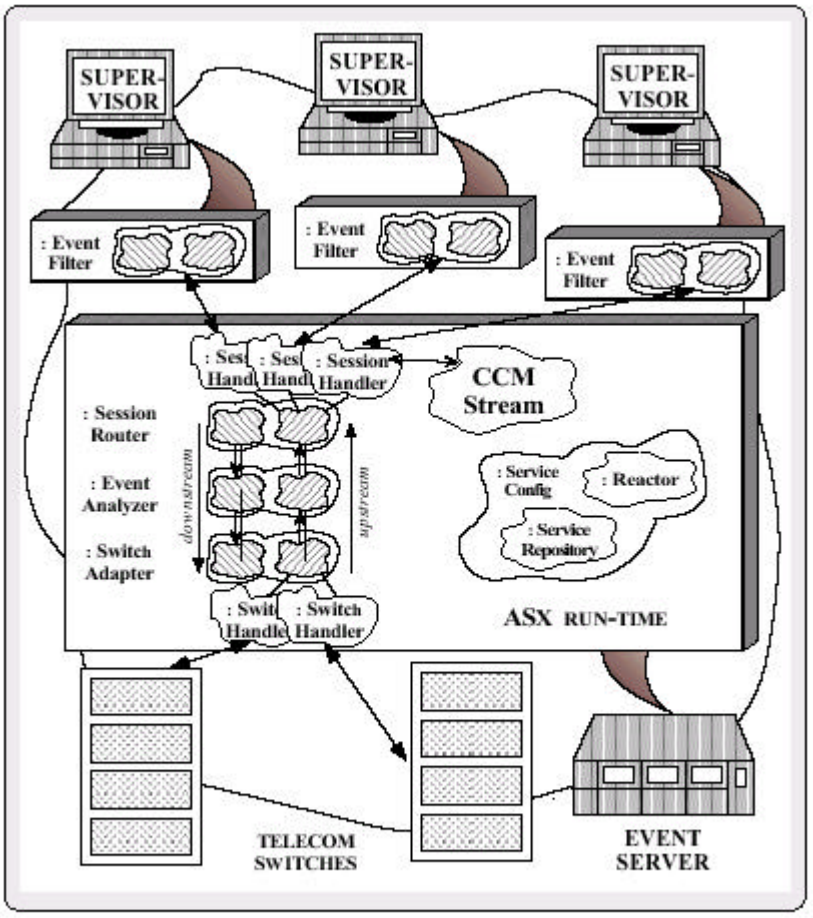

Figure 18: Reconfiguration of PBX Monitoring System

4.4 服务器重新配置

静态配置服务到通信应用程序有一些缺点。例如，如果太多服务配置到服务器端并且许多活动客户同时访问这些服务，就会产生性能瓶颈。配置太多服务到客户端也会遭遇性能瓶颈，因为客户经常执行在功能较少的终端系统。总之，很难事先决定适当的应用程序服务的参与者，因为过程特性和工作负载是随时间而变化的。因此，ASX框架的一个基本目标是开发面向对象的服务配置机制以允许开发人员将对哪个服务在客户端运行，哪个在服务器端运行的考虑推迟到开发周期的后期。

为了方便灵活的重新配置，ASX框架提供的运行时控制环境使开发人员可以在安装时静态改变应用程序配置或在运行时动态配置。这是有用的，因为不同的操作系统/硬件平台和不同的网络特性通常需要不同的服务配置。例如，在一些配置中服务器执行大部分的工作，而在另一些环境中，客户执行大部分的工作。此外，不同的终端系统配置可能适合不同的环境（例如使用的是不是多处理器服务器平台或高速网络）。图18解释了图17所示的配置如何改变以更加有效地在服务器造成了主要瓶颈的分布式环境中执行。

重新配置过程通过下面的脚本完成：

```
suspend Server_Stream
stream Server_Stream
{
   remove Event_Analyzer
}
remote "-h all -p 911"
{
stream Server_Stream
{
dynamic Event_Analyzer
Module * /svcs/EA.so : _alloc()
}
}
resume Server_Stream
```

这个新脚本将处理功能从服务器移动到客户端，通过动态断开ACE_Modules与服务器Stream的链接和动态链接这些模块到客户Stream。

ASX框架取代了一个先前的使用ad hoc技术的结构（例如参数传递和共享内存）来在服务器内相关的服务中交换信息。与先前的方法相反的是，高度统一的ACE_Module互连机制极大的简化了可移植性和配置性。

5 结论

ADAPTIVE Communication Environment是一个包含面向对象组件的工具包，它通过具体化成功的设计模式和软件结构减少了分布式软件的复杂性。ACE在可重用的面向对象组件和框架中实现了通常的通信相关的活动（例如本地和远程IPC，事件分离和服务处理调度，服务初始化配置机制等）。

ACE可用从WWW上通过下面的地址获得：www.cs.wustl.edu/~schmidt/ACE.html．其中包含了源码，文档和在华盛顿大学开发的实例测试。ACE目前已经在许多公司的商业软件中得到应用，包括Bellcore，Siemens，DEC，Motorola，Ericsson，Kodak和McDonnell Douglas。ACE可用于Win32，以及Unix的大多数版本（如SunOS4.x，HPUX等）和POSIX系统（VxWorks和MVS OpenEdition）。ACE提供了C++和Java两个实现版本。