第1章 ACE自适配通信环境：用于开发通信软件的面向对象网络编程工具包

Douglas C. Schmidt

摘　要

ACE自适配通信环境（Adaptive Communication Environment）是一种面向对象（OO）的工具包，它实现了通信软件的许多基本的设计模式。ACE的目标用户是在UNIX和Win32平台上开发高性能通信服务和应用的开发者。ACE简化了使用进程间通信、事件多路分离、显式动态链接和并发的OO网络应用和服务的开发。通过在运行时将服务与应用动态链接进应用，并在一个或多个进程或线程中执行这些服务，ACE使系统的配置和重配置得以自动化。

本论文描述ACE的结构和功能，并使用来自像电信、企业级医学成像和WWW服务这样的领域的例子阐释核心的ACE特性。ACE可以自由使用，并正在被用于许多商业项目（比如爱立信、Bellcore、西门子、摩托罗拉、柯达，和McDonnell Douglas），以及许多学院和工业研究项目。ACE已被移植到多种OS（操作系统）平台上，包括Win32和大多数的UNIX/POSIX实现。此外，同时有C++和Java版本的ACE可用。

1.1 介绍

1.1.1 问题：分布式软件危机

对健壮的和高性能的分布式计算系统的需求一直在稳定地增长。这些类型的系统的例子包括全球个人通信系统、网络管理平台，企业级医学成像系统、在线金融分析系统，以及实时航空控制系统。对于以下方面来说，分布式计算是一种有前途的技术：通过连接性和相互配合促进协作；通过并行处理改善性能；通过复制改善可靠性和可用性；通过模块性改善可伸缩性和可移植性；通过动态配置和重配置改善可扩展性；以及通过资源共享和开发系统提高成本效用。

尽管分布式计算提供了许多潜在的好处，开发通信软件仍然是昂贵而易错的。面向对象编程语言、组件和构架（Framework）是被广泛鼓吹的、用以降低软件成本并提高软件质量的技术。去除那些过分的宣传，OO的主要好处源于对模块性和可扩展性的强调，它将易变的实现细节封装在稳定的接口后面，并增强了软件的复用。

多年来，在某些已被广泛探索的领域中的开发者已经成功地应用了OO技术和工具。例如，Microsoft MFC GUI构架和OCX组件是PC平台上用于创建图形商业应用的事实上的工业标准。尽管这些工具有着自身的局限，它们仍然演示了复用通用构架和组件的生产效率优势。

在像电信、医学成像、航空控制和在线事务处理这样的更复杂的领域中，软件开发者历来就缺少标准的、成型的中间件组件。结果，开发者在很大程度上是从头开始构建、验证和维护软件系统。在一个政府经济干预减少的艰难的全球竞争时代，这样的作坊式开发过程正在变得难以容忍的昂贵和费时。在业界，这样的情形导致了一场“分布式软件危机”：计算硬件和网络在变小、变快、变得更为便宜；而分布式软件的开发和维护在变大、变慢、变得更为昂贵。

构建分布式软件的挑战源于与分布式系统相关联的*固有的*和*非固有的*复杂性[1]。固有的复杂性源于开发分布式软件的基本的挑战，其中主要的有：检测和恢复网络及主机失败、最小化通信响应延迟的影响，以及确定服务组件和工作负载在网络的处理单元上的最优划分。

非固有的复杂性源于用以开发分布式软件的工具和技术的局限。例如，许多标准的网络机制（比如socket[2]和TLI[3]）和可复用组件库（比如X windows和Sun RPC）缺乏类型安全的、可移植的、可重入的和可扩展的*应用编程接口*（API）。同样地，通用网络编程接口，如socket和TLI，使用弱类型的整型句柄，可能会导致微妙的运行时错误[4]。

复杂性的另一来源起因于算法分解的普遍使用[5]，它致使软件系统不可扩展和不可复用[6]。尽管图形用户接口（GUI）普遍采用面向对象技术构建，典型的分布式软件通常仍然使用算法分解进行开发。在一些流行的网络编程教科书[7, 8, 3]中的例子基于面向算法的设计和实现技术，从而更加恶化了前述问题。

可扩展性和最大限度复用的缺乏对于复杂的分布式软件是特别成问题的。可扩展性是确保服务和特性的及时修改和增强的基本要求。复用是有效利用专家开发者的领域知识、以避免重新开发和重新验证“反复出现的需求和软件挑战的通用解决方案”的基本要求。

1.1.2 解决方案：面向对象的设计模式和构架

面向对象的设计模式和构架有助于减少对分布式软件的核心概念和抽象的昂贵的重新发现和发明，它们因此而备受重视。模式提供了一种封装设计知识的方法，这些设计知识为标准的分布式软件开发问题提供解决方案[9]。例如，模式对于描述重复出现的“*微型结构*”（比如反应堆（Reactor）[10]和主动对象（Active Object）[11]）十分有用，这些微型结构是对一些已被证明可用于构建分布式通信软件的通用对象结构的抽象。但是，被文档化为模式的抽象并不直接产生可复用代码。因此，有必要通过*构架*的创建和使用来增加对模式的研究。

通过集成成组的抽象类，并定义这些类的协作的标准途径，构架为应用提供了可复用的软件组件[12]。构架实例化设计模式族，以帮助开发者避免对通用分布式软件组件的昂贵的重新发明。其成果是“半完成”的应用骨架，它可以通过继承和实例化构架中的可复用“积木”组件来进行定制。因为构架与关键的分布式编程任务（比如服务初始化、错误处理、流控制、事件多路分离、并发控制）紧密地集成在一起，复用的范围可以显著地大于使用传统函数库，甚或是通常的OO类库。

本论文被组织如下：1.2给出ACE工具包的结构和功能的综述；1.3详细描述ACE C++包装组件和较高级的ACE构架组件及模式；1.4检查若干使用ACE构建的网络应用的实现；还有1.5给出结束语

1.2 ACE综述

为阐释OO模式和构架是怎样被成功地应用于分布式软件的，本论文考查自适配通信环境（ACE）[6]。ACE是可以自由使用的OO工具包，其中包含有丰富的、可跨越广泛的OS平台执行通用网络编程任务的可复用包装、类属，以及构架。ACE提供的任务包括：

- *事件多路分离和事件处理器分派*[13, 14, 10, 15]；
- *连接建立和服务初始化*[16, 17, 18]；
- *进程间通信*[19, 4]*和共享内存管理*；
- *分布式通信服务的动态配置*[20, 21]；
- *并发/并行和同步*[22, 23, 11, 24]；
- *更高级的分布式服务组件*（比如名字服务、事件服务、日志服务、时间服务和令牌服务）。

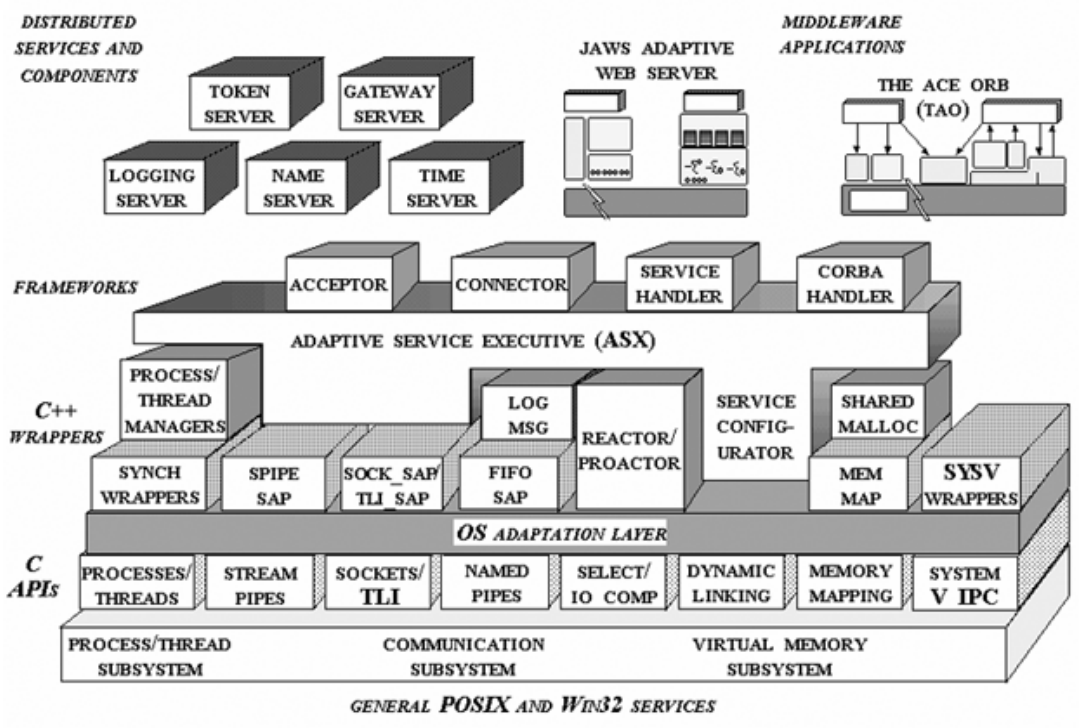

图1-1 ACE自适配通信环境中的组件

ACE工具包的设计采用分层的体系结构。图1-1演示了ACE组件间的纵向和横向关系。ACE的较低层是封装现有的OS网络编程机制的OO*包装*（wrapper）。ACE的高层扩展这些包装，以提供OO*构架*和*组件*、覆盖更为广泛的面向应用的网络任务和服务。这一部分的余下部分给出对ACE中类属的结构和功能的综述（如图1-2所示）。1.3提供了对ACE的网络编程特性和组件的深入讨论。

贯穿本论文，ACE组件通过Booch表示法[5]来进行图解。实心矩形表示类属，它将一定数量的相关类合成进一个公共的名字空间。实心云表示对象；如嵌套则表示对象间的合成关系；而无方向的边表示在两个对象间存在某种类型的链接。虚线云表示类；有向边表示类之间的继承关系；而一端有小圆圈的无向边表示两个类之间的合成或是使用关系。在三角形内标记的“A”标识一个类为*抽象类*[25]。抽象类不能被直接实例化，而必须被子类化。在实例化抽象类子类的任何对象之前，该子类必须提供所有抽象方法的定义。

1.2.1 ACE OS适配层

ACE的源码树含有超过85,000行C++代码。其中大约9,000行代码（也就是，大约为总数的10%）为*OS适配层*所特有。该层将ACE的较高层和与下列OS机制相关联的平台特有的依赖屏蔽开来：

- 多线程和同步
- 进程间通信
- 事件多路分离
- 显式动态链接
- 内存映射文件和共享内存

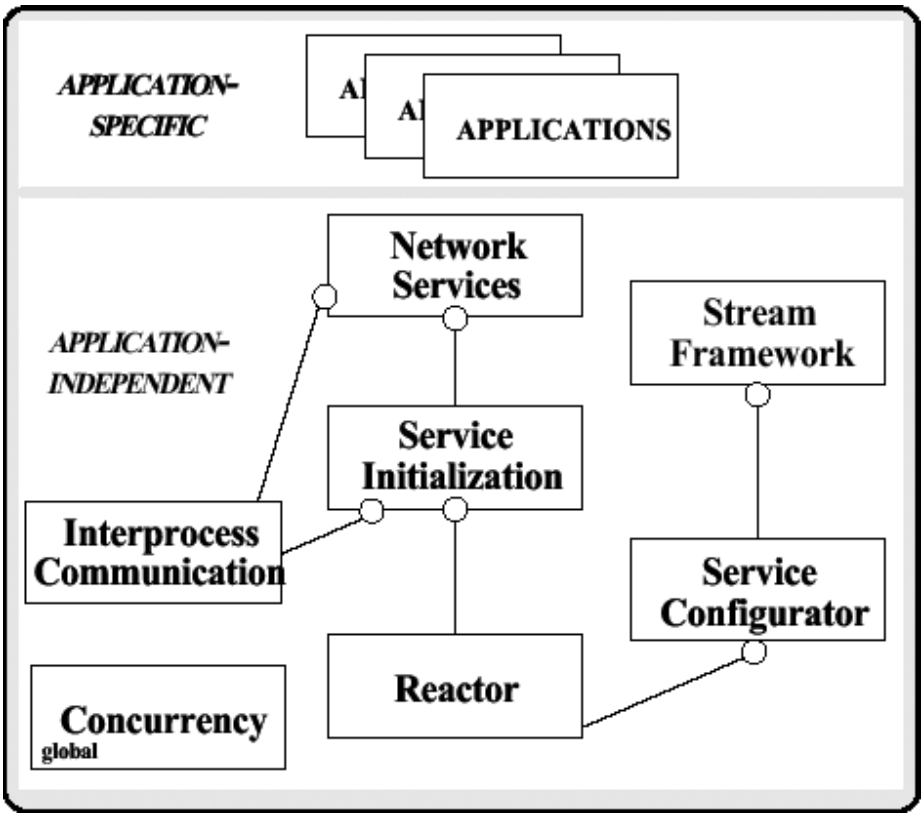

图1-2 ACE中的类属

1.2.2 ACE OO包装

在OS适配层之上是许多OO包装，它们封装并增强在像Win32和UNIX这样的现代操作系统上可用的并发、进程间通信（IPC）、以及虚拟内存机制（在图1-1底部演示）。应用可以通过有选择地继承、聚合（aggregating）、和/或实例化下列ACE包装类属来合并和编写这些组件：

- IPC SAP - 它封装本地和/或远地的IPC服务访问点（IPC SAP）机制，比如socket、TLI、UNIX FIFO和STREAM管道，以及Win32命名管道[19, 4]；
- **服务初始化** - ACE提供一组连接器（Connector）和接受器（Acceptor）组件[18]，分别使主动和被动的初始化角色与一旦初始化完成后通信服务执行的任务去耦合（decouple）。

- **并发机制** - ACE抽象较低级的OS多线程和多进程机制（比如互斥体和信号量[22]），以创建较高级的OO并发抽象（比如主动对象Active Object[11]）；
- **内存管理机制** - ACE内存管理组件为管理共享内存和局部内存的动态分配和释放提供了灵活和可扩展的抽象；
- CORBA**集成** - ACE可与CORBA实现集成在一起（比如单线程和多线程Orbix）。

通过采用类型安全的OO接口封装OS通信、并发和虚拟内存机制，OO包装的使用提高了应用的健壮性。这也减少了应用直接访问用弱类型C接口编写的底层OS库的需求。因此，像C++和Java这样的OO语言的编译器可以在编译时、而非运行时检测类型系统违例。ACE的C++版本大量使用内联（inlining），以消除包装层提供额外的类型安全性和抽象所导致的性能下降。

1.2.3 ACE构架

ACE含有一个更高层的网络编程构架，集成并增强了较低层的OS包装。该构架支持由应用服务组成的并发网络看守的动态配置。ACE的构架部分包括以下类属：

- **反应堆（**Reactor**）** - ACE反应堆[10]提供可扩展的、面向对象的多路分离器，它分派处理器、以响应多种类型的事件（例如，基于I/O的、基于定时器的、基于信号的，以及基于同步的事件）；
- **服务配置器（**Service configurator**）** - ACE服务配置器[21]支持这样的服务构造：其服务可在安装时和/或运行时动态配置；
- **流（**Stream**）** - ACE流组件[6]简化了由一或多个层次相关的服务（比如协议栈）组成的并发通信应用的开发。

1.2.4 ACE网络服务组件

除了包装和构架，ACE还提供了一个网络服务组件的标准库。这些组件在ACE中扮演两种角色：

1. 它们演示怎样利用ACE IPC包装、反应堆、服务配置器、服务初始化、并发、内存管理，以及流组件；
2. 它们为常见的分布式系统任务提供了可复用组件，比如日志[13]、名字、锁定和时间同步[21]；

当与OO语言特性（比如类、继承、动态绑定和参数化类型）和设计模式（比如抽象工厂（Abstract Factory）、构建器（Builder）和服务配置器）相结合时，可复用的ACE组件促进了通信服务和应用的开发，无需修改、重编译、重链接、甚至不用重启运行中的软件，它们就可以被更新和扩展[20]。

1.3 ACE**组件详述**

1.3.1 IPC_SAP：本地和远地IPC机制

ACE提供了植根于IPC SAP（“进程间通信服务访问点”）基类的一个类属“森林”。标准的基于I/O句柄的OS本地和远地IPC机制提供了面向连接和无连接的协议，IPC SAP对这些机制进行了封装。如图1-3所示，该类属“森林”包括SOCK SAP（封装socket API）、TLI SAP（封装TLI API）、SPIPE SAP（封装UNIX SunOS 5.x STREAM管道API），以及FIFO SAP（封装UNIX命名管道API）。

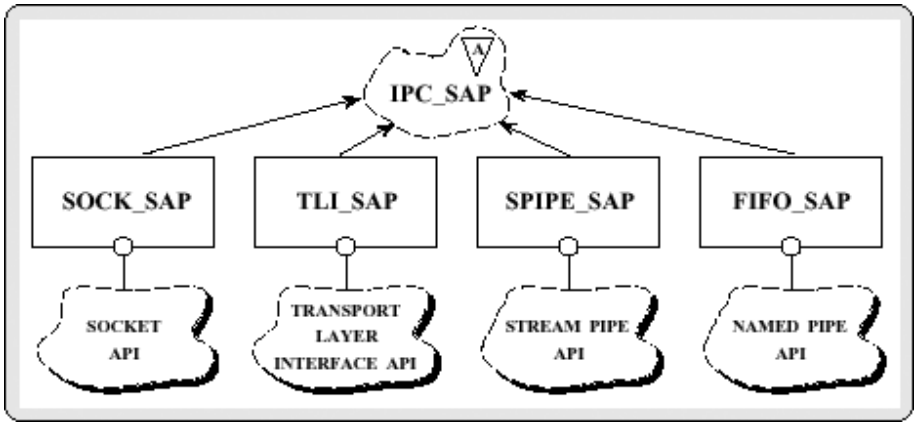

图1-3 IPC SAP类属关系

每一类属都被组织成一个继承层次。所有子类都提供定义良好的、本地或远地通信机制的子集的接口。在一个层次里的子类共同包含了一种特定通信抽象的全部功能（比如Internet域或UNIX域协议族）。类的使用（相对于单独的函数）帮助简化了网络编程：

- *使应用与易错的细节相屏蔽*：例如，图1-3中所示的ACE_Addr类层次通过类型安全的OO接口支持若干不同的网络寻址格式，而非直接使用麻烦而易错的基于C的struct sockaddr数据结构。
- *合并若干操作、以形成单一操作*：例如，SOCK Acceptor的构造器执行创建被动模式服务器端点所需的多个socket系统调用（比如socket、bind和listen）。
- *将IPC机制参数化进应用*：类构成了通过所需的IPC机制的类型来参数化应用的基础。如1.3.1.2所讨论的，这样有助于改善可移植性。
- *增强代码共享*：基于继承的层次划分增加了在多种IPC机制间共享的通用代码的数量（比如像fcntl和ioctl这样的低级OS设备控制系统调用的OO接口）。

下面的部分讨论IPC SAP中的每个类属。

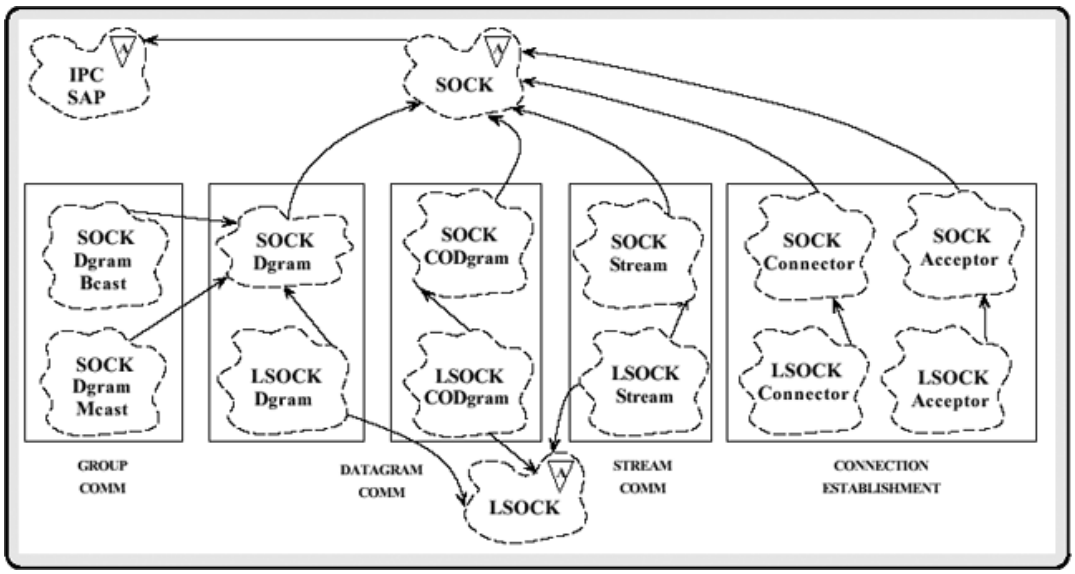

图1-4 SOCK SAP类属

1.3.1.1 SOCK SAP

SOCK SAP[4]类属为应用提供Internet域和UNIX域协议族[8]的面向对象的接口。通过继承或实例化图1-4所示的适当的SOCK SAP子类，应用可以访问底层的Internet域或UNIX域的socket类型的功能。ACE_SOCK *子类封装Internet域的功能，而ACE_LSOCK *子类封装UNIX域的功能。如图1-4所示，这些子类可进一步划分为（1）*Dgram组件（提供不可靠、无连接、面向消息的功能） vs. *ACE_Stream组件（提供可靠、面向连接的字节流的功能）和（2）ACE_*_Acceptor组件（提供通常用于服务器的连接建立功能） vs. *Stream组件（提供同时用于客户和服务器的双向字节流传输功能）。

使用OO包装来封装socket接口有助于（1）在编译时检测许多微妙的应用类型系统违例，（2）推动传输层接口的平台无关性，以改善应用的可移植性，以及（3）极大地减少应用代码的数量和花费在较低级网络编程细节上的开发工作。为演示后面一点，下面的例子程序实现了一个简单的客户应用，使用ACE_SOCK_Dgram_Bcast类来向局域网子网中所有在指定的端口号上侦听的服务器广播消息：

```
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);
}
```

把这个简洁的例子与直接使用socket接口实现广播所需的成打的C源码相比较很有启发意义。

1.3.1.2 TLI SAP

TLI SAP类属提供系统V传输层接口（Transport Layer Interface）的OO接口。TLI的TLI SAP继承层次几乎与socket的SCOK SAP包装相同。主要的区别是TLI和TLI SAP并不定义UNIX域协议族的接口。此外，目前TLI没有被移植到Win32平台。

通过结合C++特性（比如缺省参数值和模板）和tirdwr（read/write兼容性STREAM模块），开发可在编译时参数化、以在基于socket或基于TLI传输接口上正确操作的应用变得相对直截了当了。例如，下面的代码演示怎样将C++模板用于参数化应用所需的IPC机制。这些代码是从在1.4.1中描述的分布式日志工具中摘录出来的。在下面的代码中，一个派生自ACE_Event_Handler的子类被一种特定类型的传输接口及其相应的协议地址类参数化：

```
/* Logging_Handler header file */
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_;
}
```

如下所示，取决于底层OS平台的特定属性（比如是基于BSD的SunOS 4.x还是基于系统V的SunOS 5.x），日志应用可以实例化Client Handler类、以使用SOCK SAP或是TLI SAP：

```
/* 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 */

{

/* ... */

};
```

在开发必须具有跨平台可移植性的应用时，这种基于模板的方法所带来的更多的灵活性极其有用。特别地，因为SunOS 5.2的socket实现不是线程安全的，而SunOS 4.x的TLI实现含有许多严重缺陷，在SunOS的变种上进行跨平台开发时，必须具有根据传输接口来参数化应用的能力。

TLI SAP还使应用与许多TLI接口的特性相屏蔽。例如，在一个并发服务器中，在qlen > 1[3]的情况下，要正确处理t_listen和t_accept的非直观和易错的行为，需要编写一些微妙的应用级代码；而在TLI Acceptor类的accept方法中这些代码被封装起来。该方法接受来自客户的到来的连接请求。通过使用C++缺省参数值，基于TLI SAP和基于SOCK SAP的应用的调用accept方法的标准方法在语法上是相同的。

1.3.1.3 SPIPE SAP

SPIPE SAP类属为高性能本地IPC提供OO包装。在Win32平台上，SPIPE SAP类属在命名管道之上实现。Win32命名管道机制主要用于在同一机器的进程间高效地传输数据。对于本地IPC，它通常比使用socket更为高效[27]

在UNIX平台上，SPIPE SAP类属通过已安装（mounted）的STREAM管道和connld[28]来实现。SunOS 5.x提供fattach系统调用，将管道句柄安装到UNIX文件系统中的指定位置。通过将connld流模块推入管道已安装的一端，可以创建服务器应用。当与服务器运行在同一主机上的客户应用随后打开与已安装的管道相关联的文件时，客户和服务器都将获得一个I/O句柄，标识一个唯一的、非多路服用和双向的通信信道。

SPIPE SAP继承层次是SOCK SAP及TLI SAP所用层次的“镜像“。它提供与SOCK SAP ACE_LSOCK *类（它们自己封装了UNIX域的socket）相近似的功能。但是，在SunOS 5.x平台上，SPIPE SAP比ACE_LSOCK*接口更为灵活，因为它使得STREAM模块可以分别被“推入”和“弹出”SPIPE SAP终点。SPIPE SAP还支持在同一主机上执行的进程和/或线程间的字节流和按优先级排序的面向消息的数据的双向传送[29]。

1.3.1.4 FIFO SAP

FIFO SAP类属封装UNIX命名管道机制（也称为FIFO）。不像STREAM管道，命名管道只提供从一或多个发送者到单一接收者的单向数据信道。而且，来自不同发送者的消息都被放入同一个通信信道。因而，在每一消息中必须明确地包含某种类型的多路分离标识符（demultiplexing identifier），以使接收者能够确定是哪一个发送者发送的消息。

SunOS 5.x中的基于STREAM的命名管道同时实现提供面向消息和面向字节流的数据传送机制。相反，某些平台（比如SunOS 4.x）只提供面向字节流的命名管道。因此，除非总是使用定长消息，在SunOS 4.x中通过命名管道发送的消息必须通过某种形式的字节计数、或是特别的结束符来加以区分，以使接收者能从通信信道字节流中提取消息。为减少这样的局限，ACE FIFO SAP实现包含有对SunOS 5.x的面向消息的机制进行模拟的逻辑。

1.3.1.5 **其他通信机制**

除了封装像socket和TLI这样的基于句柄的I/O通信机制，ACE还提供了内存映射文件和系统V UNIX IPC机制的OO封装：

- **内存映射文件：**ACE_Mem_Map类提供Win32和UNIX（比如mmap系统调用族）上可用的内存映射文件机制的OO接口。这些调用利用底层的OS虚拟内存机制[30]来将文件映射到进程的地址空间。映射文件的内容可直接通过指针访问。指针接口常常比通过标准read/write I/O系统调用间接地访问数据块要更为方便和高效。此外，内存映射文件的内容可以很方便地在两个或多个进程间共享。

  现有的Win32和UNIX的内存映射文件接口的风格有一点巴洛克（baroque，过分雕饰）。例如，开发者必须手工完成许多“簿记”工作（比如显式地打开文件、确定它的长度、执行多种映射，等等）。相反，ACE_Mem_Map OO封装提供的接口采用了缺省值和多种有若干类型特征（type signature）变体的构造器（例如，“从已打开的文件句柄映射”、“从文件名映射”，等等），以简化常见的内存映射文件的使用模式。

  例如，下面的程序使用ACE_Mem_Map OO封装来映射一个通过命令行指定的文件，并将它的各行反向打印出来：

```
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;

}
```

把这个OO包装接口的使用与直接使用像read这样的I/O系统调用所需的远为冗长的C接口相比较，很有启发意义。

- **系统V IPC机制：**SunOS UNIX提供了一套共享内存、同步和消息传递机制，即俗话所说的“系统V IPC”[29]。这些机制所提供的大多数功能已包含在较新的SunOS UNIX机制中（比如，对应地，mmap、线程同步[31]、和STREAM管道原语）。但是，特定类型的应用（比如数据库引擎）可能会从系统V IPC机制中获益（比如消息队列的对等天性、高效的信号量多操作原子语义，以及系统V IPC机制在一系列UNIX OS 平台上的广泛可用性）。然而，要正确地理解和使用系统V IPC机制（特别是信号量）有一点挑战性，因为这些接口非常通用，而且直到最近，有关它们的行为特性的文档都非常地少。

  ACE系统V IPC包装接口将开发者与无数多余的细节屏蔽开来。例如，对于利用标准wait和signal信号量的应用来说，ACE OO包装版本的系统V IPC信号量使用起来更为直观和简单；如下面的来自典型的生产者/消费者例子的代码片段所示：

```
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 ();
}
}
```

把这个简洁的OO包装接口与直接使用系统V信号量所需的远为冗长的C接口相比较，很有启发意义。

1.3.2 反应堆（Reactor）：事件多路分离和事件处理器分派

通信软件对许多不同种类的事件（比如基于定时器、基于I/O、基于信号和基于同步的事件）进行多路分离和处理。例如，WWW服务器通常在内部使用事件循环、监控一个周知的Internet端口（通常是端口80）。该端口与一个应用特有的处理器（handler）相关联，此处理器侦听客户在端口80上的连接。当客户连接时，WWW服务器接受连接，并创建一个事件处理器为此HTTP请求服务。例如，如果Netscape浏览器发送一个GET请求，WWW服务器将返回所请求的内容给浏览器。

为统一并自动化事件驱动的处理活动，ACE提供了一个事件多路分离和事件处理器分派构架，称为ACE_Reactor[10]。Reactor将UNIX和Windows NT事件多路分离机制（比如select和poll）的功能封装在一个可移植和扩展的OO包装中[10]。这些OS事件多路分离系统调用检测在一或更多I/O句柄上同时发生的不同类型的输入和输出事件。

为改善应用的可移植性，不管使用了何种事件多路分离机制，ACE_Reactor都提供同样的接口。另外，ACE_Reactor还封装了在多线程事件处理环境中正确而高效地执行回调方式的分派所必需的互斥机

制。

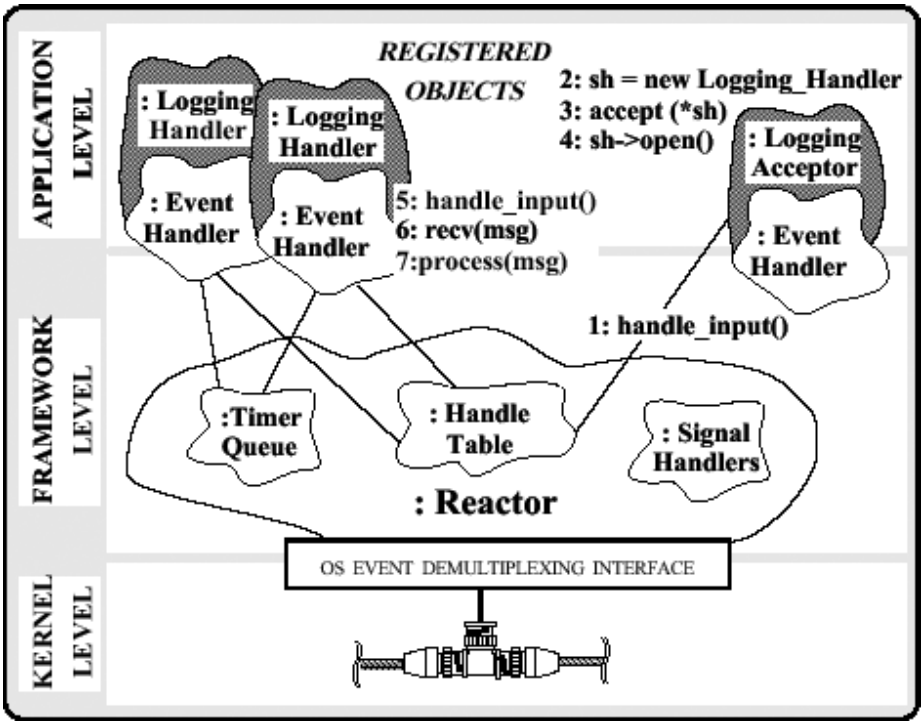

图1-5 ACE_Reactor类属组件

ACE_Reactor中的对象的结构在图1-5中演示。这些对象负责（1）事件（比如定时器驱动的呼出队列生成的*临时事件*、通信端口上收到的*I/O事件*，以及*信号事件*）的多路分离和（2）分派预登记事件处理器的适当方法，以处理这些事件。如图1-5所示，所有事件处理器对象都派生自ACE_Event_Handler抽象基类。该类指定一个被用于ACE_Reactor接口，以针对特定事件的到达、分派特定的应用特有的方法。

ACE_Reactor使用在Event Handler中声明的虚方法，以集成基于I/O句柄、基于定时器，和基于信号的事件的多路分离。基于I/O句柄的事件经由handle_input、handle_output和handle_exceptions方法分派；基于定时器的事件经由handle_timeout方法分派；而基于信号的事件经由handle_signal分派。

ACE_Event_Handler的子类（比如1.4.1描述的Logging_Handler）可以通过定义另外的方法和数据成员来扩展基类的接口。此外，ACE_Event_Handler接口中的虚方法可以有选择地被子类重载，以实现应用特有的功能。例如，在1.4.2介绍的PBX监控服务器中，应用特有的子类定义了Event Handler对象，通过继承和/或实例化1.3.1描述的SOCK SAP或TLI SAP传输接口类的对象，与客户进行通信。在ACE_Event_Handler基类中的虚方法被子类定义后，应用可以实例化所得的事件处理器对象。

下面的例子实现一个简单的程序，它使用双向通信信道，在两个进程间持续地前后交换消息。此例演示服务是怎样从ACE_Event_Handler继承的。它还描述ACE_Reactor怎样被用于多路分离和分派基于I/O、基于信号和基于定时器的事件。下面所示的Ping_Pong类从ACE_Event_Handler继承接口，并实现它自己的应用特有的功能：

```
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;
}
```

主程序通过打开适当的通信信道开始运行。接着，程序派生一个子进程，在两个进程中各实例化一个名为callback的Ping_Pong事件处理器对象，为基于I/O、基于信号，和基于定时器的事件而将callback对象登记到ACE_Reactor的一个实例上，然后进入事件循环。如下所示：

```
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;
}
```

基于定时器和基于信号的事件的callback事件处理器存储在ACE_Reactor中的相应的表中。同样地，当调用register_handler方法登记基于I/O的事件处理器时，ACE_Reactor在一个内部表中存储适当的句柄。当应用随后通过调用ACE_Reactor::handle_events方法执行它的主事件循环时，该句柄被作为参数传递给底层的OS I/O多路分离系统调用（例如，select或poll）。

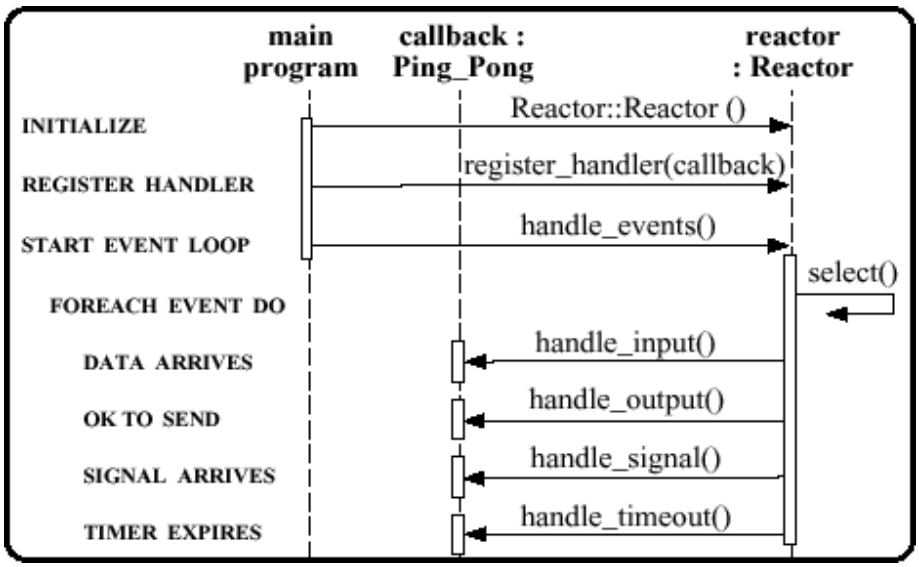

图1-6 ACE Reactor交互图

当与预登记的事件处理器callback对象相关联的输入、输出，和定时器事件在运行时发生时，ACE_Reactor自动检测这些事件并分派适当的事件处理器对象的方法。被分派的callback对象的方法负

责完成应用特有的功能（比如写入消息到通信信道、从通信信道读取消息，或是设置触发程序终止的标志）。这些组件间的协作通过对象交互图在图1-6中描述。

1.3.3 并发：多线程和同步机制

ACE并发类属包含的OO包装（例如，ACE_Mutex、ACE_Condition、ACE_Semaphore和ACE_RW_Mutex）封装了相应的Solaris[31]和POSIX Pthreads[32]多线程和同步机制。这些包装使在类中作为成员域出现的同步对象的初始化过程得以自动化，并且还简化了线程和同步机制的常见的使用模式。例如，下面的代码演示了怎样将SunOS mutex_t和cond_t同步机制的ACE封装用于典型的共享资源管理类：

```
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的构造器是怎样将ACE_Mutex对象lock_与Condition对象绑定在一起的。与底层的SunOS cond_t cond_wait接口相比较，这样的方式简化了ACE_Condition::wait调用接口。

尽管ACE_Mutex包装为同步多线程控制提供了一种相对优雅的方法，它们仍是潜在地易错的，因为开发者有可能会忘记调用release方法（或是由于程序员的疏忽，或是由于C++异常的发生）。为改善应用的健壮性，ACE同步机制有效地利用了C++类构造器和析构器语义。为确保ACE_Mutex锁被自动获取和释放，ACE提供了名为ACE_Guard的助手类，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类的对象定义一个代码块，在块开始时获取ACE_Mutex，而在块结束时自动释放。

注意ACE_Guard类被定义为模板，由互斥机制参数化。有若干不同类型的互斥语义[33]。每种互斥共有一个通用接口（也就是，acquire/release），但具有不同的序列化和性能属性。ACE支持的两种互斥是*非递归*和*递归锁*。

- **非递归锁：**非递归锁提供互斥的一种高效的形式，它定义一个*临界区*，每一时刻只有单个线程可在其中执行。它们之所以是非递归的，是因为当前拥有锁的线程在将其释放前不可以再次获取它。否则，就会立即发生死锁。SunOS 5.x通过它的mutex_t、rwlock_t，和sema_t类型（POSIX Pthreads不提供后两种同步机制）为非递归锁提供支持。ASX构架提供Mutex、RW_Mutex，和Semaphore包装，以分别封装这些语义。
- **递归锁：**另外一方面，递归锁允许acquire方法嵌套调用，只要当前拥有该锁的线程就是试图重新获取它的线程。递归锁对于回调驱动的事件分派构架（比如1.3.2描述的反应堆）特别有用，在其中构架的事件循环执行对预登记的用户定义的对象的回调。因为随后用户定义的对象可能经由它的方法入口重入分派构架，必须使用递归锁以防止在回调过程中构架持有的锁发生死锁。
  下面的C++模板类为Solaris线程和POSIX Pthreads（它们自己不提供递归锁语义）的同步语义实现了递归锁语义：

```
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_;
};
```

注意这个类的接口与ACE中可用的其他锁定机制一致[22]。

下面的代码演示怎样将ACE_Guard和ACE_Recursive_Thread_Mutex用于回调机制中：

```
int  Callback::dispatch (const Event_Handler *eh,  Event *event)
{
// Constructor acquires the lock on entry.
ACE_Guard<ACE_Recursive_Thread_Mutex<ACE_Mutex> > mon (this->lock_);

eh->handle_event (event);
// Destructor of Guard releases the lock on exit.
}
```

该代码确保Event_Handler对象的登记作为临界区执行。该例子还演示了C++习语的使用[34]，在其中，当ACE_Guard对象创建时，类的构造器自动在同步对象上获取锁。同样地，当mon对象出了作用域，类的

析构器自动解锁对象。而且，mon的析构器将被自动调用以释放互斥锁，而不管if/else语句的哪一分支从方法返回。此外，如果在register_handler方法的处理过程中发生了C++异常，锁也将会被自动释放。ACE_Recursive_Thread_Mutex用于确保dispatch方法分派的应用特有的handle_event回调不会导致死锁，如果它们重入Callback对象的话。

ACE还提供一个ACE_Thread_Manager类，其中包括一组用于管理相互协作、以实现集体行为的线程组的机制。例如，ACE_Thread_Manager类提供的机制（比如suspend_all和resume_all）允许任意数目的参与线程被原子地挂起和恢复。

1.3.4 服务配置器（Service Configurator）：显式动态链接机制

*静态链接*是这样一种技术：在编译时和/或静态链接时将所有对象文件绑定在一起，以组成完整的可执行程序。相反，*动态链接*使得对象文件可以在初始调用程序或迟后在运行时的任意时刻被加入地址空间，和/或被从中删除。SunOS 4.x和5.x同时支持*隐式的*和*显式的*动态链接：

- *隐式动态链接*被用于实现共享对象文件，也称为共享库[35]。共享对象文件可减少主要和辅助存储的使用，因为只有一份共享对象的拷贝存在于内存中和磁盘上，而不管执行该代码的进程的数量是多少。而且，特定的地址解析和重定位操作可以延迟至动态链接的函数被初次引用时。这样的“懒惰计算”方案将进程启动时的链接编辑开销降到了最小。
- *显式动态链接*提供的接口允许应用获取、利用，和/或移除在共享对象文件中定义的符号的运行时地址绑定[36]。显式动态链接机制显著地增强了通信软件的功能和灵活性，因为服务可以在运行时插入和/或删除，而*不用*终止和/或重启整个应用。SunOS 5.x通过dlopen/dlsym/dlcose例程支持显式动态链接，而Win32通过LoadLibrary/GetProcAddress调用支持这一特性。

ACE提供Service Configurator类属来在一组类和继承层次中封装SunOS的显式链接机制。Service Configurator有效地利用了其他ACE组件来扩展传统的看守配置和控制构架的功能[20]（比如listen[3]、inetd[2]，和Windws NT Service Control Manager[37]），这些功能为下列行为提供自动的支持：（1）并发、多服务通信软件的静态和动态配置，（2）为I/O活动监控通信端口组，以及（3）分派在所监控端口上接收到的消息给适当的应用特有的服务。这一部分的余下部分讨论Service Configurator类属的主要组件。

1.3.4.1 ACE_Service_Object继承层次

Service Configurator中的主要配置单元是*服务*（service）。服务可以是简单的（比如返回一天中的当前时间）或高度复杂的（比如PBX事件话务的分布式、实时路由器[6,38]）。为了给定义、配置和使用通信软件提供一致的环境，所有的应用服务都派生自ACE_Service_Object继承层次（在图1-7(1)中演示）。

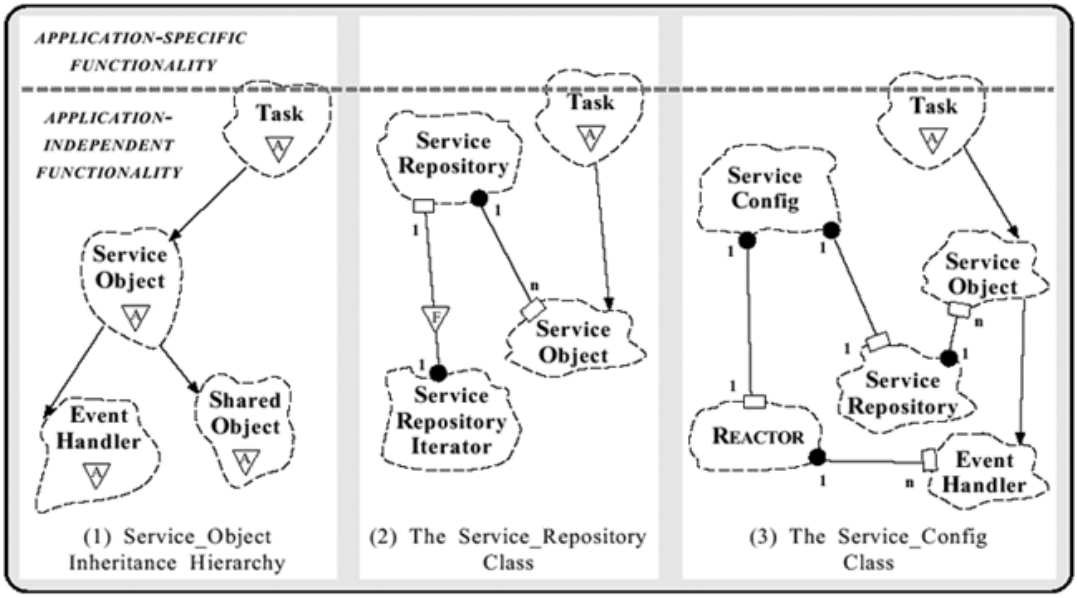

图1-7 Service Configurator类属的组件关系

ACE_Service_Object类是一个有着继承关系的多层类型层次的焦点。在此类型层次中的抽象类提供的标准接口可以有选择地被应用特有的子类实现，以访问特定的、应用无关的Service Configurator机制。这些机制提供透明的动态链接、事件处理器登记、事件多路分离和服务分派。通过使处理器对象的应用特有部分与底层的应用无关的Service Configurator机制去耦合，显著地减少了在运行中的应用中插入和移除服务所必需的工作。

ACE_Service_Object继承层次由ACE_Event_Handler和ACE_Shared_Object抽象基类组成。ACE_Event_handler类已在上面的1.3.2中描述。其他类的行为在下面概述。

- ACE_Shared_Object**抽象基类：**该基类指定用于将服务处理器对象动态链接进应用的地址空间的接口。ACE_Shared_Objecti抽象基类输出三个抽象方法：init、fini，和info。这些方法在Service Configurator提供的应用无关的可复用组件和利用这些组件的应用特有功能之间建立了协定。通过使用抽象方法，Service Configurator确保服务处理器实现遵从它提供特定的配置相关信息的义务。这些信息随后被Service Configurator用于在运行时自动链接、初始化、标识服务，以及解除服务的链接。

  ACE_Shared_Object基类独立于ACE_Event_Handler类而定义，以清晰地区分它们的两组不相关的事务。例如，某些应用（比如编译器或文本编辑器）可能会从动态链接中获益，尽管它们可能不需要通信端口事件多路分离。相反，其他应用（比如ftp服务器）可能需要事件多路分离，但可能不需要动态链接。通过将这些接口分离成两个基类，应用就能够选择服务配置器的一个子集，而不会带来不必要的存储开销。

- ACE_Service_Object**抽象派生类：**通常，安装和管理复杂的分布式系统需要有动态链接、事件多路分离和服务分派的支持，以使应用服务动态配置和重配置自动化。因而，Service Configurator定义了ACE_Service_Object类，将ACE_Event_Handler和ACE_Shared_Object抽象基类的接口结合在一起。所得到的抽象派生类提供的接口可被开发者用作实现服务、并将其配置进服务配置器的基础。

  在开发过程中，ACE_Service_Object的应用特有的子类必须实现由ACE_Service_Object类接口指定的suspend和resume抽象方法。这些方法被Service Configurator自动调用，以

响应外部的事件。应用开发者可以控制对象为挂起服务所采取的动作，而无须完全移除它或解除它的链接；唤醒先前挂起的服务也是一样。

此外，应用特有子类还必须实现ACE_Service_Object子类所继承（不是定义）的四个抽象方法（init、fini、info，和get_handle）。init方法用作服务处理器在运行时初始化过程中的入口。当ACE_Service_Object的实例被动态链接时，该方法负责执行应用特有的初始化。同样地，当ACE_Service_Object在运行时被解除链接并从应用中移除时，Service Configurator自动调用fini方法。该方法通常执行终止操作，由其释放动态分配的资源（比如内存、I/O句柄或同步锁）。info方法格式化用户可读的一个字符串，简洁地报告服务访问信息及为服务功能建立文档。客户可以查询应用以获取此信息，并将它用于与应用中运行的某个特定的服务联系。最后，get_handle方法被Reactor用于从服务处理器对象中提取底层的I/O句柄。此I/O句柄标识一个可被用于从客户那里接受连接或接收数据的传输端点。

- **应用特有的具体派生子类：**Service Object是一个抽象类，因为它的接口包含有继承自Event Handler和Shared Object抽象基类的抽象方法。因而，开发者必须提供具体的子类（1）定义上面描述的六个抽象方法，以及（2）实现必需的应用特有的功能。为完成后一任务，子类通常定义由Service Object接口输出的某些特定的虚方法。例如，常常实现handle_input方法，用以从客户那里接受连接或数据。

  图1-7（1）中描述的ACE_Acceptor类是应用无关的子类的一个例子；它是分布式日志工具的一部分，接受连接请求。该类将在1.4.1介绍的例子中作进一步描述。

1.3.4.2 ACE_Service_Repository类

Service Configurator类属同时支持单服务和多服务通信软件的配置。因而，为简化运行时管理，常常有必要个别和/或共同地控制及协调由应用的正在活动的服务组成的各个ACE_Service_Object。ACE_Service_Repository是用于协调本地和远地查询的对象管理器，这些查询涉及由基于Service Configurator的应用提供的服务。在对象管理器中的搜索结构将服务名（以ASCII字符串表示）与ACE_Service_Object（以对象代码表示）绑定在一起。一个服务名唯一地标识在仓库中（repository）存储的一个ACE_Service_Object的实例。

ACE_Service_Repository中的每一条目都含有一个指向应用特有派生类的ACE_Service_Object部分的指针（如图1-7(2)所示）。这使得Service Configurator能够静态或动态地对应用进行ACE_Service_Object的装载、启用、挂起、恢复，或卸载。对于动态链接的ACE_Service_Object，仓库还维护了底层共享对象文件的句柄。当ACE_Service_Object所提供的服务不再被需要时，这个句柄用于从运行中的应用解除其链接并卸载之。

与ACE_Service_Repository一起，还提供了一个迭代器（iterator）类。该类用于访问仓库中的每一个ACE_Service_Object，而不会不恰当地危及数据封装。

1.3.4.3 ACE_Service_Config类

ACE_Service_Config类是Service Configurator构架中的统摄组件。如图1-7(3)所示，该类集成了其他的Service Configurator构架组件（比如ACE_Service_Repository和ACE_Reactor），以使并发、多服务通信软件的静态和/或动态配置自动化。

ACE_Service_Config类使用配置文件（称为svc.conf）来指导它的配置和重配置活动。每一应用都可与一个独立的svc.conf配置文件相关联。同样地，一组应用也可以被单个的svc.conf文件描述。图1-8使用扩展的Backus/Naur格式（EBNF）描述了svc.conf文件中的主要语法成分。文件中的每个*服务配置条目*以*服务配置指令*起头，它指定要执行的配置活动。表1-1总结了有效的服务配置指令。

```
<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
```

图1-8 服务配置条目的EBNF格式

指令	描述
dynamic	动态链接和启用服务
static	启用静态链接的服务
remove	完全地移除服务
suspend	挂起服务，而不移除它
resume	恢复先前挂起的服务

表1-1 服务配置指令

对于每个动态链接的服务，相应的服务配置条目含有指示共享对象文件位置的属性，以及需用于在运行时初始化服务的参数。通过将服务属性和初始化参数统一进单一配置文件，显著地简化了应用中服

务的安装和管理。svc.conf文件有助于使应用的结构与它的服务的行为去耦合。基于在svc.conf文件中指定的应用特有的属性和参数，这样的去耦合也允许对构架提供的机制进行“懒惰的”配置和重配置。

图1-9描述的状态转移图演示了Service Configurator类属中的一些方法，这些方法被调用，以响应在服务配置、执行和重配置的过程中发生的事件。例如，当CONFIGURE和RECONFIGURE事件发生时，ACE_Service_Config类的process_directives方法被调用，以查询svc.conf文件。在应用的新实例被初始配置时这个文件第一次被查询。无论何时接收到预先指定的外部事件（比如UNIX SIGHUP信号或来自socket的通知）、触发了应用的重配置，这个文件都会被再次查询。

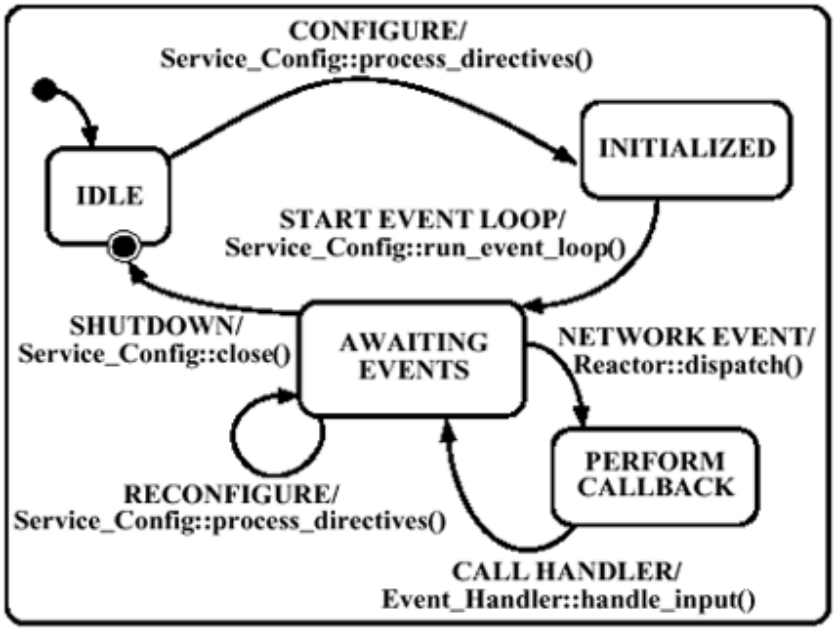

图1-9 服务配置、执行和重配置的状态转移图

1.3.5 流：分层服务的集成

流类属是自适配通信环境的主要焦点。该类属含有自适配服务执行体（ASX）构架，它集成了较低级的OO包装组件（像IPC SAP）和较高级的类属（像Reactor和Service Configurator）。ASX构架有助于简化分层地集成在一起的通信软件的开发，特别是用户级通信协议栈和网络服务器。ASX构架设计用于改善应用特有的服务、以及这些服务所依赖的底层OS并发、IPC、显式动态链接和多路分离机制的模块性、可扩展性、复用性和可移植性。

ASX构架为通信软件的开发者提供下面两种好处：

1. *它嵌入、封装，并实现了通常用于开发通信软件的一些关键设计模式。*通过确定大型系统开发中的基本挑战，设计模式有助于提高软件的质量。这些挑战包括开发者之间的体系结构知识的交流；适应新的设计范式或体系样式；解决非功能性的压力，比如复用性、可移植性和可扩展性；以及避开那些通常只能通过经验来学习的开发陷阱和缺陷。

2. *它严格地区分了关键的开发事务*。ACE将通信软件的开发划分为两种不同的范畴：（1）*不依赖于应用的事务*，它们对于大多数或所有的通信软件来说都是通用的（比如端口监控；消息缓冲、排队和多路分理；服务分派；本地/远地进程间通信；并发控制；以及应用配置、安装和运行时服务管理），以及（2）*应用特有的事务*，它们依赖于特定的应用。通过复用ACE提供的OO包装和构架，开发者从花费时间重新发明一些经常重复出现的任务的方案中被解放出来。于是，这使得他们可以专注于那些组成特定应用的、关键的更高级功能需求和设计事务。

1.3.5.1 主要的ASX特性

通过使应用特有的处理策略与下列配置相关的开发活动和机制去耦合，ASX构架增强了通信软件的灵活性：

- **与每个应用进程相关联的服务的类型和数目：**ASX构架允许应用将一或多个服务合并进单一的管理单元。这样的配置通信软件的多服务方法有助于（1）通过自动完成通用的服务初始化活动，简化开发并复用代码，（2）通过“按需”生成服务处理器，降低OS资源的消耗（比如进程表槽），（3）无须修改已有的源代码或终止正在执行的分派进程（比如inetd超级服务器）就能够更新应用服务，以及（4）通过一组一致的配置管理操作，统一网络服务的管理。
- **服务被配置进应用的时间点：**ASX构架有效地对Service Configurator构架（在1.3.4中描述）进行利用，以提供一个可扩展的、面向对象的接口，使显式动态链接的OS机制的使用得以自动化。通过允许内部服务在应用开始执行时或运行时被配置，动态链接增强了通信软件的可扩展性。该特性使得应用的服务可被动态配置，而无需修改、重编译、重链接，或重启活动的服务。在ASX构架中，选择静态或动态配置可针对每个服务单独进行。而且，选择可延迟至应用开始执行的时候。
- **执行代理的类型：**在ASX构架中，服务可在运行时通过若干不同类型的进程和线程执行代理来执行。通过使服务功能与用于调用服务的执行代理去耦合，ASX构架扩展了开发者可用的应用并发配置方式的选择范围。

  高效的应用并发配置常常依赖于特定的服务需求和平台特性。例如，基于进程的配置对于实现长持续时间的服务（比如Internet ftp和telnet）来说也许是适当的，这样的配置使它们的安全机制基于进程所有权。在这种情况下，每个服务（或服务的每个活动实例）可被映射到不同的进程，并在多处理器平台上并行执行。但是，在其他环境中采用另外的配置可能更为合适。例如，在分离的线程中实现相互协作的服务（比如在分布式数据库引擎的终端系统中的服务）常常更加简单和高效，因为它们经常性地访问公用数据结构。在该方法中，每个服务可以在同一进程的不同线程中执行，以降低调度、上下文切换和同步的开销[6]。

- **层次相关的服务被结合进应用的顺序：**复杂服务可使用一系列互连的独立服务对象组成，它们通过传递消息进行通信。这些对象可以通过本质上任意的配置结合在一起，以满足应用要求和增强组件复用。
- **基于I/O句柄和基于定时器事件的多路分离机制：**这些机制用于将到来的连接请求和数据分派给预登记的应用特有的处理器。通过一个可扩展的和类型安全的面向对象接口，ASX构架使用Reactor类属来集成基于I/O句柄、基于定时器，和基于信号的事件的多路分离。
- **底层IPC机制：**应用服务可以使用1.3.1描述的IPC SAP机制来与本地或远地终端系统上的通信实体交换数据。不像弱类型的、“基于句柄”的socket和TLI接口，IPC SAP包装使得应用能够通过类型安全的、可移植的接口访问底层OS IPC机制。

ASX构架合并了来自若干模块化的通信构架的概念，其中包括系统V STREAMS[39]、x-kernel[40]和来自面向对象操作系统Choices的Conduit构架（对这些和其他通信构架的考察见[42]）。这些构架全都支持通过将“积木块”协议和服务组件互连来灵活地配置通信子系统。总之，通过使处理功能与周边构架的基础构造去耦合，这些构架鼓励了标准的与通信相关的组件的开发（比如消息管理器、基于定时器的事件分派器、多路分离器[40]和各种协议功能[43]）。如下所述，ASX构架包含的一些额外特性可进一步使处理功能与底层的处理体系结构去耦合。

不像STREAMS，被配置进ASX构架的应用服务在用户空间、而不是内核空间中执行。在用户空间、而不是OS内核中开发通用通信软件有着若干优点：

- **对一般OS特性的访问：**在用户空间中运行的应用可以访问全面的OS机制（比如动态链接、内存映射文件、多线程、大虚拟地址空间、进程间通信机制、文件系统和数据库）。相反，驻留内核的组件常常局限于一组有限的内核特有的机制。尽管有一些习语可以克服其中的一些局限（例如，维护一个用户级看守、代表驻留内存的组件完成文件I/O），这些工作方法往往不那么优雅和可移植。
- **增强的开发环境：**较高级编程工具（比如符号调试器）可用于开发用户空间中的应用。相反，由于在内核编程环境中，调试工具的原始和微妙的时序干扰，在OS内核中开发网络服务是一项复杂而艰巨的任务[44]。在这样受限的环境中期待开发者有效地编程是很冒险的。
- **增强的系统健壮性：**在用户级进程或线程中产生的异常情况（比如废弃的NULL指针、除0错误，等等）只会影响出错进程或线程。但是，OS内核中的异常情况可能导致整个操作系统的“恐慌”和崩溃。而且，在每次崩溃后重启OS很快会变得冗长乏味。
- **可移植性：**在不同OS平台间（即使是同一OS平台的不同变种）移植内核级驱动程序通常比移植用户级应用组件要承担更多的复杂性。SunOS 5.x DDI/DKI API意图减少UNIX平台上的一些移植性问题，但那并未解决在其他平台上（比如OS/2、Windows NT、VMS和Novell Netware）的应用的移植性问题。

在OS内核中实现分布式服务的主要原因是提高性能。例如，一个通信协议栈的驻留内核的实现常常有助于减少调度、上下文切换和跨越保护域边界的开销，并且因为使用了“线性的”（而非分页的）内存，还可以表现出更可预测的响应时间。但是，对于许多通信软件应用来说，用户空间开发所带来的更多的灵活性、简单性、健壮性和可移植性是关键性的需求，可以弥补潜在的性能下降。

1.3.5.2 Stream类属组件

Stream类属中的组件负责协调一或多个ACE_Stream对象的配置和在运行时的执行。 ACE_Stream是用户应用与之协作的对象，用以配置和执行ASX构架中的应用特有的服务。如图1-10所示，ACE_Stream包含有一系列互连的ACE_Module对象，它们可被（1）开发者在安装时或（2）应用在运行时链接在一起。ACE_Module对象用于将应用的体系结构分解成一系列互连的、功能独立的层次。通常每一层实现一簇相关的服务功能（比如端对端传输服务或表示层格式化服务）。每个ACE_Module包含的一对ACE_Task对象将该层划分为读端和写端处理功能。

OO语言特性（比如类、继承、动态绑定和参数化类型）使开发者无须修改不依赖于应用的ASX构架组件，就能够将应用特有的功能合并进流中。例如，将服务层增加进流涉及（1）从缺省ACE_Task接口继承并有选择地在子类中覆盖若干虚函数，以提供应用特有的功能，（2）分配一个新的ACE_Module，其中包含有应用特有的ACE_Task子类的两个实例（一个用于读端，一个用于写端），以及（3）将

ACE_Module插入到流中。经由消息传递接口，邻近的相互连接的ACE_Task中的服务通过交换类型化的消息进行协作。

为避免再次发明习惯用语，流类属中的许多类名与系统V STREAMS构架中可用的类似组件相对应[39]。但是，在这两种构架中，用于支持可扩展性和并发的技术有着显著的差异。例如，为ASX流类增加应用特有的功能是通过从已有的构架组件继承若干定义良好的接口和实现来进行的。比起系统V STREAMS中所用的函数指针技术，使用继承来增加新功能提供了强得多的类型安全性。ASX流类还采用了不同的并发控制方案来减少死锁的可能性，并简化流中的Task之间的流控制。ASX流类对系统V STREAMS中所用的基于协同程序的、“无重量的”服务处理机制[45]进行了完全的重新设计和实现。这些ASX的变动意图在共享内存的多处理器平台上更为有效地利用多PE。

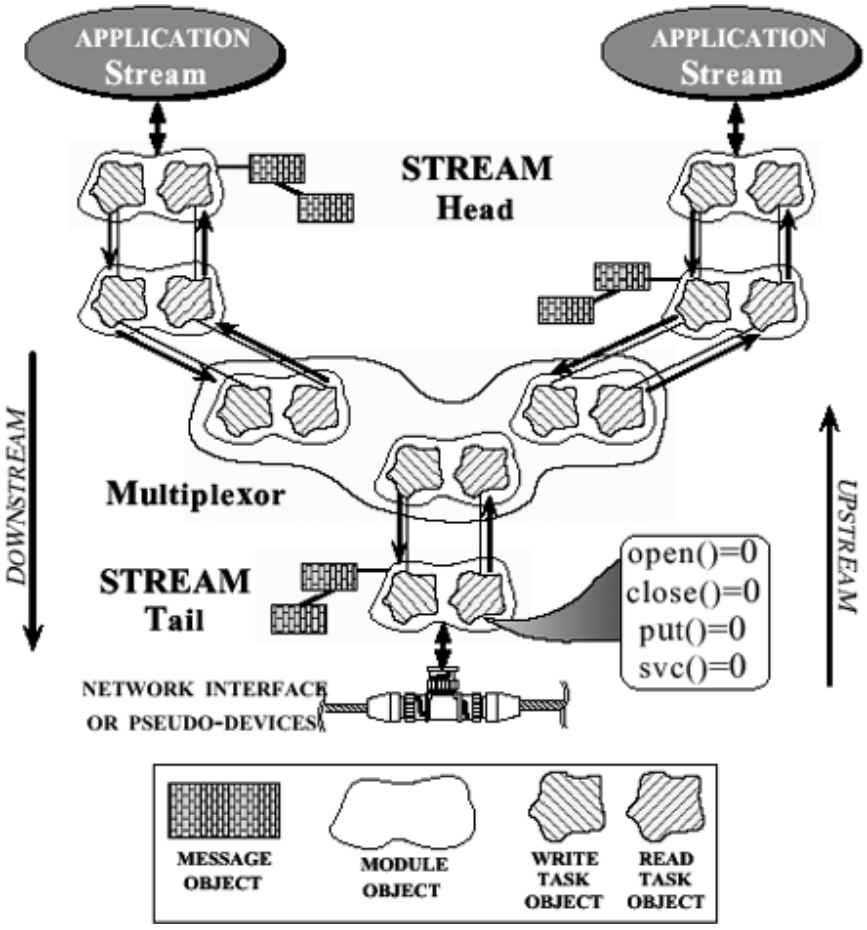

图1-10 ASX构架中的组件

余下部分详细讨论流类属的主要组件（例如，ACE_Stream类、ACE_Module类、ACE_Task类）：

- **ACE_Stream类：**ACE_Stream类定义流的应用接口。ACE_Stream对象包含有一个由一或多个层次相关的服务组成的栈，为应用提供双向的get/put风格的接口，用以通过组成特定流的互连Module发送和接收数据及控制消息。ACE_Stream还实现了一个push/pop风格的接口，允许应用通过插入和移除下面描述的ACE_Module类的对象、在运行时对流进行配置。
- **ACE_Module类：**ACE_Module类定义独立的、应用特有的功能层。流通过互连一系列ACE_Module对象构成。流中的ACE_Module对象有着松散的耦合，并通过传递类型化的消息来与邻近的

ACE_Module对象协作。每个ACE_Module对象包含有一对指针，指向应用特有的ACE_Task类的子类的对象。ACE_Task类在下面简略描述。

如图1-10所示，当流被打开时，两个缺省的ACE_Module对象（ACE_Stream_Head和ACE_Stream_Tail）自动被安装。这两个ACE_Module解释预定义的ASX构架控制消息和数据消息，这些消息在运行时在流中流动。ACE_Stream_Head类在应用和流之间提供消息缓冲接口。ACE_Stream_Tail类通常将来自网络或伪设备的消息转换为规范的内部消息格式，可被流中更高层的组件进一步处理。同样地，对于外发的消息，它将其从内部格式转换为网络消息。

- **ACE_Task抽象类：**ACE_Task类是ACE中用于创建用户定义的、处理应用消息的*主动对象*（Active Object）[11]和*被动对象*（Passive Object）的中心机制。ACE的任务可以进行下面的活动：

  - 可被动态链接
  - 可用作I/O操作的多路分离端点
  - 可与多个线程控制相关联（也就是，变成所谓的“主动对象”）
  - 可将消息存储在队列中，以用于后续处理
  - 可执行用户定义的服务

  ACE_Task抽象类定义的接口被派生类继承和实现，以提供应用特有的功能。它之所以是抽象类，是因为它的接口定义了下面描述的抽象方法（open、colse、put和svc）。通过使ACE_Stream类属提供的不依赖于应用的组件与继承并使用这些组件的、应用特有的子类去耦合，把ACE_Task定义为抽象类增强了复用。同样地，使用抽象方法允许编译器确保Task的子类遵从自己提供下面的功能的义务：

  - *初始化和终止方法*：派生自ACE_Task的子类必须实现open和close方法，执行应用特有的ACE_Task初始化和终止活动。这些活动通常分配和释放像连接控制块、I/O句柄，及同步锁这样的资源。

    ACE_Task可与ACE_Module一起或分开定义和使用。当与ACE_Module一起使用时，它们被成对地存储：一个ACE_Task子类操作读端的、自下而上发送给它的ACE_Module层的消息的处理；另一个操作写端的、自上而下发送给它的ACE_Module层的消息的处理。当ACE_Module在流中插入和移除时，Module的写端和读端的ACE_Task子类的open和close方法分别被ASX构架自动调用。

  - *应用特有的处理方法*：除了open和close，ACE_Task的子类还必须定义put和svc方法。这些方法在消息上执行应用特有的处理功能。例如，当消息到达流的头或尾时，作为调用流中的每一ACE_Task的put和/或svc方法的结果，它们被“护送”着通过一系列互连的ACE_Task。

    当在流中某一层的ACE_Task传递消息给另一层中相邻的ACE_Task时，put方法被调用。put方法相对于它的调用者*同步地*运行，也就是，它从最初调用其put方法的Task那里借用线程控制。该线程控制通常源自“上游”应用进程，“下游”处理I/O设备中断的进程池[40]，或流内部的事件分派机制（比

如面向连接的传输协议ACE_Module中的用于触发重发的定时器驱动呼出队列）。

如果ACE_Task作为*被动对象*执行（也就是，它总是从调用者那里借用线程控制），ACE_Task::put方法就是ACE_Task的入口，并用作ACE_Task在其中执行的上下文。相反，如果ACE_Task作为*主动对象*执行，ACE_Task::svc方法就用于相对于其他ACE_Task*异步地*执行应用特有的处理。不像put，svc方法不会被相邻的ACE_Task直接调用，而是被与它的ACE_Task相关联的一个单独的线程调用。该线程为ACE_Task的svc方法提供了执行上下文和线程控制。svc方法运行一个事件循环，持续地等待消息到达ACE_Task的ACE_Message_Queue（见下面的条目）。

在put或svc方法的实现中，消息可经由ACE_Task的put_next方法、被转发给流中相邻的ACE_Task.。put_next调用在相邻层中驻留的下一个ACE_Task的put方法。这个对put的调用可以借用调用者的线程控制，并立即处理消息（也就是，图1-11(1)中所示的同步处理方法）。相反，put方法可以将消息入队，并推迟给在单独的线程控制中执行的svc方法处理（也就是，图1-11(2)中所示的异步处理方法）。如在[6]中所讨论的，选择特定的处理方法对性能和编程的容易度有着显著的影响。

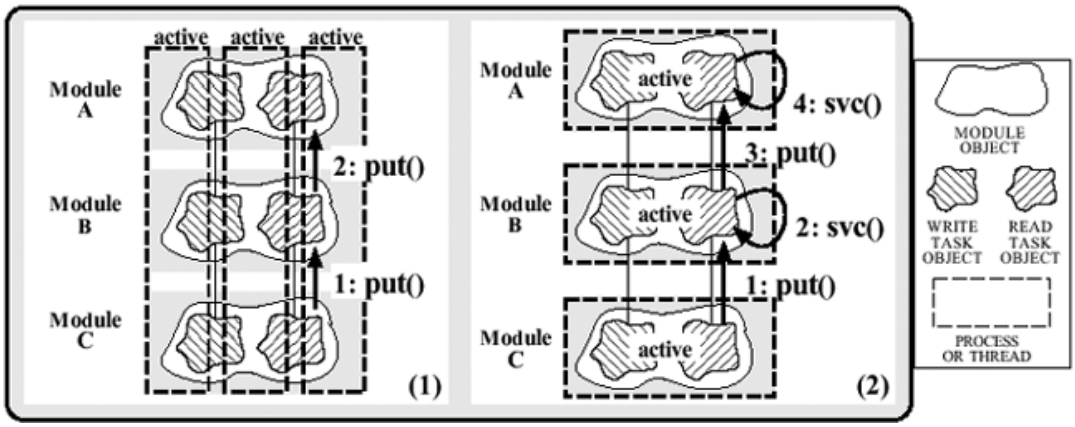

图1-11 调用put和svc方法的可选方法

- *消息排队机制*：除了open、close、put和svc抽象方法接口，每个ACE_Task还具有一个ACE_Message_Queue。ACE_Message_Queue是ACE中的一种标准组件，用于在ACE_Task之间传递信息。而且，当ACE_Task作为主动对象执行时，它的ACE_Message_Queue用于为svc方法中的后续处理缓冲一系列数据消息和控制消息。在消息到达时，svc方法使消息出队，并执行ACE_Task子类的应用特有的处理任务。

  有两种消息可出现在ACE_Message_Queue中：简单的和复合的。简单消息包含单个ACE_Message_Block，而复合消息包含多个链接在一起的ACE_Message_Block。复合消息通常由一个*控制*块和一或多个跟随的*数据*块组成。控制块包含有“簿记”信息（比如目的地址和长度域），而数据块则包含消息的

实际内容。通过传递消息指针而不是拷贝数据，使得在Task间传递消息的开销降到了最低。

ACE_Message_Queue含有一对高低水位标变量，用于在流中相邻的ACE_Module之间实现层到层的流控制。高水位标指示ACE_Message_Queue在它进行流控制前所想要缓存的消息的字节数。低水位标指示表明先前已进行流控制的ACE_Message_Queue不再被认为是“充满”时的“水位”。

1.4 ACE实例

ACE组件目前正在若干研究[46]和商业环境[6, 38, 47]中被用于增强通信软件的配置灵活性、以及可在多种硬件和软件平台上高效而可移植地运行的通信软件的组件的复用。为演示ASX构架是怎样被用于实践的，这一部分考查目前正在使用ACE组件进行开发的两种商业应用的体系结构：分布式日志工具和用于电信交换设备的分布式监控系统。

1.4.1 分布式日志实例

调试分布式软件常常极具挑战性，因为诊断输出出现在不同窗口中和/或不同的远地主机系统上。因而，ACE提供了一个分布式日志工具，以简化调试和运行时跟踪。该工具目前被用于一个商业在线事务处理系统[14]中，为高速网络环境中的群集工作站和多处理器数据库服务器提供日志服务。

如图1-12所示，分布式日志工具允许运行在多个客户主机上的应用发送日志记录给运行在指定服务器主机上的服务器日志看守。这一部分聚焦于日志工具的服务器看守部分的体系结构和配置，它们基于ASX构架提供的Service Configurator和ACE_Reactor类属来实现。分布式日志工具的完整设计和实现在[10]中描述。

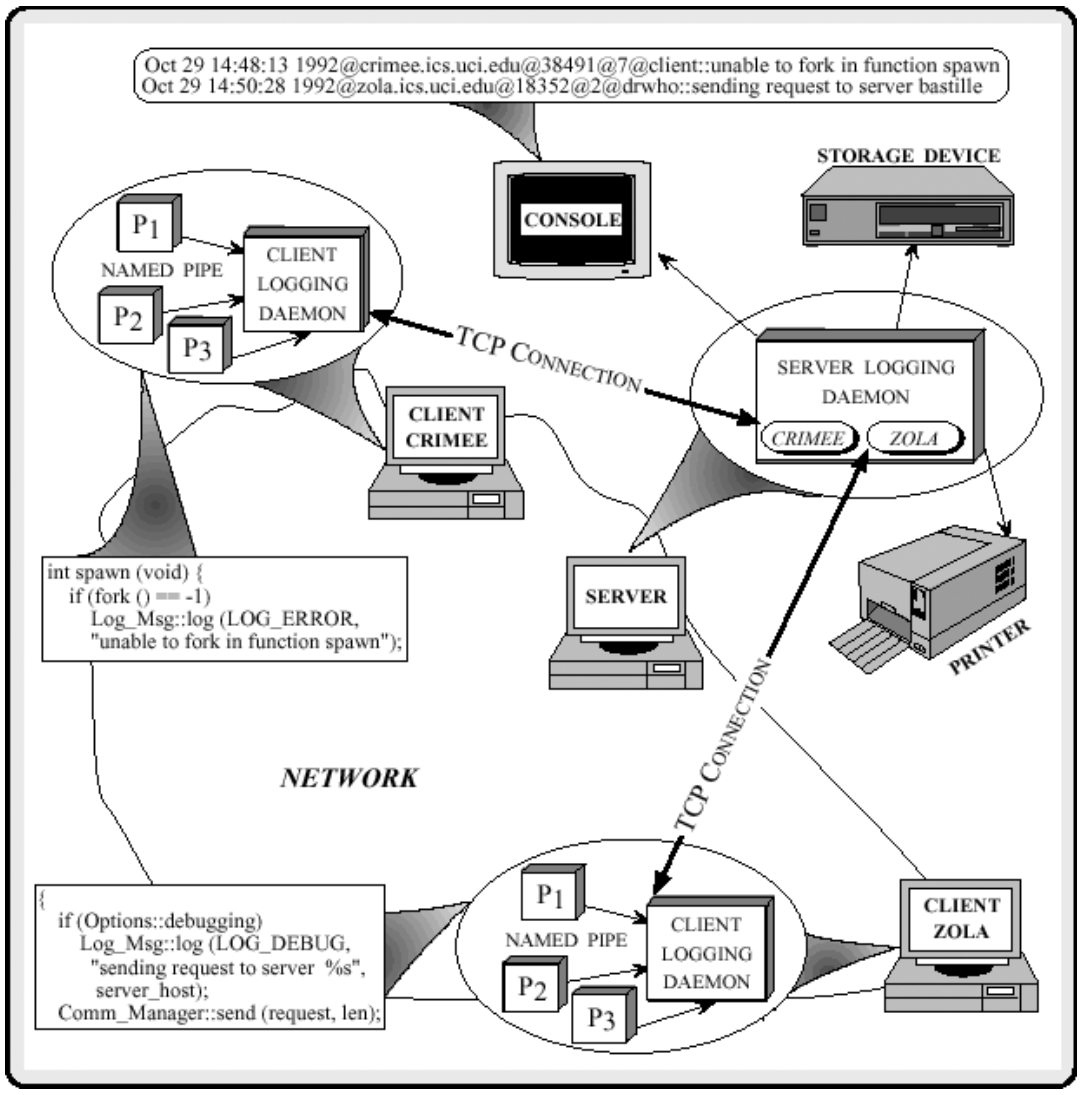

图1-12 分布式日志工具

1.4.1.1 **服务器日志看守组件**

服务器日志看守是一个并发的多服务看守，它同时处理接收自一或多个客户主机的日志记录。服务器日志看守的面向对象的设计被分解为若干模块化组件（在图1-13中显示），由它们执行一些良好定义的任务。应用特有的组件（Logging_Acceptor和Logging Handler）负责处理接收自客户的日志记录。面向连接的组件（Acceptor和Client_Handler）负责接受来自客户的连接请求和数据。最后，不依赖于应用的ASX构架组件（ACE_Reactor、Service Condigurator和IPC SAP类属）负责完成IPC、显式动态链接、事件多路分离、服务分派和并发控制。

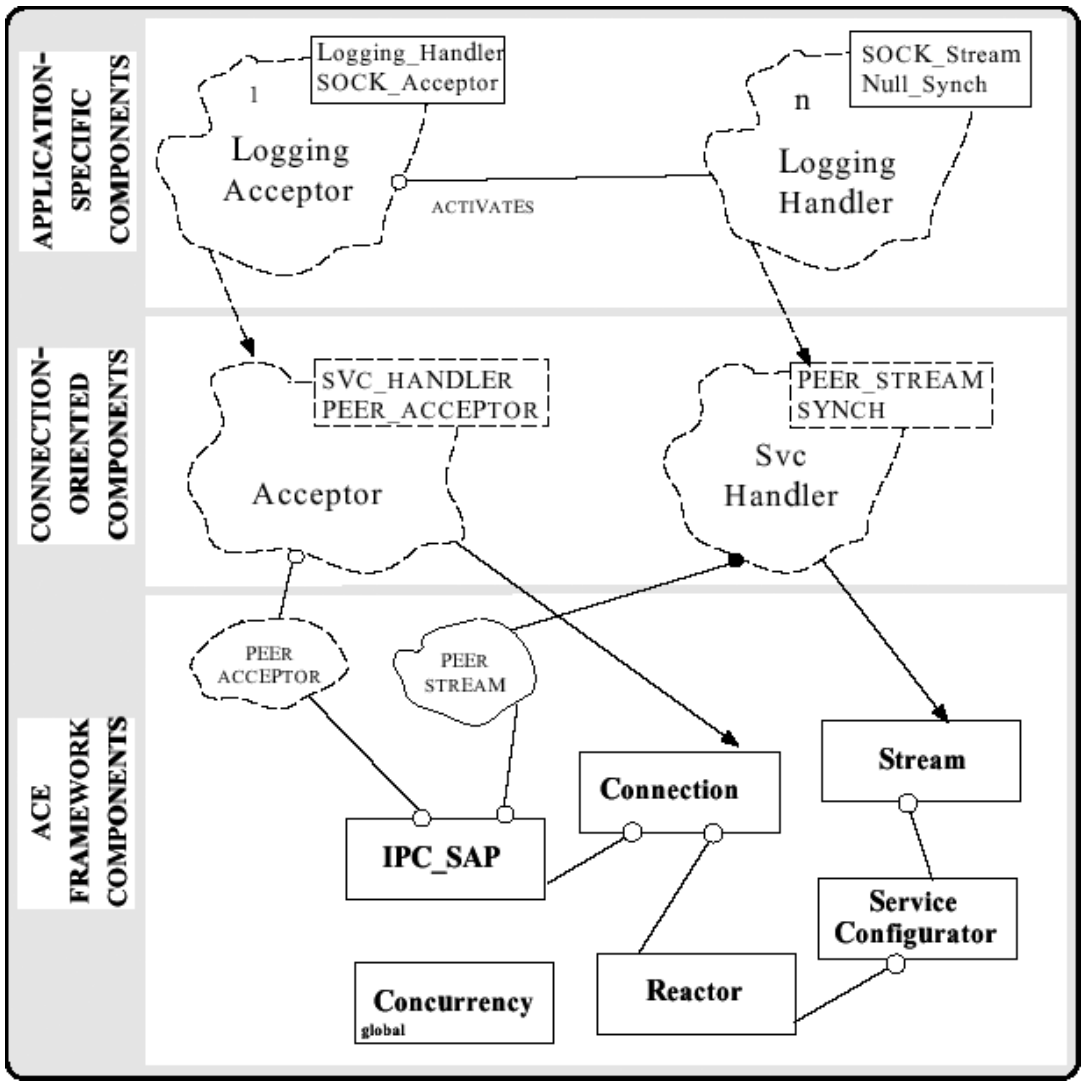

图1-13 服务器日志看守中的类组件

Logging_Handler子类是一种参数化类型，负责处理从客户主机发往服务器日志看守的日志记录。它的通信机制可以通过SOCK SAP或TLI SAP包装来实例化如下：

```
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（间接经由Client_Handler），而不是Service Object，因为它不是被动态链接进服务器日志看守的。

当来自与特定的Logging_Handler相关联的客户主机的日志记录到达时，ACE_Reactor自动分派对象的handle_input方法。该方法格式化记录，并将其显示在一或多个输出设备上（比如图1-12中所示的打印机、持久存储和/或控制台设备）。

Logging_Acceptor子类也是一个参数化类型，它负责接受来自参与日志服务的客户主机的连接请求：

```
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>

{

/* ... */

};
```

因为Logging_Acceptor继承自ACE_Service_Object(间接经由它的ACE_Acceptor基类)，它可以通过服务器日志看守的svc.conf配置文件来在运行时动态地链接进服务器日志看守、并进行相关的操作。同样地，因为Logging_Acceptor间接继承自ACE_Event_Handler接口，当来自客户的连接请求到达时，它的handle_input方法将会被ACE_Reactor自动调用。当连接请求到达时，Logging_Acceptor子类分配一个Logging_Handler对象，并将这个对象登记到ACE_Reactor。

通过使连接建立功能和日志记录接收分离成图1-13所示的两种不同的类层次，显著地增强了分布式日志工具的模块性、可复用性和可配置性。这样的分离允许ACE_Acceptor类被复用于其他类型的面向连接服务。特别地，为提供完全不同的处理功能，只需要重新实现服务的ACE_Client_Handler部分的行为。而且，参数化类型的使用减轻了对某种特定类型的IPC机制的依赖。

1.4.1.2 服务器日志看守配置

ASX构架使用Service Configurator来将日志服务动态或静态配置进服务器日志看守。动态配置的服务可在运行时插入、修改或移除，从而改善了服务的灵活性和可扩展性。下面的svc.conf文件条目用于将日志服务动态地配置进服务器日志看守：

```
dynamic Logger Service_Object *
```

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

<svc-name>符号Logger指定服务名，用于安装和运行时在ACE_Service_Repository中标识相应的Service Object。Service Object * 是位于共享对象文件中的_alloc方法的返回类型，共享对象文件的路径由路径名./Logger.so指定。服务配置器构架定位此共享对象文件，并将其动态链接进日志看守的地址空间。服务路径还指定派生自Service Object的应用特有对象的名字。在此例中，_alloc函数被用于动态分配新的Logging_Acceptor对象。该行剩下的内容（"-p 7001"）表示一组应用特有的配置参数。这些参数作为argc/argv风格的命令行参数传递给服务的init方法。Logging_Acceptor类的init方法将"-p 7001"解释为端口号，服务器日志看守将在其上侦听客户的连接请求。

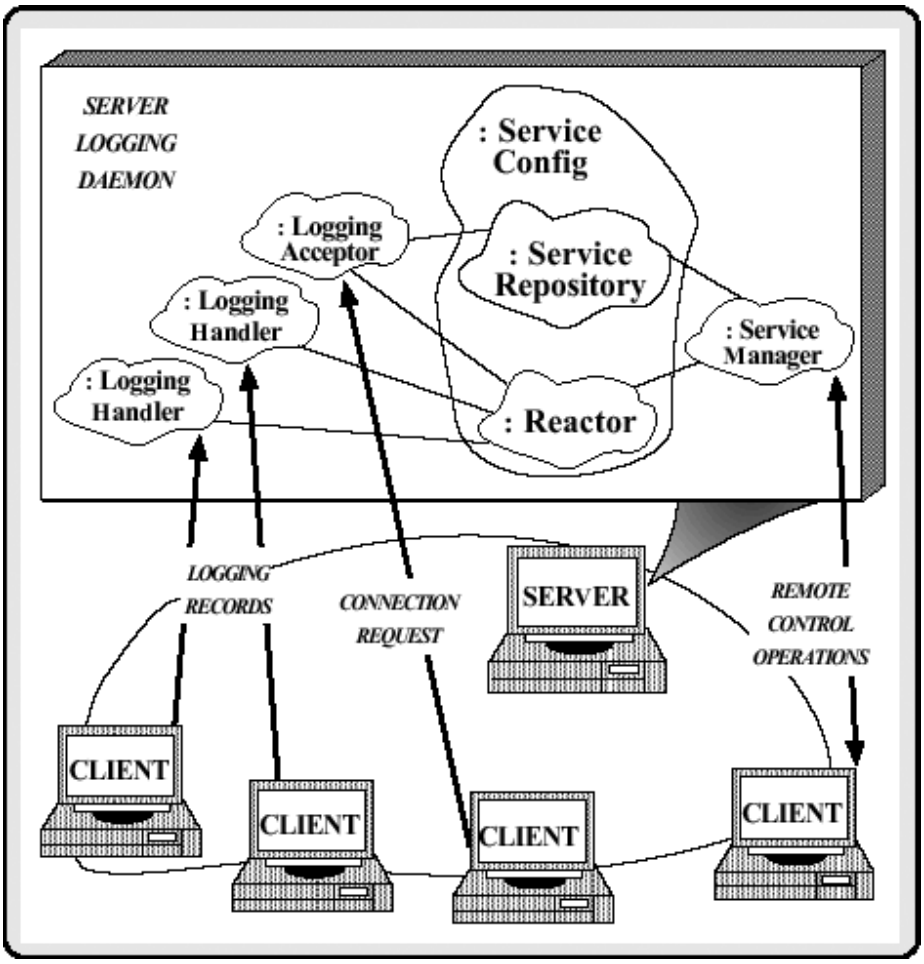

图1-14 分布式日志工具中的ACE组件

当看守一开始执行时，静态配置的服务总是可用的。例如，Service Manager是标准的Service Configurator构架组件，客户可用以获取活动看守服务的列表。下面的svc.conf文件中的条目用于在初始化过程中将Service Manager服务静态地配置进服务器日志看守中：

```
static ACE_Svc_Manager "-p 911"
```

为使static指令工作，实现ACE_Svc_Manager服务的对象代码必须与主看守驱动可执行程序静态地链接在一起。此外，ACE_Svc_Manager对象必须在动态配置发生前插入Service Repository（ACE_Service_Config构造器自动完成此工作）。由于这些限制，如果不首先从Service Repository中将其移除，静态配置的服务就不能在运行时重配置。

服务器日志看守的主驱动程序通过下面的代码实现：

```
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;

}
```

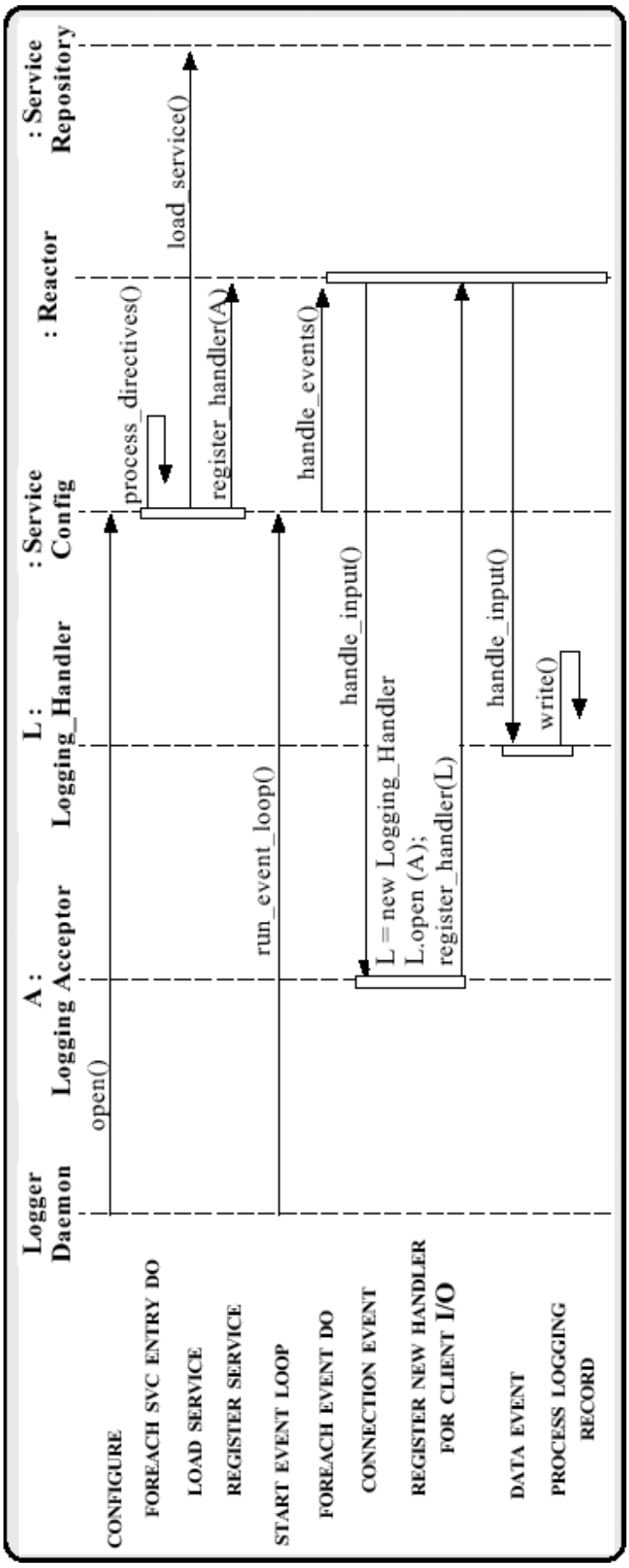

图1-15 服务器日志看守的交互图

图1-15描述多种构架与相互协作、以提供日志服务的应用特有对象之间的运行时交互。看守配置在ACE_Service_Config::open方法中完成。该方法查询下面的svc.conf文件，它指定将被配置进看守的服务：

```
static ACE_Svc_manager -p 911

dynamic Logger Service_Object *

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

通过将指定的ACE_Service_Object插入ACE_Service_Repository，并在ACE_Reactor上登记服务对象处理器的ACE_Event_Handler部分，来对svc.conf文件中的每个服务配置条目进行处理。

当所有的配置活动完成时，上面所示的主驱动程序调用ACE_Service_Config的run_reactor_event_loop方法。该方法进入事件循环，持续地调用ACE_Reactor::handle_events服务分派方法。如图1-9所示，该分派函数阻塞并等待事件的发生（比如来自客户的连接请求或I/O）。在这些事件发生时，ACE_Reactor自动分派先前登记的事件处理器，以执行指定的应用特有服务。

ASX构架还响应触发看守运行时重配置的外部事件。无论何时正在执行的基于ASX的看守接收到预指定的外部事件（比如UNIX SIGHUP信号），就会执行上面所述的动态配置步骤。取决于svc.conf文件的已被更新的内容，服务可以被增加、挂起、恢复或从看守中移除。

ASX构架的动态重配置机制使得开发者无需进行大量的重新开发和安装工作，就可以修改服务器日志看守的功能或调谐性能。例如，调试日志服务的一个有问题的实现只需要动态地重安装一个功能等价的服务，在其中包含有额外的手段来帮助隔离错误行为的来源。注意无须修改、重编译、重链接或重启当前正在执行的服务器日志看守，就可完成此重安装过程。

1.4.2 分布式PBX监控系统

图1-16演示专用分组交换机（PBX）电信交换监控系统的客户/服务器体系结构，该系统使用ASX构架组件[20]实现。在此分布式通信系统中，服务器经由高速通信链路接收并处理由一或多个与服务器相连的PBX产生的状态信息。服务器转换此状态信息，并将其通过网络转发给客户终端系统，再由后者以图形方式显示给终端用户。终端用户通常是超级用户，他们使用PBX状态信息来监控系统全体成员的性能，并预测资源的分配、以满足客户的需求。

与服务器相连的PBX设备由Device_Adapter ACE_Module控制。该ACE_Module使服务器的其余部分与PBX特有的通信特性相屏蔽。Device_Adapter ACE_Module的读端维护一组Device_Handler对象（每个PBX一个），它们负责将到来的设备事件解析并转换成规范的、不依赖于PBX的消息对象；这种对象在一种在[6]中描述的灵活的消息管理类之上构建。

在初始化之后，到来的规范消息对象被传递给Event_Analyzer ACE_Module的读端。该Module为服务器实现应用特有的功能。在Event_Analyzer中维护的一个内部寻址表被用于确定哪些客户应接收消息对象。在Event_Analyzer确定适当的目的地后，消息对象被转发给 Multicast_Router ACE_Module的读端。

Multicast_Router ACE_Module是一个可复用组件，它使应用特有的服务器代码的余下部分与客户/服务器交互的知识，以及对通信协议的特定选择屏蔽开。通过建立连接到Multicast_Router ACE_Module，客户预订接收服务器发布的事件。Multicast_Router ACE_Module的写端接受来自客户的

连接请求，并创建分离的Client_Handler对象来管理每一个客户连接。该Client_Handler对象处理在服务器和与之相关联的客户之间所有的后继数据传输和控制操作。一旦客户与服务器连接上，客户就指明它希望监控的PBX事件的类型。从这一点开始，当Multicast_Router的读端接收到来自Event_Analyzer的消息对象时，它就自动将此消息多点发送给所有预订接收该消息对象中封装的特定类型的事件的客户。

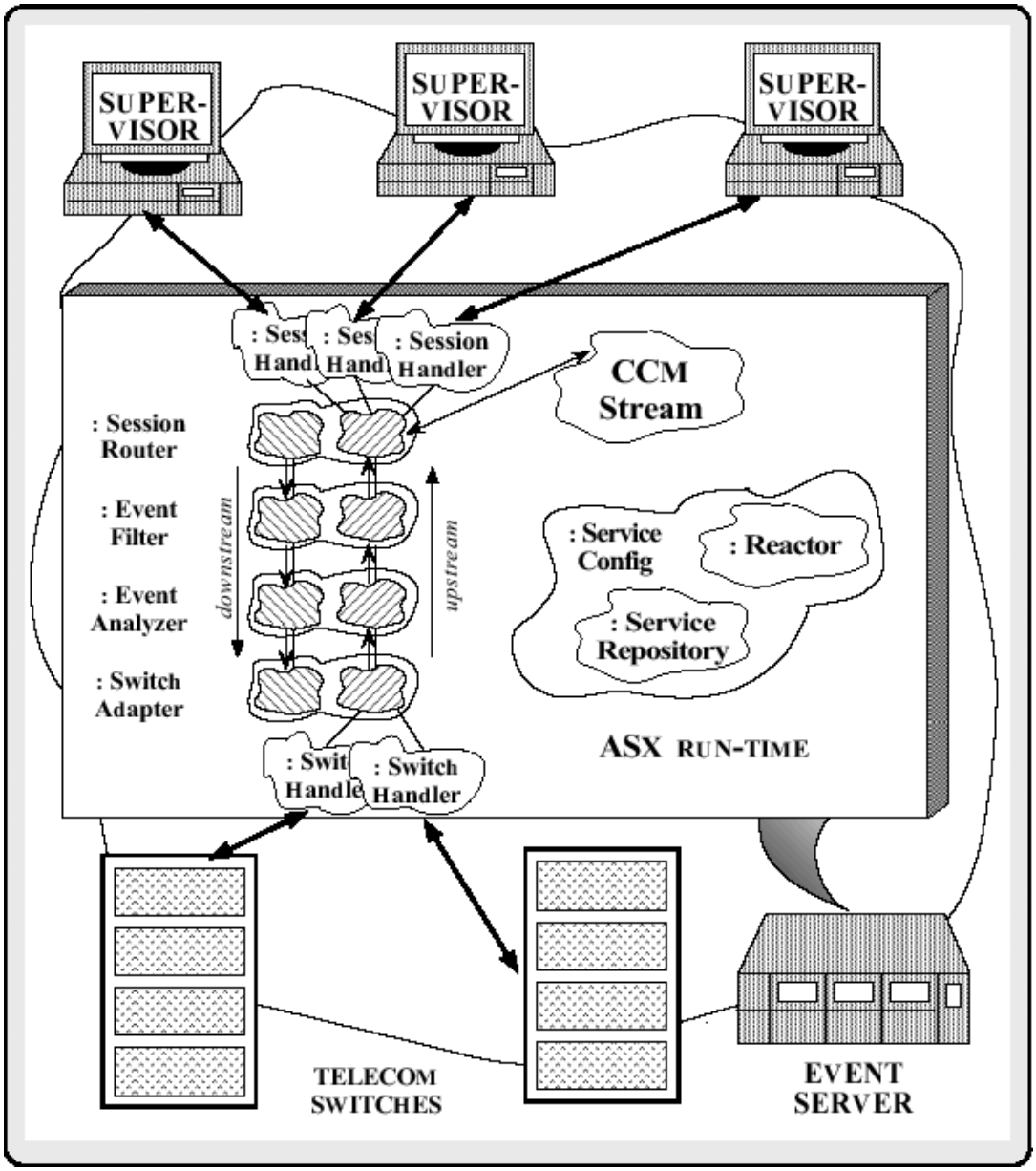

图1-16 PBX应用的ASX组件

ACE_Service_Config对象被服务器用于控制在安装时静态配置、或在运行过程中动态配置的流模块组件的初始化和终止。ACE_Service_Config对象包含有ACE_Reactor事件多路分离器的实例，用于将到来的客户消息分派给适当的Client_Handler或Device_Handler事件处理器。自客户到达的控制消息沿着流的写端向下发送，开始是Multicast_Router，接着是从相互连接的流ACE_Module的写端到Device_Adapter，后者再将控制消息发送给适当的PBX设备。同样地，Reactor检测来自PBX设备的事件，从Device_Adapter ACE_Module开始，沿着流向上分派它们。

1.4.3 服务器配置

组成PBX服务器的ACE_Module可以在任何时候配置进服务器。ASX构架通过使用svc.conf配置脚本驱动的显式动态链接来提供这样的灵活性。下面的配置脚本指示哪些服务将被动态链接进服务器的地址空间：

```
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"

}
```

该配置脚本指示各层次相关的服务被动态链接和推入Server Stream的顺序。在应用初始化过程中，Service Config类解析此配置脚本，并执行每一条目的指令。

Server Stream由三个动态配置进服务器的ACE_Module（Device_Router、Event_Analyzer，和Multicast_Router）组成。指定的共享对象文件被动态链接进服务器（如dynamic指令所指定的）。随后通过调用_alloc函数，Module对象的一个实例就被从共享对象库中提取出来。如下面所描述的，如果需要的话，这些模块可随后被更新和重新链接（例如，安装ACE_Module的更新版本），而无须完全终止执行中的PBX服务器。

1.4.4 服务器重配置

将服务静态配置进通信软件应用有着许多缺点。例如，如果过多的服务被配置进应用的服务器端，并且有过多的活动客户同时访问这些服务，就有可能导致性能瓶颈。相反，将过多的服务配置进客户端也有可能导致性能瓶颈，因为客户常常运行在不那么强大的终端系统上。一般而言，要预先确定适当的应用服务的划分是困难的，因为处理特性和工作负载可能会随着时间而发生变化。因而，ASX构架的一个主要目标就是开发面向对象的服务配置机制，允许开发者将关于哪些服务运行在客户端，哪些运行在服务器端的决定推迟到开发周期的非常迟后的阶段（也就是，安装时或运行时）。

为了便利灵活的重配置，ASX构架提供的运行时控制环境使开发者能够在安装时静态地、或在运行过程中动态地变更他们的应用服务的配置。这样的机制是有用的，因为不同的OS/硬件平台和不同的网络特性常常需要不同的服务配置。例如，在一些配置中，服务器执行大多数工作，而在其他配置中，客户完成更多的工作。而且，在不同的环境下（比如可用的是多处理器服务器平台，还是高速网络），可能需要不同的终端系统配置。图1-17演示了在服务器的处理构成主要瓶颈时，怎样对在图1-16中所示的配置进行变更，以在分布式环境中高效地进行运作。

该重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
```

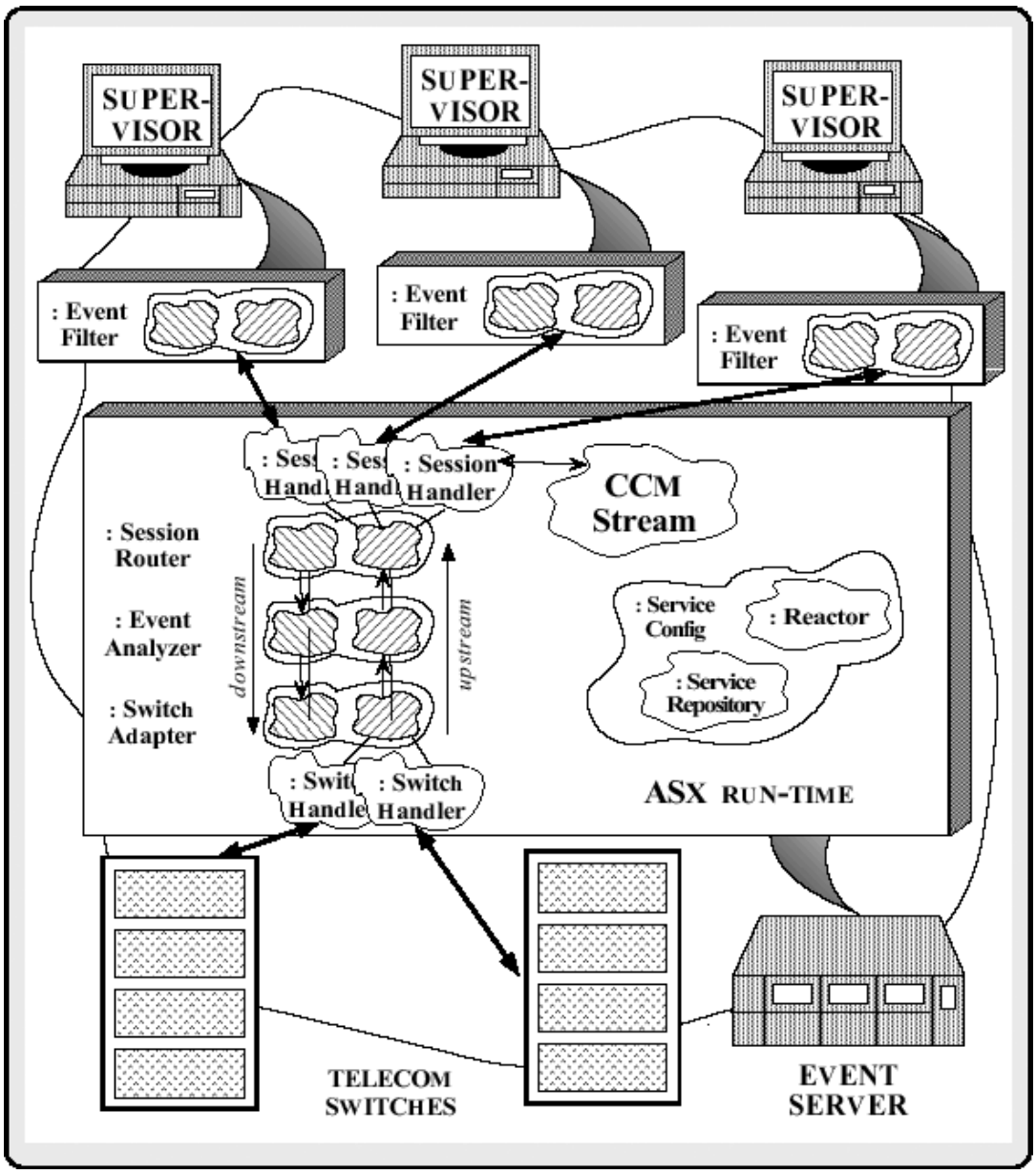

图1-17 PBX监控系统的重配置

这个新脚本通过从服务器的流中动态地解除ACE_Module的链接，并动态地将它们链接进每个客户的流中[20]，将处理功能从服务器移到了客户。ASX构架替换了以前的使用特别技术的体系结构（比如参数传递和共享内存），以在服务器中的相关服务间交换消息。与前面的方法相对照，ASX构架提供的高度统一的ACE_Module互连机制极大地改善了可移植性和可配置性。

1.5 结束语

ACE自适配通信环境是由许多OO组件组成的工具包，它通过实施成功的设计模式和软件体系结构，帮助降低了分布式软件的复杂性。ACE将许多通用的与通信相关的功能（比如本地和远地IPC[4]、事件多路分离和服务处理器分派[14]、服务初始化[16, 17]、含有整体式和层次化服务的分布式应用[20]的配置机制、分布式日志[13]，以及服务内部及服务间的并发）统一进可复用的OO组件和构架中。

ACE可在http://www.cs.wustl.edu/~schmidt/ACE.html自由获取。该发布含有源代码、文档，以及在圣路易斯的华盛顿大学开发的测试实例驱动程序。目前ACE正在许多公司中用于开发通信软件，其中包括Bellcore、西门子、DEC、摩托罗拉、爱立信、柯达，和McDonnell Douglas。ACE已被移植到Win32（也就是，Win95、WinNT、Win2K），大多数版本的UNIX（例如，SunOS 4.x和5.x、SGI IRIX、HP-UX、OSF/1、AIX、Linux和SCO），以及POSIX系统（比如VxWorks和MVS OpenEdition）。同时有C++[6]和Java[48]版本的ACE可用。

参考文献

[1] F. P. Brooks, *The Mythical Man-Month*. Reading, MA: Addison-Wesley, 1975.

[2] S. J. Leffler, M. McKusick, M. Karels, and J. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1989.

[3] S. Rago, *UNIX System V Network Programming*. Reading, MA: Addison-Wesley, 1993.

[4] D. C. Schmidt, T. H. Harrison, and E. Al-Shaer, "Object-Oriented Components for High-speed Network Programming," in *Proceedings of the 1st Conference on Object-Oriented Technologies and Systems*, (Monterey, CA), USENIX, June 1995.

[5] G. Booch, *Object Oriented Analysis and Design with Applications (2nd Edition)*. Redwood City, California: Benjamin/Cummings, 1993.

[6] D. C. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications," in *Proceedings of the 6th USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, April 1994.

[7] D.E.ComerandD.L.Stevens, *Internetworking with TCP/IP Vol III: Client - Server Programming and Applications*.Englewood Cliffs, NJ: Prentice Hall, 1992.

[8] W. R. Stevens, *UNIX Network Programming, First Edition*. Englewood Cliffs, NJ: Prentice Hall, 1990.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[10] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching," in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), Reading, MA: Addison-Wesley, 1995.

[11] R. G. Lavender and D. C. Schmidt, "Active Object: an Object Behavioral Pattern for Concurrent Programming," in *Pattern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), Reading, MA: Addison-Wesley, 1996.

[12] R. Johnson and B. Foote, "Designing Reusable Classes," *Journal of Object-Oriented Programming*, vol. 1, pp. 22-35, June/July 1988.

[13] D. C. Schmidt, "The Reactor: An Object-Oriented Interface for Event-Driven UNIX I/O Multiplexing (Part 1 of 2)," *C++ Report*, vol. 5, February 1993.

[14] D. C. Schmidt, "The Object-Oriented Design and Implementation of the Reactor: A C++ Wrapper for UNIX I/O Multiplexing (Part 2 of 2)," *C++ Report*, vol. 5, September 1993.

[15] T. H. Harrison, D. C. Schmidt, and I. Pyarali, "Asynchronous Completion Token: an Object Behavioral Pattern for Efficient Asynchronous Event Handling," in *The 3rd Annual Conference on the Pattern Languagesof Programs (Washington University technical report#WUCS-97-07)*, (Monticello, Illinois), pp. 1-7, February 1997.

[16] D. C. Schmidt, "Design Patterns for Initializing Network Services: Introducing the Acceptor and Connector Patterns," *C++ Report*, vol. 7, November/December 1995.

[17] D. C. Schmidt, "Connector: a Design Pattern for Actively Initializing Network Services," *C++ Report*, vol. 8, January 1996.

[18] D. C. Schmidt, "Acceptor and Connector: Design Patterns for Initializing Communication Services," in *The 1st European Pattern Languages of Programming Conference (Washington University technical report #WUCS-97-07)*, July 1997.

[19] D. C. Schmidt, "IPC SAP: An Object-Oriented Interface to Interprocess Communication Services," *C++ Report*, vol. 4, November/December 1992.

[20] D. C. Schmidt and T. Suda, "An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems," *IEE/BCS Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Systems)*, vol. 2, pp. 280 - 293, December 1994.

[21] P. Jain and D. C. Schmidt, "Service Configurator: APattern for Dynamic Configuration and Reconfiguration of Communication Services," in *The 3rd Pattern Languagesof Programming Conference (Washington University technical report #WUCS-97- 07)*, February 1997.

[22] D. C. Schmidt, "An OO Encapsulation of Lightweight OS Concurrency Mechanisms in the ACE Toolkit," Tech. Rep. WUCS-95-31, Washington University, St. Louis, September 1995.

[23] D. C. Schmidt and C. D. Cranor, "Half-Sync/Half-Async: an Architectural Pattern for Efficient and Well-structured Concurrent I/O," in *Pattern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), Reading, MA: Addison-Wesley, 1996.

[24] D. C. Schmidt and T. Harrison, "Double-Checked Locking - An Object Behavioral Pattern for Initializing and Accessing Thread-safe Objects Efficiently," in *The 3rd Pattern Languages of Programming Conference (Washington University technical report #WUCS-97-07)*, February 1997.

[25] Bjarne Stroustrup and Margret Ellis, *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[26] I. Pyarali, T. H. Harrison, and D. C. Schmidt, "Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging," *USENIX Computing Systems*, vol. 9, November/December 1996.

[27] R. Davis, *Win32 Network Programming*. Reading, MA: Addison-Wesley, 1996.

[28] D. L. Presotto and D. M. Ritchie, "Interprocess Communication in the Ninth Edition UNIX System," *UNIX Research SystemPapers, Tenth Edition*, vol. 2, no. 8, pp. 523 - 530,1990.

[29] W. R. Stevens, *Advanced Programming in the UNIX Environment*. Reading, Massachusetts: Addison Wesley, 1992.

[30] R. Gingell, J. Moran, and W. Shannon, "Virtual Memory Architecture in SunOS," in *Proceedings of the Summer 1987 USENIX Technical Conference*, (Phoenix, Arizona), 1987.

[31] J. Eykholt, S. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams, "Beyond Multiprocessing... Multithreading the SunOS Kernel," in *Proceedings of the Summer USENIX Conference*, (San Antonio, Texas), June 1992.

[32] IEEE, *Threads Extension for Portable Operating Systems (Draft 10)*, February 1996.

[33] D. C. Schmidt, "Transparently Parameterizing Synchronization Mechanisms into a Concurrent Distributed Application," *C++ Report*, vol. 6, July/August 1994. 24

[34] G. Booch and M. Vilot, "Simplifying the Booch Components," *C++ Report*, vol. 5, June 1993.

[35] R. Gingell, M. Lee, X. Dang, and M. Weeks, "Shared Libraries in SunOS," in *Proceedingsof the Summer 1987 USENIXTechnical Conference*, (Phoenix, Arizona), 1987.

[36] W. W. Ho and R. Olsson, "An Approach to Genuine Dynamic Linking," *Software: Practice and Experience*, vol. 21, pp. 375 - 390, Apr. 1991.

[37] H. Custer, *Inside Windows NT*. Redmond, Washington: Microsoft Press, 1993.

[38] D. C. Schmidt and P. Stephenson, "Experiences Using Design Patterns to Evolve System Software Across Diverse OS Platforms," in *Proceedings of the 9th European Conference on Object-Oriented Programming*, (Aarhus, Denmark), ACM, August 1995.

[39] D. Ritchie, "A Stream Input - Output System," *AT&TBell Labs Technical Journal*, vol. 63, pp. 311 - 324, Oct. 1984.

[40] N. C. Hutchinson and L. L. Peterson, "The x-kernel: An Architecture for Implementing Network Protocols," *IEEE Transactions on Software Engineering*, vol. 17, pp. 64 - 76, January 1991.

[41] J. M. Zweig, "The Conduit: a Communication Abstraction in C++," in *Proceedings of the 2nd USENIX C++ Conference*, pp. 191–203, USENIX Association, April 1990.

[42] D. C. Schmidt and T. Suda, "Transport System Architecture Services for High-Performance Communications Systems," *IEEE Journal on Selected Areas in Communication*, vol. 11, pp. 489–506, May 1993.

[43] D. C. Schmidt, B. Stiller, T. Suda, A. Tantawy, and M. Zitterbart, "Language Support for Flexible, Application-Tailored Protocol Configuration," in *Proceedings of the 18th Conference on Local Computer Networks*, (Minneapolis, Minnesota), pp. 369–378, IEEE, Sept. 1993.

[44] A. McRae, "Hardware Profiling of Kernels," in *USENIX Winter Conference*, (San Diego, CA), USENIX Association, Jan. 1993.

[45] S. Saxena, J. K. Peacock, F. Yang, V. Verma, and M. Krishnan, "Pitfalls in Multithreading SVR4 STREAMS and other Weightless Processes," in *Proceedings of the Winter USENIX Conference*, (San Diego, CA), pp. 85–106, Jan. 1993.

[46] D. C. Schmidt, D. F. Box, and T. Suda, "ADAPTIVE: A Dynamically Assembled Protocol Transformation, Integration, and eValuation Environment," *Journal of Concurrency: Practice and Experience*, vol. 5, pp. 269–286, June 1993.

[47] D. C. Schmidt, "A Family of Design Patterns for Application-level Gateways," *The Theory and Practice of Object Systems (Special Issue on Patterns and Pattern Languages)*, vol. 2, no. 1, 1996.

[48] P. Jain and D. Schmidt, "Experiences Converting a C++ Communication Software Framework to Java," *C++Report*, vol. 9, January 1997.