第3章 IPC SAP：用于高效、可移植和灵活的网络编程的C++包装

Douglas C. Schmidt

本论文的一个扩展版本[1]（含有在以太网和ATM网络上的性能评测）可在 http://www.cs.wustl.edu/schmidt/COOTS-95.ps.Z处获取。

3.1 介绍

本论文描述采用C++包装类来封装OS进程间通信（IPC）机制的面向对象（OO）技术，并聚焦于ACE构架[2]中的IPC SAP组件所提供的C++包装。ACE是一组可复用C++类库和OO构架组件，它们简化了可移植、高性能和实时通信软件的开发。IPC SAP是ACE中的一种组件，它提供了一个OO网络编程接口族来封装socket接口[3]、系统V传输层接口（TLI）[4]、SVR4 STREAM管道[5]、UNIX FIFO[6]和Windows NT命名管道[7]。

IPC SAP中的C++包装将开发者及应用与OS的本地和远地IPC机制的不可移植的细节屏蔽开。IPC SAP封装的IPC机制包括标准的面向连接的和无连接的协议，比如在UNIX/POSIX、Win32和实时操作系统中可用的TCP、UDP和IPX/SPX。IPC SAP利用OO技术和C++特性来提供一组丰富的组件，简化了高效、可移植和灵活的通信软件的开发。

本论文被组织如下：3.2概述用于编写通信软件的抽象层级；3.3描述现有的网络编程接口；3.4概述它们的局限；3.5介绍IPC SAP的OO设计和实现，并解释它是怎样克服现有网络编程接口的局限的；3.6详细检查socket、TLI、STREAM管道和FIFO的C++包装；3.7演示若干例子，使用IPC SAP来实现客户/服务器流式应用；3.8讨论指导IPC SAP的设计的原则；3.9则总结使用C++来为本地OS接口开发OO包装的优点和缺点。

3.2 网络编程接口综述

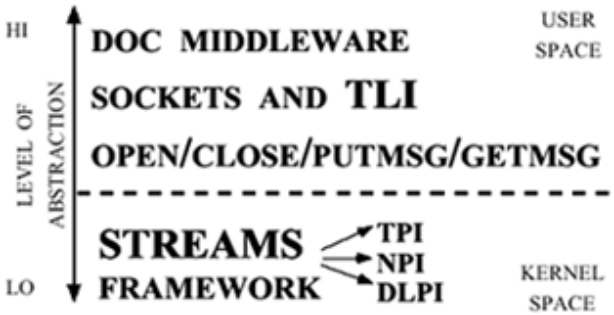

图3-1 网络编程的抽象层次

编写健壮、可扩展和高效的通信软件是困难的。开发者必须掌握许多复杂的OS和通信概念，比如：

- 网络寻址和服务标识。
- 表示转换（比如加密、压缩和有可选处理器字节序的异种终端系统间的网络字节序转换）。
- 进程和线程创建及同步。
- 本地和远地进程间通信（IPC）机制的系统调用和库函数接口。

许多编程工具和接口已被创建用来帮助简化通信软件的开发。图3-1演示在当代的OS平台（比如UNIX和Win32）上可用的IPC接口。如图所示，应用可以访问本地和远地IPC的网络编程接口的若干层次。这一部分的余下部分概述每一抽象层，范围从高级分布式对象计算（DOC）中间件到用户级网络编程接口，再到低级的内核编程接口。

3.2.1 DOC中间件

以一种“请求—响应”方式来与客户交换数据的应用常常使用分布式对象计算（DOC）中间件来开发。DOC中间件的宽泛定义包括对象请求代理（ORB），像CORBA[8]和Microsoft的DCOM[9]；以及面向消息的中间件，像Mqseries。DOC中间件使分布式应用开发的许多麻烦而易错的方面得以自动完成，包括：

- 认证、授权和数据安全；
- 服务定位和绑定；
- 服务登记和启用；
- 事件多路分离和分派；
- 在像TCP这样的面向字节流的通信协议之上实现消息帧；
- 涉及网络字节序和参数整编（marshaling）的表示转换问题。

此外，DOC中间件还提供一组高级工具，比如IDL编译器和名字服务，将开发者与较低级的OS系统调用（它们在网络上传输和接收包）的复杂性屏蔽开。

3.2.2 用户级网络编程接口

DOC中间件通常建构在网络编程接口之上，比如socket[3]、TLI[4]，或Windows NT命名管道。与较高级的DOC中间件相比，通过用户级网络编程接口来开发应用有若干优点：

- *最小化不必要功能的时间和空间开销*：应用可以忽略不必要的功能，比如ASCII数据或内存区域的表示层转换。

- *允许对行为进行细粒度控制*：网络编程接口使得对行为的控制粒度更为精细，比如允许多点传输和信号驱动的异步I/O。

- *增强可移植性*：像socket这样的网络编程接口可用于广泛的OS平台，而DOC中间件则不是这样。

对于一类特定的被称为“流式应用”[10]的应用，DOC中间件提供的请求 - 响应和“单路”通信机制并不特别适用。流式应用的特征是高带宽、无类型字节流或相对简单的数据类型的长持续时间的通信，对通信性能有着严格的要求。交互式电话会议、医学成像和视频点播是流式应用的范例。

流式应用服务质量需求（QoS）常常不能忍受DOC中间件所带来的性能开销[11]。这样的开销源于未优化的表示格式转换、未优化的内存管理、低效的接收者端多路分离、停 - 等流控制、同步的发送端方法请求，以及非自适配的重发定时器方案。传统上，满足流式应用的需求涉及到对像socket[3]或TLI[4]这样的网络编程接口的直接访问。

3.2.3 内核级网络编程接口

在OS内核的通信子系统中有较低级的网络编程接口。例如，SVR4 putmsg和getmsg系统调用可用于直接访问系统V STREAMS[14]中的传输供应者接口（TPI）[12]和数据链路供应者接口（DLPI）[13]。

还有可能开发像路由器或网络文件系统这样的网络服务，它们整个地驻留在OS内核中[5]。但是，在这一级进行编程通常不能在不同的OS平台间移植。而且，甚至也不能在同一OS的不同版本间移植。

3.2.4 评估

使用用户级或内核级网络编程接口、而不是DOC中间件，通常要更难进行编程。像socket和TLI这样的传统网络编程库缺少类型安全、可移植、可重入和可扩展的接口。例如，socket端点通过弱类型的描述符实现，从而增加了在运行时发生微妙错误的潜在可能性[15]。

本文中描述的IPC SAP组件通过封装网络编程接口的大量复杂性，在设计空间中提供了一个“中点”。IPC SAP的目标是提高通信软件的*正确性*、*易用性*和*可移植性/可复用性*，而又不损害它的性能。IPC SAP与ACE构架[2]一起发布，并被用于许多公司的商业项目中，包括Bellcore、波音、朗讯、摩托罗拉、Nortel、SAIC和西门子，等等。

3.3 网络编程接口考察

这一部分考察像socket和TLI这样的传统网络编程接口的行为和局限。

3.3.1 背景

在许多操作系统中，比如UNIX和Win32，通信协议栈驻留在OS内核的保护地址空间中。运行在用户地址空间中的应用程序通过像socket、TLI或Win32命名管道这样的接口来访问驻留内核的协议栈。这些接口对本地和远地的通信端点这样来进行管理：允许应用打开到远地主机的连接、磋商和启用/禁用特定的选项、交换数据，以及在传输完成时关闭全部或部分连接。

socket和TLI松散地建模在UNIX文件I/O接口之上，后者定义了open、read、write、close、ioctl、lseek和select函数[14]。但是，socket和TLI还提供了额外的功能，没有直接被标准的UNIX文件I/O接口所支持。这些额外的功能源于文件I/O和网络I/O之间语法和语义的差异。例如，在分布式环境中，UNIX系统用于标识文件的路径名并非是全局唯一的。因此，采用了一种不同的命名方案（比如IP主机地址）来唯一地标识网络应用。

socket和TLI接口提供类似的功能。它们支持一种多*通信域*[3]的通用接口。域指定协议族和地址族。每个协议族都含有一个协议栈，实现域中特定的通信类型。常用的协议栈提供可靠、双向、面向连接的消息和流的服务（例如，像TCP、TP4和SPX这样的协议），以及不可靠、无连接的数据报服务（例如，像UDP、CLNP和IPX这样的协议）。

地址族定义地址格式（例如，地址的字节长度、字段的数目和类型和字段顺序）以及一组驻留内核的对地址格式进行解释的函数（例如，决定一个IP数据报要发到哪个子网）。

3.3.2给出了socket综述，3.3.3简要描述了TLI，3.3.4讨论STREAM管道，而3.3.5讨论UNIX FIFO。对这些接口的完整讨论超出了本论文的范围（更多详情参见[5, 3, 7, 6, 16]）。

3.3.2 socket接口

socket接口最初是在BSD UNIX中开发的，用以提供TCP/IP协议组[3]的接口。从应用的视点来看，socket是本地的通信端点，与驻留在本地或远地的地址绑定在一起。socket可通过*句柄*（也称为*描述符*）来访问。

在UNIX中，socket句柄与其他句柄共享同一个名字空间，例如，文件、管道和终端设备句柄。句柄提供一种封装机制，将应用与内部的OS数据结构的知识屏蔽开。句柄标识特定的由OS维护的通信端点。

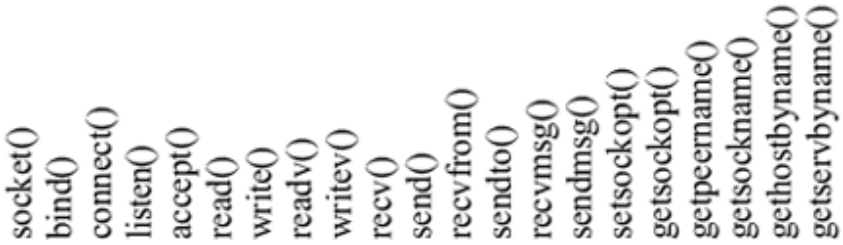

图3-2 socket接口中的函数

socket接口如图3-2所示。该接口含有大约两打的系统调用，可分为以下类型：

本地管理：socket接口为管理本地上下文信息提供以下函数：

- socket：分配最小的未用socket句柄；
- bind：将socket句柄与本地或远地地址相关联；
- getsockname和getpeername：分别确定socket所连接的本地或远地地址；
- close：释放socket句柄，使它可用于后面的复用。

连接建立和连接终止：socket接口为建立和终止连接提供以下函数：

- connect：客户通常使用connect来*主动地*与服务器建立连接；
- listen：服务器使用listen来指示它想要*被动地*侦听进入的客户连接请求；
- accept：服务器使用accept来创建新的通信端点，以为客户服务；
- shutdown：有选择地终止一个双向连接的读端和/或写端流。

数据传输机制：socket接口提供以下函数来发送和接收数据：

- read/write：通过特定句柄接收和传输数据缓冲区；
- send/recv：与read/write类似，但它们提供一个额外的参数来控制特定的socket特有操作（比如交换“紧急”数据，或“偷看”接收队列中的数据，而又不把它从队列中移除）；
- sendto/recvfrom：交换无连接数据报；

- readv/writev：分别支持“分散读”和“集中写”语义（这些操作优化用户/内核模式切换并简化内存管理）；

- sendmsg/recvmsg：通用函数，包含了所有其他数据传输函数的行为。对于UNIX域的socket，sendmsg和recvmsg函数还提供在同一主机的任意进程间传递“访问权限”（比如打开文件句柄）的能力。

注意这些接口也可被用于其他类型的I/O，比如文件和终端。

选项（option）管理：socket接口定义以下函数，允许用户改变socket行为的缺省语义：

- setsockopt和getsockopt：修改或查询在协议栈不同层次中的选项。选项包括多点传送、广播，以及设置/获取发送和接收传输缓冲区的大小；

- fcntl和ioctl：是UNIX系统调用，使在socket上能够进行异步I/O、非阻塞I/O，以及紧急消息递送。

除了上面描述的socket函数，通信软件还可使用以下标准库函数和系统调用：

- gethostbyname和gethostbyaddr：处理网络寻址的多种情况，比如映射主机名到IP地址；

- getservbyname：通过服务的端口号或人类可读的名字来对它们进行标识；

- ntohl、ntohs、htonl、htons：执行网络字节序转换；

- select：在成组的打开的句柄上执行基于I/O和基于定时器的事件多路分离。

3.3.3 TLI接口

TLI是访问通信协议栈的一种可选接口。基本上，TLI提供一组和socket一样的服务。但是，它更加强调使应用与底层传输供应者的细节屏蔽开来。[5]详细地讨论TLI。

3.3.4 STREAM管道

STREAM管道是对原始的UNIX管道机制的增强。早先的UNIX管道提供单一的从作者端点到读者端点的单向字节流。STREAM管道支持在执行在同一主机上的进程和/或线程间进行双向的字节流和按优先级排序的消息的递送[16]。尽管pipe系统调用接口保持不变，STREAM管道还提供了额外的功能，大致等价于UNIX域的SOCK_STREAM socket。但是它们比UNIX域的socket要更灵活一些，因为它们使STREAM模块可被“压入”或是“弹出”管道端点。

缺省地，流管道仅在它的两个端点间提供单一数据通道。因此，如果多个发送者向管道写入，所有的消息都被放置到同一个通信通道中。这常常太过受限，因为多路分离单个通道上来自多个客户的数据必须进行人工编程。例如，每个消息都必须包含一个标识符，使接收者能够确定是哪一个发送者传输的消息。通过使用已安装的（mounted）STREAM管道和connld模块[17]，应用可以将一个单独的非多路复用的I/O通道专用于服务器和客户的每一实例之间。

STREAM管道和connld的工作方式如下：服务器调用pipe系统调用，创建双向通信端点。Fattach系统调用可以将管道句柄安装（mount）到UNIX文件系统中的指定位置。通过将connld STREAM模块压入STREAM管道的已安装的一端，就可以创建服务器应用。在运行服务器的同一主机上运行的客户应用随即打开与已安装管道相关联的文件。在这一点，connld模块确保客户和服务器分别收到一个唯一的I/O句柄，标识一个非多路复用、双向的通信信道。

3.3.5 FIFO接口

UNIX FIFO（也称为命名管道[6]）是STREAM管道的受限形式。不像STREAM管道，FIFO仅提供单向的、从一或多个发送者到单个接收者的数据通道。而且，来自不同发送者的消息都被放入同一个通信通道中。因此，必须在每个消息中明确地包括某种类型的多路分离标识符，以使接收者能够确定是哪一个发送者传输的消息。

SVR4 UNIX中基于STREAM的FIFO实现同时提供消息和字节流递送语义。相反，一些早期版本的UNIX（比如SVR3和SunOS 4.x）仅提供面向字节流的FIFO。因此，除非总是使用定长消息，每个经由FIFO发送的消息必须通过某种形式的字节计数或特殊结束符来进行区分，从而使接收者能够从FIFO字节流中提取消息。FIFO在[5, 6, 16]中进一步描述。

3.4 问题：现有IPC接口的局限

socket、TLI、STREAM管道和FIFO为访问本地和远地IPC机制提供了广泛的接口。但是这些接口都有若干局限。下面的讨论聚焦于socket接口的局限，但是其中的大多数也适用于其他网络编程接口。

高错误可能性：在UNIX和Win32中，socket、文件、管道、终端和其他设备的句柄是用“弱类型”的整数或指针值来标识的。这样的弱类型检查会导致微妙的运行时错误。例如，socket接口无法确保用于不同通信角色（比如主动 vs. 被动连接建立，或数据报 vs. 流通信）的socket函数的正确使用。而且，编译器无法检测或阻止句柄的错误使用，因为句柄是弱类型的。因而，可能会不正确地对句柄进行操作，例如，在为建立连接而设置的句柄上调用数据传输操作。

```
#include <sys/types.h>

#include <sys/socket.h>


const int PORT_NUM = 10000;
```

```
int buggy_echo_server (void)
{
sockaddr s_addr;
int length; // (1) uninitialized variable.
char buf[BUFSIZ];
int s_fd, n_fd;

// Create a local endpoint of communication.
if (s_fd = socket (PF_UNIX, SOCK_DGRAM, 0) == -1)
       return -1;
// Set up the address information to become a server.
// (2) forgot to "zero out" structure first...
s_addr.sin_family = AF_INET;
// (3) used the wrong address family ...
s_addr.sin_port = PORT_NUM;
// (4) forgot to use htons() on PORT_NUM...
s_addr.sin_addr.s_addr = INADDR_ANY;
if (bind (s_fd, (sockaddr *) &s_addr, sizeof s_addr) == -1)
       perror ("bind"), exit (1);
// (5) forgot to call listen()

// Create a new endpoint of communication.
// (6) doesn't make sense to accept a SOCK_DGRAM!
if (n_fd = accept (s_fd, &s_addr, &length) == -1)
{
       // (7) Omitted a crucial set of parens...
       int n;
       // (8) doesn't make sense to read from the s_fd!
       while ((n = read (s_fd, buf, sizeof buf)) > 0)
              // (9) forgot to check for "short-writes"
              write (n_fd, buf, n);
       // Remainder omitted...
}
}
```

图3-3 臭虫成灾的Echo服务器

图3-3描述下列在使用socket接口时发生的微妙和“过于常见”的错误：

1. 忘记将accept的len参数初始化为struct sockaddr_in的大小；
2. 忘记将socket地址结构中的所有字节初始化为“0”；
3. 使用了与socket的协议族相矛盾的地址族类型；
4. 忽略了使用htons库函数来将端口号从主机字节序转换到网络字节序，反之亦然。
5. 创建被动模式的SOCK_STREAM socket时遗漏了listen系统调用；
6. 对SOCK_DGRAM socket使用了accept函数；
7. 在赋值表达式中错误地遗漏了一组关键的括号；
8. 试图从被动模式socket中读，而这样的socket只能用于接受连接；
9. 没有能适当地检测和处理由于缓冲而发生的“短写”（short-writes）。

上面所列问题中的一些是C的经典问题。例如，如果遗漏了下面这个表达式中的括号

```
if (n_fd = accept (s_fd, &s_addr, &length) == -1)
```

n_fd的值将总是被设为0或者1（取决于accept()是否等于-1）。

一个更深的问题是C数据结构缺乏足够的抽象。例如，通用的sockaddr地址结构使得开发者必须使用强制类型转换来提供Internet域和UNIX域地址的一种继承形式。这些“子类”地址结构，sockaddr_in和sockaddr_un，分别对sockaddr“基类”进行重定义。

一般而言，强制类型转换的使用，与弱类型的、基于句柄的socket接口一起，使得编译器很难在编译时检测错误。相反，错误检查被推延到运行时，这使得错误处理变得更为复杂，并且降低了应用的健壮性。

复杂的接口：socket提供了单一接口来支持多种协议族，像TCP/IP、IPX/SPX、ISO OSI和UNIX域的socket。socket接口含有许多函数，支持不同的*通信角色*（比如主动 vs. 被动连接建立）、*通信优化*（比如在单个系统调用中发送多个缓冲区的writev），以及用于不常使用的操作的*选项*，比如广播、多点传送、异步I/O和紧急数据递送。

尽管socket将这些功能组合进一个通用的接口，所得到的机制仍然是复杂而又难以掌握的。这样的复杂性源于socket接口过于宽泛的和*一维的*（one-dimensional）设计。例如，如图3-2所示，所有函数都出现在单一的抽象层中。这样的设计增加了正确学习和使用socket所需的努力。这样，程序员必须理解整个socket接口，即使他们只使用其中一部分。

但是，如果仔细地检查socket，很清楚该接口可以被分解为下面三个函数簇：

1. *通信服务类型*：也就是，流 vs. 数据报 vs. 有连接的数据报；

2. *通信角色*：也就是，主动的 vs. 被动的（客户通常是主动的，而服务器通常是被动的）
3. *通信域*：也就是，本地 vs. 本地/远地。

图3-4根据这三个标准来对相关的socket函数进行分类：

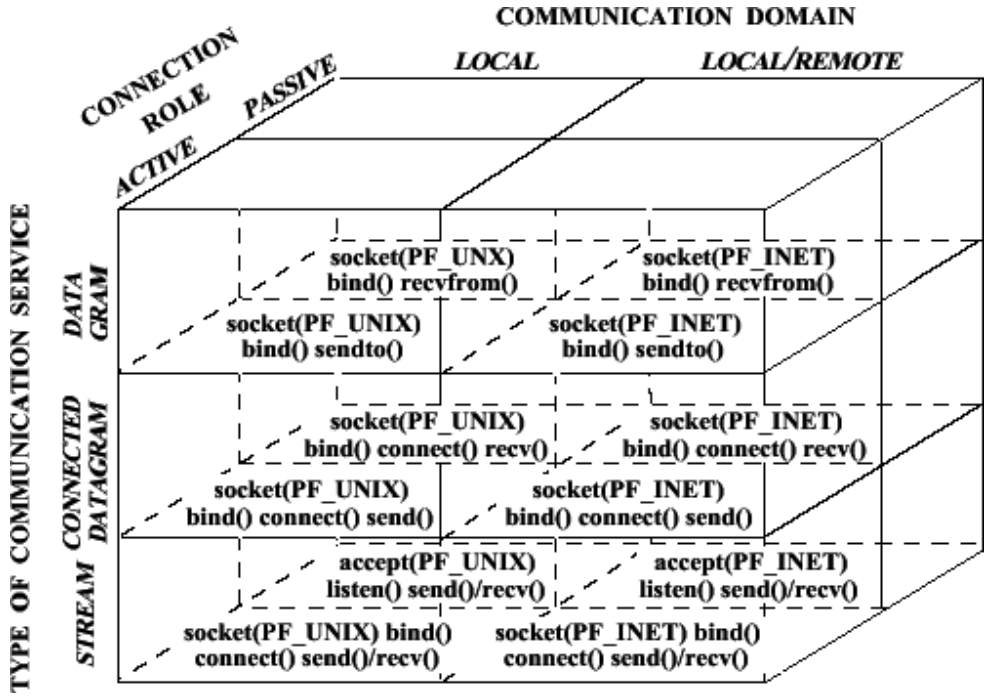

图3-4 socket的各个维度

但是，因为接口是一维的，这样自然的分类被弄得含混不清。3.6演示了怎样将此分类重新构造为一个类层次，以简化socket接口并增强通信软件的类型安全性。

不统一：socket接口的另一问题是它的若干打函数缺乏统一的命名习惯。不统一的命名使得开发者很难确定socket接口的范围。例如，socket、bind、accept和connect之间的相关并不显而易见。其他网络编程接口通过在每个函数前面添加公共前缀来解决这一问题。例如，在TLI库的每个函数前都有t_前缀。

但是，TLI接口也含有有着过于复杂的语义的操作。例如，不像socket，TLI选项处理接口没有以一种标准的方式来规定。这使得开发者很难编写可移植的应用来访问标准的TCP/IP选项。同样地，在qlen > 1的并发服务器中，需要使用微妙的应用级代码来处理t_listen和t_accept的非直观和易错的行为[5]。

3.5 解决方案：IPC SAP C++包装

3.5.1 综述

IPC SAP封装常用的基于句柄的IPC接口，比如socket、TLI、STREAM管道和FIFO。如图3-5所示，IPC SAP被设计为类属的一座“森林”，包括SOCK SAP（封装socket）、TLI SAP（封装TLI接口）、SPIPE SAP（封装UNIX SVR4 STREAM管道接口），以及FIFO SAP（封装UNIX FIFO接口）。

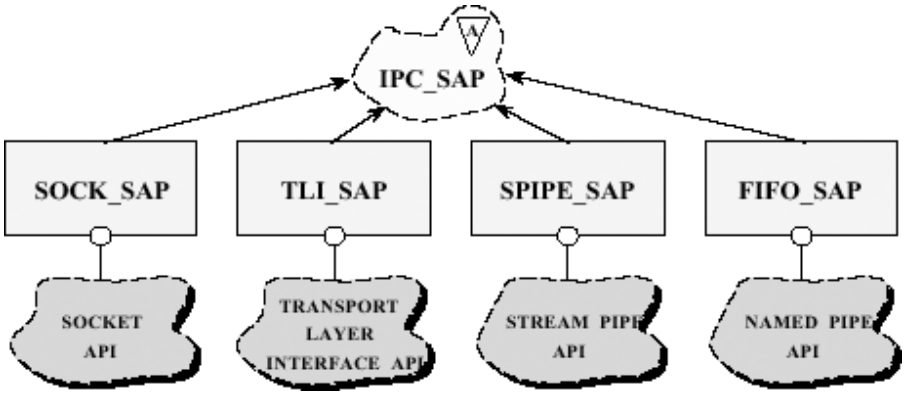

图3-5 IPC SAP类属关系

每个类属都被组织为继承层次。所有子类都给现有IPC机制的子集提供定义良好的接口。在一个层次中的所有子类共同地包含了一种特定通信抽象（比如Internet域或UNIX域的协议族）的全部功能。这一部分描述IPC SAP的设计目标，概述它的类属，并讨论在其OO设计之下的法则。

3.5.2 IPC SAP设计目标

IPC SAP被设计用于改善通信软件的*正确性*、*易学性*和*易用性*、*可移植性*，以及*可复用性*，同时维持高水平的性能和功能。这一部分讨论IPC SAP是怎样实现这些目标的。

3.5.2.1 提高正确性

socket的若干问题都与它的弱类型检查有关。通过只允许对类的实例进行“类型安全”的操作，IPC SAP提高了网络应用代码的正确性。为强制实施类型安全性，IPC SAP确保它的所有对象都通过构造器来适当地初始化。此外，对IPC SAP对象只能进行良好定义的操作。

IPC SAP还被设计用于防止偶然的类型安全性违例。例如。SOCK SAP类属中的组件可防止偶然地对数据报对象进行面向连接的操作。因此，不可能在数据报对象上调用accept方法，在连接器和接受器工厂对象上接收（recv）或发送（send）数据，或是在面向连接的对象上调用sendto方法。

因为IPC SAP类是强类型的，任何执行非法操作的企图都会在编译时、而非运行时被拒绝。图3-14所示的buggy_echo_server的SOCK SAP修正版对这一点进行了演示。此例更正了图3-3中所标识出的

所有socket问题。

3.5.2.2 增强易学性和易用性

简化常用IPC操作的使用是一个与正确性有关的目标。通过提供更简单的接口，开发者能够把注意力集中在编写应用上，而不是与低级网络代码搅在一起。一般而言，IPC SAP这样来简化它的网络编程接口：

提供辅助类，使应用与易错细节相屏蔽：例如，IPC SAP含有如图3-6所示的Addr类层次。该层次通过类型安全的C++接口来支持若干不同的网络寻址格式。Addr层次消除了若干常见的编程错误，这些错误都与直接使用基于C的struct sockaddr数据结构有关系。例如，不再有可能忘记把sockaddr地址结构清零。

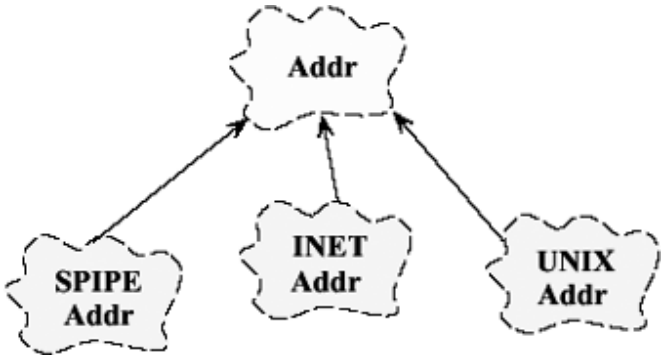

图3-6 IPC SAP地址类层次

组合若干操作，以形成单一操作：例如，ACE_SOCK_Acceptor是用于被动连接建立的工厂。它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）。

为典型的方法参数值提供缺省参数：例如，accept的寻址参数常为NULL指针。为简化编程，这些值在SOCK_Acceptor::accept中作为C++缺省参数被给出，以使程序员不必显式地提供它们。

利用traits（特性）来传达“元类”信息：例如，所有IPC SAP类都含有一组统一的traits。这些traits进行类型定义，以指定与各自的IPC SAP类型相关联的地址类（例如，ACE_INET_Addr）和/或流类（例如，ACE_TLI_Stream）。如下所示：

```
class ACE_SOCK_Connector

{
```

```
public:

// Traits

typedef ACE_INET_Addr PEER_ADDR;

typedef ACE_SOCK_Stream PEER_STREAM;

// ...

};


class ACE_TLI_Connector : public ACE_SOCK

{

public:

// Traits

typedef ACE_INET_Addr PEER_ADDR;

typedef ACE_TLI_Stream PEER_STREAM;

//...

};
```

如3.7所示，traits与C++参数化类型的联合使用支持一种强大的称为“泛型编程”（generic programming）的设计范式 [18]。

3.5.2.3 提高可复用性

在IPC SAP中使用了基于继承的层次分解，以增加多种IPC机制所共享的通用代码的数量。例如，IPC SAP给像fcntl和ioctl这样的较低级的OS设备控制系统调用提供了一种C++接口。通过在不同的子类间共享代码，继承增强了在IPC SAP实现中的复用。

例如，IPC SAP根基类提供的标准方法和数据被其他的派生类所共享。这些共享组件提供句柄和与其相关的set/get方法。此外，还提供了一些方法来在句柄上启用和禁止异步I/O、非阻塞I/O，以及紧急消息递送。

3.5.2.4 可移植性

若干C++特性有助于增强IPC SAP的可移植性。例如，IPC SAP提供一种不依赖于平台的接口，通过使用C++模板来改善通信软件的可移植性。如图3-7所示，SOCK SAP和TLI SAP类的一个子集提供了同样的OO接口。每个平台可能会拥有不同的用于本地和远地网络编程（例如，socket vs. TLI）的底层接口。但是，有可能编写出应用，使用两个类中的任何一个来透明地进行参数化。这增强了应用的跨平台（这些平台可能不同时支持socket和TLI的平台）可移植性。

通过允许应用被它们所需IPC机制的类型参数化，类的使用（相对于独立的函数）有助于简化网络编程。如3.8所讨论的，参数化有助于改善平台间（这些平台支持不同的网络编程接口，比如socket或TLI）的可移植性。

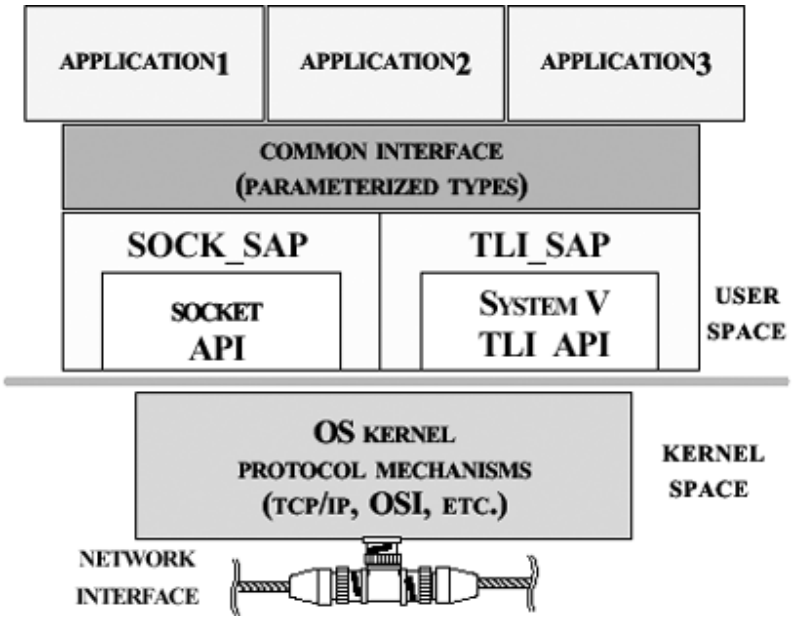

图3-7 使用模板增强可移植性

3.5.2.5 性能

为鼓励开发者用IPC SAP替换现有接口，IPC SAP被设计为能高效地运作。下列技术帮助改善了性能，而又没有牺牲清晰性和模块性：

使用内联函数：许多IPC SAP方法都被指定为C++内联函数，从而消除了调用IPC SAP方法的额外运行时开销。内联是一种合理的方法，因为每个方法都非常短（平均每个方法大约3行）。

避用虚函数：在IPC SAP继承层次中没有使用虚函数，从而改善了性能，因为（1）消除了间接的vtable函数指针分派，以及（2）便利了确实很短而又经常访问的方法（比如发送和接收用户数据）的直接内联。

3.6 IPC SAP的面向对象设计

这一部分描述组成IPC SAP的C++类属的OO设计，并特别强调了socket的SOCK SAP C++包装的设计。SOCK SAP已被移植到许多UNIX平台、以及WinSock网络编程接口上。对这一层面的细节不感兴趣的读者可能会想跳到3.8，在其中讨论的是SOCK SAP包装类的设计之下的一般法则。

3.6.1 SOCK SAP综述

SOCK SAP被设计用于克服3.4描述的socket的局限。使用C++包装来封装socket接口的主要好处是：

- *增强类型安全性*：SOCK SAP在编译时检测许多微妙的应用类型系统违例。
- *可移植性*：SOCK SAP提供了可移植的、平台无关的网络编程接口。
- *易用性*：SOCK SAP极大地减少了花费在较低级网络编程细节上的应用代码数量和开发工作。
- *高效*：SOCK SAP增强了上面所列的软件质量，而又没有牺牲性能[1]。

SOCK SAP类属为应用提供Internet域和UNIX域协议族[6]的OO接口。SOCK SAP由大约12个C++类组成。其总体结构对应于如图3-8所示的*通信服务*、*连接角色*和*通信域*的分类。将图3-4和图3-8进行比较富有启发意义。图3-8中的组件更为简洁，因为它们使用C++包装在依据继承关联的类中封装了多种socket机制的行为。

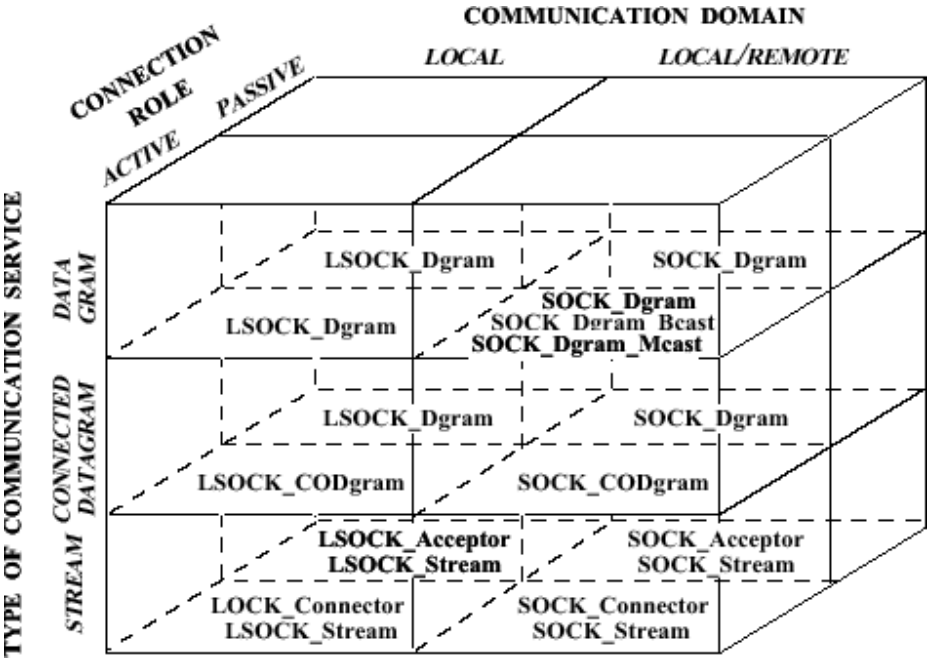

图3-8 SOCK SAP类和通信维度的分类

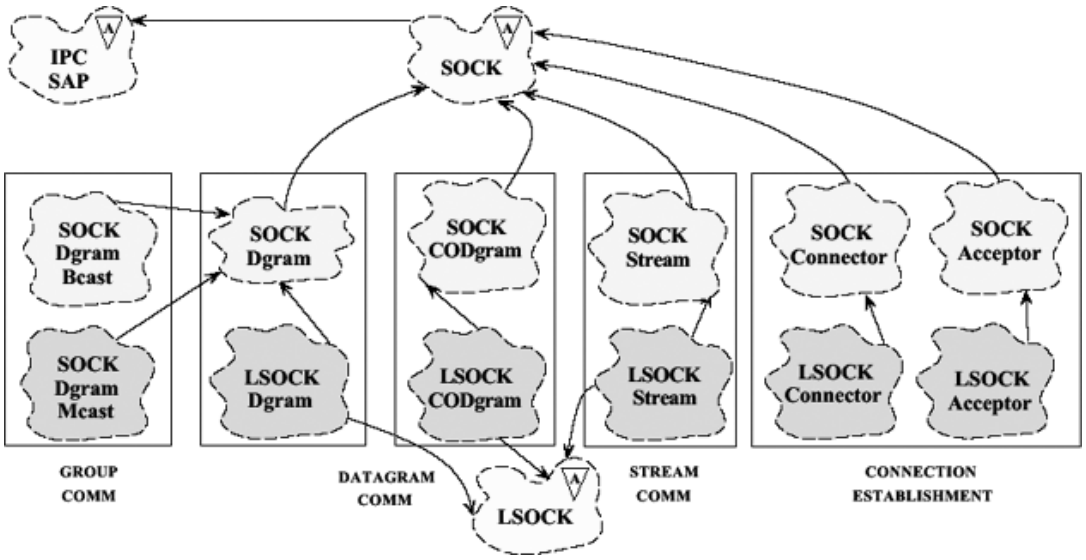

图3-9 SOCK SAP类属

SOCK SAP中的每个类都为组成全部类属的机制的一个子集提供一种抽象接口。多种类型的Internet域和UNIX域socket的功能是通过继承机制从下面描述的适当的类那里获得的。这些类以及它们的关系在图3-9中通过Booch表示法[19]显示。

应用通过继承或实例化图3-9中所示的适当的SOCK SAP子类来访问底层的Internet域或UNIX域socket类型的功能。如下所述，ACE_SOCK* 子类封装Internet域的功能，而ACE_LSOCK* 子类封装UNIX域的功能。

3.6.1.1 基类

IPC SAP、ACE_SOCK和ACE_LSOCK类锚定继承层次，并使应用能够进行后续的派生和代码共享。这些类的对象不能被实例化，因为它们的构造器被声明在类定义的protected区域中。

IPC SAP：该类是进程间通信机制C++包装的IPC SAP层次的根。它提供所有IPC SAP（也就是，SOCK SAP、TLI SAP、SPIPE SAP和FIFO SAP）组件共有的机制。例如，它提供了方法，可将句柄设置为非阻塞模式，或者启用异步的、信号驱动的I/O。

SOCK：该类是SOCK SAP层次的根。它提供所有其他类共有的机制，比如打开和关闭本地通信端点，以及处理选项（像选择socket队列大小及启用组通信）。

LSOCK：该类提供的机制允许应用在本地主机（因而有前缀’L’）上的不相关进程间发送和接收已打开的文件句柄。注意系统V和BSD UNIX都支持这一特性，而Windows NT则不支持。其他类从ACE_LSOCK继承以获得这一功能。

SOCK SAP在网络地址格式和通信语义的基础上区分ACE_LSOCK* 和ACE_SOCK*。特别地，ACE_LSOCK* 类使用UNIX路径名作为地址，并且仅允许机器内的IPC。而另一方面，ACE_SOCK* 类使用Internet协议（IP）地址和端口号，并同时允许机器内和机器间的IPC。

3.6.1.2 连接建立

客户和服务器间的不对称连接角色是通信软件的典型情况。通常，服务器*被动地*侦听客户*主动发*起的连接[20]。下面的面向连接的SOCK SAP类捕捉了被动/主动的连接建立的结构和数据传输关系：

ACE_SOCK_Acceptor和ACE_LSOCK_Acceptor：这两个类是被动地建立新通信端点、以响应主动连接请求的工厂[21]。两者分别生成ACE_SOCK_Stream和ACE_LSOCK_Stream连接端点对象。

ACE_SOCK_Connector和ACE_LSOCK_Connector：这两个类是主动地建立新通信端点的工厂。它们建立与远地端点的连接，并在连接建立时生成适当的*Stream对象。连接可以被同步地或异步地发起。两个工厂分别生成ACE_SOCK_Stream和ACE_LSOCK_Stream连接端点对象。

注意*Acceptor和Connector类不提供发送和接收数据的方法。相反，它们是生成下面描述的*Stream数据传输对象的工厂。使用强类型的工厂接口可以在编译时检测和防止本地和非本地*Stream对象的偶然误用。相反，socket接口仅能在运行时检测这些类型不匹配。

3.6.1.3 流通信

尽管建立连接需要区分主动和被动角色，一旦连接建立，数据就可以根据端点所用的协议以任意的顺序来进行交换。SOCK SAP在下面的类中隔离了数据传输行为：

ACE_SOCK_Stream和ACE_LSOCK_Stream：这些类由上面描述的*Acceptor或*Connector工厂创建。*Stream类为在两个进程间传输数据提供机制。ACE_LSOCK_Stream对象在同一主机上的进程间交换数据；ACE_SOCK_Stream对象在可驻留在不同主机上的进程间交换数据。

被重载的send和recv *Stream方法提供标准的UNIX write和read语义。因而，send或recv分别读或写的字节数可能会少于所请求的字节数。这些“短写”（short-writes）或“短读”（short-reads）之所以发生，是由于OS中的缓冲和传输协议中的流控制。为减少编程工作，*Stream类提供send_n和recv_n方法，允许传输和接收正好n个字节。另外还提供了“分散读”和“集中写”方法，以高效地同时发送和接收多个数据缓冲区。

3.6.1.4 数据报通信

本论文聚焦于面向连接的流通信，但是，socket接口也提供无连接的服务，它使用Internet协议组中的IP和UDP协议。IP和UDP是不可靠的数据报服务，不保证特定的消息会到达它的目的地。无连接服务被用于那些可容忍一定程度的丢失的应用（比如rwho看守[6]）。此外，IP和UDP还提供像TCP和Sun RPC这样的较高级可靠协议的基础。

SOCK SAP socket包装通过下面的类来封装socket数据报通信：

ACE_SOCK_Dgram和ACE_LSOCK_Dgram：这两个类为在运行在本地和/或远地主机上的进程间交换数据报提供机制。不像下面描述的有连接数据报，每个send和recv操作都必须为发送或接收数据报提供服务地址。ACE_LSOCK_Dgram同时继承ACE_SOCK_Dgram和ACE_LSOCK的所有操作。它仅在同一主机上的进程间交换数据报。而ACE_SOCK_Dgram类可以在本地和/或远地主机上的进程间交换数据报。

ACE_SOCK_CODgram和ACE_LSOCK_CODgram：这两个类提供一种“有连接数据报”机制。不像上面所描述的无连接类，这两个类允许send和recv操作在交换数据报时省略服务地址。注意有连接数据报机制只是一种语法上的方便，因为没有其他的语义与数据传输相关联（也就是，数据递送还是不可靠的）。ACE_SOCK_CODgram的机制从 ACE_SOCK基类继承。ACE_LSOCK_CODgram同时继承ACE_SOCK_CODgram和ACE_LSOCK（它提供传递文件句柄的能力）的机制。

3.6.1.5 组通信（Group Communication）

标准的TCP和UDP通信是点对点的。但是，有些应用可从提供组通信的更为灵活的递送机制中获益。因此，下面的类封装了Internet协议组提供的多点传送和广播协议：

ACE_SOCK_Dgram_Mcast：该类提供的机制用于将UDP数据报多点传送给运行在本地子网中的本地和/或远地主机上的进程。该类的接口支持将数据报多点传送给特定的多点传送组。该类还将开发者与有效利用多点传送所需的低级细节屏蔽开来。

ACE_SOCK_Dgram_Bcast：该类提供的机制用于将UDP数据报广播给本地子网中的本地和/或远地主机。该类的接口支持将数据报广播给（1）所有与主机相连的网络接口，或是（2）一个特定的网络接口。该类还将开发者与有效利用广播所需的低级细节屏蔽开来。

ACE_SOCK_Dgram_Bcast类在下面用于将一个消息广播给LAN子网中在指定端口号上侦听的所有服务器：

```
int main (int argc, char *argv[])

{

ACE_SOCK_Dgram_Bcast b_sap (ACE_Addr::sap_any);

char *msg;

u_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");


return 0;

}
```

将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源码行相比较富有启发意义。

3.6.2 网络寻址

设计一种高效而通用的网络寻址接口是困难的。困难源于用一种节省空间且统一的接口来表示不同的网络寻址格式的企图。不同的地址格式要存储以不同大小表示的不同类型的信息。

例如，Internet域的服务（比如ftp或telnet）用两个字段来标识：（1）一个四字节的IP地址（唯一地标识遍及Internet的远地主机），以及（2）一个两字节的端口号（用于将到来的协议数据单元多路分离给适当的客户或是在远地主机上的服务器进程）。相反，UNIX域的socket通过UNIX路径名（长度最多可以到108字节，并只在单个本地主机上有意义）来会合。

现有的由socket接口提供的基于sockaddr的网络寻址结构是麻烦而易错的。它要求开发者明确地把地址结构中的所有字节清零。相反，图3-6所示的SOCK SAP寻址类含有用于操作网络地址的机制。

Addr基类的构造器确保所有的字段被自动地正确初始化。而且，在不同的地址族间存在的不同的大小、格式和功能被封装在派生的地址子类中。这使得开发者更容易对网络寻址方案进行扩展、以包括新的通信域。例如，UNIX Addr子类与ACE_LSOCK*类相关联，ACE_INET_Addr子类与ACE_SOCK*和ACE_TLI*类相关联，还有SPIPE Addr子类与SPIPE SAP中的STREAM管道包装相关联。

3.6.3 TLI SAP

TLI SAP类属提供系统V传输层接口（TLI）的C++接口。TLI的TLI SAP继承层次几乎与socket的SOCK SAP C++包装类相同。主要的差异是TLI和TLI SAP没有定义UNIX域协议族的接口。通过使C++特性（比如缺省参数值和模板）与tirdwr（read/write兼容性STREAM模块）相联合，开发可在编译时参数化、以在socket或TLI网络编程接口上正确运行的应用变得相对直截了当了。

下面的代码演示怎样应用模板来参数化应用所使用的IPC机制。该代码是从[22]描述的分布式日志工具中提取的。在下面的代码中，用一种特定类型的网络编程接口和相应的协议地址类对一个派生自Event Handler的子类进行了参数化：

```
// Logging_Handler header file.

template <class PEER_STREAM>

class Logging_Handler : public Event_Handler

{

public:

Logging_Handler (void);

virtual ?Logging_Handler (void);



virtual int handle_close (int);

virtual int handle_input (int);

virtual int get_handle (void) const

{

       return this->xport_sap.get_handle ();

}



protected:

PEER_STREAM xport_sap;

};
```

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

```
#if defined (MT_SAFE_SOCKETS)

typedef ACE_SOCK_Stream PEER_STREAM;

#else

typedef ACE_TLI_Stream PEER_STREAM;

// Logging application.

#endif // MT_SAFE_SOCKETS.
```

```
class Logging_Handler :

public Logging_Handler<PEER_STREAM>

{

// ...

};
```

在开发运行在多种OS平台上的可移植应用时，模板所提供的增强的灵活性是有用的。例如，在跨越SunOS平台的多种变种时，能使用网络编程接口来对应用进行参数化的能力是必需的。特别地，SunOS 5.2中的socket实现不是线程安全的，而SunOS 4.x中的TLI实现含有许多严重的缺陷。

TLI SAP还将应用与TLI接口的许多特性屏蔽开来。例如，在qlen > 1的并发服务器中，[5]，ACE_TLI_Acceptor类的accept方法封装了处理t_listen和t_accept的非直观而又易错的行为所需的微妙的应用级代码。accept方法被动地建立客户连接请求。通过使用C++缺省参数值，对于基于TLI SAP和基于SOCK SAP的应用来说，调用accept方法的标准方法在语法上都是一样的。

3.6.4 SPIPE SAP和FIFO SAP

SPIPE SAP类属为已安装STREAM管道和connld[17]提供一种C++包装接口。SPIPE SAP继承层次是SOCK SAP和TLI SAP所用的层次的镜像。它提供与SOCK SAP ACE_LSOCK* 类（它们封装的是UNIX域的socket）相类似的功能。但是，SPIPE SAP比ACE_LSOCK* 接口更灵活，因为它使STREAM模块可以分别被“压入”或“弹出”SPIPE SAP端点。SPIPE SAP还支持在运行在同一主机上的进程和/或线程间的字节流和按优先级排序的消息数据的双向递送[16]。

FIFO SAP类属封装UNIX FIFO机制。

3.7 SOCK SAP C++包装类编程

这一部分通过使用ACE SOCK SAP C++包装开发一个客户/服务器流式应用来对它们进行演示。该应用是[1]中描述的ttcp程序的简化版本。为了比较，该应用还用socket进行了编写。为保持简短，例子中的大多数错误检查都被省略了。自然，健壮的程序应该检查库和系统调用的返回值。

图3-10和图3-11介绍一个用C编写的客户/服务器程序，它使用Internet域的socket和select来实现流应用。图3-11所示的服务器创建一个被动模式的侦听者socket，并等待客户与它连接。一旦连接，服务器接收来自客户的数据，并将其显示在它的标准输出流上。图3-10所示的客户端建立一个到服务器的TCP连接，并将它的标准输入流通过连接进行传输。客户使用非阻塞连接来限制它等待连接被接受或拒绝的时间数量。

大多数的返回值错误检查被省略了，以节省空间。但是，即使是要使这个简单的例子正确工作，所有socket初始化、网络寻址和流控制细节都必须被显式地编写；注意到这一点富有启发意义。而且，图3-10和3-11中的代码对于不同时支持socket和select的平台来说是不可移植的。

图3-12和3-13使用SOCK SAP来重新实现C版本的客户/服务器程序。该SOCK SAP程序实现了与图3-10和图3-11所介绍的相同的功能。与基于socket的C实现相比，SOCK SAP C++程序展示了下列好处：

增强的清晰性：例如，网络寻址和主机定位由图3-6所示的Addr类来处理，它隐藏了在图3-10和3-11中必须显式编写的微妙而又易错的细节。而且，非阻塞连接建立的低级细节是由SOCK Connector工厂来完成的。此外，模板*traits*的使用使在对参数化函数进行实例化时必须指定的类型参数的数目减到了最少。

增强的类型安全性：例如，ACE_SOCK_Acceptor和ACE_SOCK_Connector连接工厂创建ACE_SOCK_Stream对象，从而防止了在运行时发生图3-3所示的类型错误。

更小的程序大小：使主动和被动连接建立局限在ACE_SOCK_Acceptor和ACE_SOCK_Connector连接工厂中大量地减少了代码的行数。此外，为构造器和方法参数提供的缺省值减少了常见的使用模式所需的参数数目。

增强的可移植性：例如，由于使用了模板traits，在socket和TLI之间切换只需要将客户中的

```
send_data <ACE_TLI_Connector> (s_addr);
```

改变为

```
send_data <ACE_SOCK_Connector> (s_addr);
```

以及将服务器中的

```
recv_data<ACE_SOCK_Acceptor> (s_addr);
```

改变为

```
recv_data<ACE_TLI_Acceptor> (s_addr);
```

如3.8所示，可用条件编译指令来进一步使通信软件与对特定类型的网络编程接口的依赖去耦合。

```
#define PORT_NUM 10000
#define TIMEOUT 5

/* Socket client. */

void send_data (const char host[], u_short port_num)
{
struct sockaddr_in peer_addr;
struct hostent *hp;
char buf[BUFSIZ];
int s_sd, w_bytes, r_bytes, n;

/* Create a local endpoint of communication */
s_sd = socket (PF_INET, SOCK_STREAM, 0);

/* Set s_sd to non-blocking mode. */
n = fcntl (s_sd, F_GETFL, 0);
fcntl (s_sd, F_SETFL, n | O_NONBLOCK);

/* Determine IP address of the server */
hp = gethostbyname (host);

/* Set up address information to contact server */
memset ((void *) &peer_addr, 0, sizeof peer_addr);
peer_addr.sin_family = AF_INET;
peer_addr.sin_port = port_num;
memcpy (&peer_addr.sin_addr,
hp->h_addr, hp->h_length);

/* Establish non-blocking connection server. */
if (connect (s_sd, (struct sockaddr *) &peer_addr,
       sizeof peer_addr) == -1)
{
       if (errno == EINPROGRESS)
       {
```

```
            struct timeval tv = {TIMEOUT, 0};

            fd_set rd_sds, wr_sds;

            FD_ZERO (&rd_sds);

            FD_ZERO (&wr_sds);

            FD_SET (s_sd, &wr_sds);

            FD_SET (s_sd, &rd_sds);


            /* Wait up to TIMEOUT seconds to connect. */

            if (select (s_sd + 1, &rd_sds, &wr_sds, 0, &tv) <= 0)

                        perror ("connection timedout"), exit (1);


            // Recheck if connection is established.

            if (connect (s_sd, (struct sockaddr *) &peer_addr,

                        sizeof peer_addr) == -1 && errno != EISCONN)

                              perror ("connect failed"), exit
                              (1);

      }

}


/* Send data to server (correctly handles

"short writes" due to flow control) */

while ((r_bytes = read (0, buf, sizeof buf)) > 0)

      for (w_bytes = 0; w_bytes < r_bytes; w_bytes += n)

            n = write (s_sd, buf + w_bytes, r_bytes - w_bytes);


/* Close down the connection. */

close (s_sd);

}


int main (int argc, char *argv[])

{

char *host = argc > 1 ? argv[1] : "ics.uci.edu";

u_short port_num = htons (argc > 2 ? atoi (argv[2]) : PORT_NUM);


/* Send data to the server. */

send_data (host, port_num);
```

```
return 0;
}
```

图3-10 基于socket的客户例子

```
#define PORT_NUM 10000

/* Socket server. */

void recv_data (u_short port_num)
{
struct sockaddr_in s_addr;
int s_sd;

/* Create a local endpoint of communication */
s_sd = socket (PF_INET, SOCK_STREAM, 0);

/* Set up the address information for a server */
memset ((void *) &s_addr, 0, sizeof s_addr);
s_addr.sin_family = AF_INET;
s_addr.sin_port = port_num;
s_addr.sin_addr.s_addr = INADDR_ANY;

/* Associate address with endpoint */
bind (s_sd, (struct sockaddr *) &s_addr, sizeof s_addr);

/* Make endpoint listen for service requests */
listen (s_sd, 5);

/* Performs the iterative server activities */
for (;;)
{
       char buf[BUFSIZ];
```

```
        int r_bytes, n_sd;
        struct sockaddr_in peer_addr;
        int peer_addr_len = sizeof peer_addr;
        struct hostent *hp;

        /* Create a new endpoint of communication */
        while ((n_sd = accept (s_sd, &peer_addr,
                &peer_addr_len)) == -1 && errno == EINTR)
                        continue;

        hp = gethostbyaddr (&peer_addr.sin_addr, peer_addr_len, AF_INET);
        printf ("client %s\n", hp->h_name);

        /* Read data from client (terminate on error) */
        while ((r_bytes = read (n_sd, buf, sizeof buf)) > 0)
                write (1, buf, r_bytes);

        /* Close the new endpoint (listening endpoint remains open) */
        close (n_sd);
}

/* NOTREACHED */
}

int main (int argc, char *argv[])
{
u_short port_num = htons (argc > 1 ? atoi (argv[1]) : PORT_NUM);

// Receive data from clients.
recv_data (port_num);

return 0;
}
```

图3-11 基于socket的服务器例子

```
static const int PORT_NUM = 10000;
static const int TIMEOUT = 5;

// SOCK_SAP Client.

template <class CONNECTOR>
void send_data (CONNECTOR::PEER_ADDR peer_addr)
{
// Data transfer object.
CONNECTOR::PEER_STREAM peer_stream;

// Establish connection without blocking.
CONNECTOR connector (peer_stream, peer_addr, ACE_NONBLOCK);

if (peer_stream.get_handle () == -1)
{
       // If non-blocking connection is in progress,
              // wait up to TIMEOUT seconds to complete.
       Time_Value timeout (TIMEOUT);
       if (errno != EWOULDBLOCK ||
              connector.complete (peer_stream, peer_addr, &timeout) == -1)
                     perror ("connector"), exit (1);
}

// Send data to server (send_n() handles
// "short writes" correctly).
char buf[BUFSIZ];

for (int r_bytes; (r_bytes = read (0, buf, sizeof buf)) > 0;)
       peer_stream.send_n (buf, r_bytes);

// Explicitly close the connection.
peer_stream.close ();
}
```

```
int main (int argc, char *argv[])
{
char *host = argc > 1 ? argv[1] : "ics.uci.edu";
u_short port_num = htons (argc > 2 ? atoi (argv[2]) : PORT_NUM);

// Address of the server.
ACE_INET_Addr s_addr (port_num, host)

// Use SOCK SAP wrappers on client's side.
send_data <ACE_SOCK_Connector> (s_addr);

return 0;
}
```

图3-12 基于SOCK SAP的客户例子

```
static const int PORT_NUM = 10000;

// SOCK_SAP Server.

template <class ACCEPTOR>
void recv_data (ACCEPTOR::PEER_ADDR s_addr)
{
// Factory for passive connection establishment.
ACCEPTOR acceptor (s_addr);

// Data transfer object.
ACCEPTOR::PEER_STREAM peer_stream;

// Remote peer address.
ACCEPTOR::PEER_ADDR peer_addr;

// Performs iterative server activities.
for (;;)
```

```
{
	// Create a new STREAM endpoint
	// (automatically restarted if errno == EINTR).
	acceptor.accept (peer_stream, &peer_addr);

	printf ("client %s\n", peer_addr.get_host_name ());

	// Read data from client (terminate on error).
	char buf[BUFSIZ];

	for (int r_bytes = 0;;)
	{
		r_bytes = peer_stream.recv (buf, sizeof buf);
		if (r_bytes > 0)
			write (1, buf, r_bytes);
		else
			break;
	}

	// Close peer_stream endpoint
	// (acceptor endpoint stays open).
	peer_stream.close ();
}

/* NOTREACHED */
}

int main (int argc, char *argv[])
{
u_short port_num = argc == 1 ? PORT_NUM : atoi (argv[1]);

// Port for the server.
ACE_INET_Addr s_addr (port_num);

// Use Socket wrappers on server's side.
recv_data<ACE_SOCK_Acceptor> (s_addr);
```

```
return 0;
}
```

图3-13 基于SOCK SAP的服务器例子

3.8 socket包装设计原则

这一部分描述下列贯穿SOCK SAP类属所应用的设计原则：

- 在编译时强制实现类型安全性
- 允许受控的类型安全性违例
- 为常见情况进行简化
- 用层次类属替代一维的接口
- 通过参数化类型增强可移植性
- 内联性能关键的方法
- 定义辅助类隐藏易错细节

尽管这些原则已广为人知，并被广泛应用于像图形用户接口这样的领域中，但在通信软件领域中它们还没有被那么广泛地应用。

3.8.1 在编译时强制实现类型安全性

在3.4讨论的socket的若干局限源于在其接口中缺乏类型安全性。为强制实现类型安全性，SOCK SAP确保它所有的对象都通过构造器适当地初始化。此外，为防止偶然的类型安全性违例，只允许对SOCK SAP对象进行合法的操作。后一点已在图3-14所示的echo_server的SOCK SAP修订版进行演示。该版本更正了图3-3中所标识出的的socket和C的问题。因为SOCK SAP类是强类型的，非法操作在编译时、而不是运行时被拒绝。例如，不可能在ACE_SOCK_Acceptor连接工厂上调用recv或send方法，因为这些方法不是其接口的一部分。同样地，返回值只用于传达操作的成功或失败，从而减少了在赋值表达式中误用的潜在可能性。

```
int echo_server (ACE_INET_Addr s_addr)
```

```
{
// Initialize the passive mode server.
ACE_SOCK_Acceptor acceptor (s_addr);

// Data transfer object.
ACE_SOCK_Stream peer_stream;

// Client remote address object.
ACE_INET_Addr peer_addr;

// Accept a new connection.
if (acceptor.accept (peer_stream, &peer_addr) != -1)
{
       char buf[BUFSIZ];
       for (size_t n; peer_stream.recv (buf, sizeof buf, n) > 0;)
              // Handles "short-writes."
              if (peer_stream.send_n (buf, n) != n)
                           // Remainder omitted.
}
}
```

图3-14 Echo服务器的SOCK SAP修订版

3.8.2 允许受控的类型安全性违例

该原则通过IPC SAP根类所提供的get_handle和set_handle方法进行例示。这两个方法分别提取和指派底层的句柄。通过提供get_handle和set_handle，IPC SAP允许应用在必须与需要句柄的UNIX系统调用（比如select）协作时直接绕过IPC SAP的类型检查机制。陈述此原则的另一方式是“让SOCK SAP的正确使用更容易，不正确使用更困难，但不是不可能以类设计者没有预见到的方式来使用它。”

3.8.3 为常见情况进行简化

此原则以下列途径应用于ACE C++ socket包装类中：

为常用方法参数提供缺省值：例如，ACE_SOCK_Connector构造器有六个参数：

```
ACE_SOCK_Connector(ACE_SOCK_Stream &new_stream,
       const ACE_SOCK_Addr &remote_sap,
       ACE_Time_Value *timeout = 0,
       const ACE_SOCK_Addr &local_sap = (ACE_SOCK_Addr &) Addr::sap_any,
       int protocol_family = PF_INET,
       int protocol = 0);
```

但是，在调用与调用间通常只有前两个是变化的：

```
ACE_SOCK_Stream stream;

// Compiler supplies default values.
ACE_SOCK_Connector con (stream, ACE_INET_Addr (port, host));
// ...
```

因此，为简化编程，在ACE_SOCK_Connector中给出了其他参数的缺省值，以使程序员无需每次都提供它们。

定义节俭的接口：此原则让使用一种特定抽象的代价局部化。IPC SAP限定应用开发者所必须记忆的细节数量。它为开发者提供群集的类，执行不同类型的通信（比如面向连接的 vs. 无连接的）和不同的连接角色（比如主动的 vs. 被动的）。为减少犯错的机会，ACE_SOCK_Acceptor类只允许为程序而应用的操作扮演被动角色，而ACE_SOCK_Connector只允许为程序而应用的操作扮演主动角色。此外，与使用高度通用的UNIX sendmsg/recvmsg函数相比，使用ACE_SOCK_SAP来发送和接收打开的文件句柄有着一个简单得多的接口。例如，使用ACE_LSOCK* 类来传递socket句柄是非常简洁的：

```
ACE_LSOCK_Stream stream;
ACE_LSOCK_Acceptor acceptor ("/tmp/foo");

// Accept connection.
acceptor.accept (stream);

// Pass the Socket handle back to caller.
stream.send_handle (stream.get_handle ());
```

与此相比较，使用socket接口来实现所需的代码：

```
int n_sd;
int u_sd;
sockaddr_un addr;
u_char a[2];
iovec iov;
msghdr send_msg;

u_sd = socket (PF_UNIX, SOCK_STREAM, 0);

memset ((void *) &addr, 0, sizeof addr);
addr.sun_family = AF_UNIX;
strcpy (addr.sun_path, "/tmp/foo");

bind (u_sd, &addr, sizeof addr.sun_family + strlen ("/tmp/foo"));
listen (u_sd, 5);

// Accept connection.
n_sd = accept (u_sd, 0, 0);

// Sanity check.
a[0] = 0xab; a[1] = 0xcd;

iov.iov_base = (char *) a;
iov.iov_len = sizeof a;

send_msg.msg_iov = &iov;
send_msg.msg_iovlen = 1;
send_msg.msg_name = (char *) 0;
send_msg.msg_namelen = 0;
send_msg.msg_accrights = (char *) &n_sd;
send_msg.msg_accrightslen = sizeof n_sd;

// Pass the Socket handle back to caller.
sendmsg (n_sd, &send_msg, 0);
```

将多个操作组合进单一操作：创建一个传统的被动模式socket需要多个调用：

```
int s_sd = socket (PF_INET, SOCK_STREAM, 0);

sockaddr_in addr;

memset (&addr, 0, sizeof addr);

addr.sin_family = AF_INET;

addr.sin_port = htons (port);

addr.sin_addr.s_addr = INADDR_ANY;

bind (s_sd, &addr, addr_len);

listen (s_sd);

// ...
```

相反，ACE_SOCK_Acceptor是一个用于被动连接建立的工厂，它的构造器执行创建被动模式侦听者端点所需的socket调用：socket、bind和listen。因此，应用只需简单地如下编写：

```
ACE_INET_Addr addr (port);

ACE_SOCK_Acceptor acceptor (addr);
```

3.8.4 用层次类属替代一维接口

此原则涉及使用层次相关的类属来重构现有的一维socket接口（如图3-9所示）。用于构造SOCK SAP类属的准则涉及标识、群集和封装相关的socket函数，以最大化复用和类组件的共享。

继承支持SOCK SAP类属的不同功能子集。例如，不是所有的操作系统都支持传递打开的文件句柄（例如，Windows NT）。因而，可以在继承层次中省略ACE_LSOCK类（在3.6.1中描述），而不会影响SOCK SAP设计中的其他类的接口。

继承还增强代码复用和改善模块性。基类表示类属组件间的*相似性*，而派生类表示*差异性*。例如，IPC SAP设计向着IPC SAP和SOCK SAP基类中的继承层次的“根”放置了共享的机制，包括用于打开/关闭和设置/取回底层socket句柄的操作，以及对所有派生的SOCK SAP类来说都通用的特定的选项管理函数。而位于继承层次“底部”的子类实现专门的操作，为所提供的通信类型（比如流 vs. 数据报通信，或本地 vs. 远地通信）而进行定制。这样的方法避免了不必要的代码重复，因为更为专门的派生类会复用继承层次的根上所提供的更为通用的机制。

3.8.5通过参数化类型增强可移植性

```
template <class ACCEPTOR>

int echo_server (ACCEPTOR::PEER_ADDR s_addr)
```

```
{
// Initialize the passive mode server.
ACCEPTOR acceptor (s_addr);

// Data transfer object.
ACCEPTOR::PEER_STREAM peer_stream;

// Remote address object.
ACCEPTOR::PEER_ADDR peer_addr;

// Accept a new connection.
if (acceptor.accept (peer_stream, &peer_addr) != -1)
{
       char buf[BUFSIZ];
       for (size_t n; peer_stream.recv (buf, sizeof buf, n) > 0;)
              if (peer_stream.send_n (buf, n) != n)
                           // Remainder omitted.
}
}
```

图3-15 Echo服务器的模板版本

通过允许经由参数化类型来整个地替换网络编程接口，用C++类（而不是独立的C函数）包装socket有助于改善可移植性。参数化类型使应用与对特定的网络编程接口的依赖去耦合。图3-15通过将echo_server修改成为C++函数模板来演示这一技术。取决于底层OS平台的特定属性（比如它更为高效地实现了TLI还是socket），echo_server可以通过SOCK SAP 或TLI SAP类来实例化。如下所示：

```
// Conditionally select IPC mechanism.
#if defined (USE_SOCKETS)
typedef ACE_SOCK_Acceptor ACCEPTOR;
#else // USE_TLI
typedef ACE_TLI_Acceptor ACCEPTOR;
#endif // USE_SOCKETS.

const int PORT_NUM = 10000;
```

```
int main (void)

{

// ...


// Invoke the echo_server with appropriate

// network programming interfaces. Note the

// use of template traits for addr class.

ACCEPTOR::PEER_ADDR addr (PORT_NUM);

echo_server<ACCEPTOR> (addr);

}
```

一般而言，比起其他一些传统方法，比如实现多个版本或在源码中到处使用杂乱的条件编译指令，使用参数化类型要更没有侵入性并且更可扩展一点。

例如，SOCK SAP和TLI SAP类提供同样的OO接口（在图3-7中描述）。特定的OS平台可能拥有不同的底层网络编程接口，比如有socket，但没有TLI，或反之亦然。使用IPC SAP，应用可被透明地编写，通过SOCK SAP或TLI SAP来进行参数化。C++模板支持“类型一致”（type conformance）的一种松散形式，并不强迫接口包含*所有*可能的功能。相反，模板被用于参数化仔细设计的应用代码，用以调用不同通信抽象的通用方法子集（例如，open、close、send、recv，等等）。

模板提供的类型抽象改善了支持不同网络编程接口（比如socket或TLI）的平台间的可移植性。例如，对于开发跨越多种SunOS平台的应用来说，对网络编程接口的参数化是有用的。SunOS 5.2中的socket实现不是线程安全的，而SunOS 4.x中的TLI实现含有许多严重的缺陷。

3.8.6 内联性能关键的方法

为鼓励开发者用C++包装替换现有的低级网络编程接口，SOCK SAP实现必须高效地运作。为确保这一点，在关键的性能路径上的方法（比如ACE_SOCK_Stream recv和send方法）被指定为C++内联函数，以消除运行时函数调用开销。内联在时间和空间上都是高效的，因为这些方法非常短小（每个方法大约2或3行）。内联的使用意味着应该保守地使用虚函数，因为大多数当代的C++编译器不能充分地把虚函数开销优化掉。

3.8.7 定义辅助类隐藏易错的编程细节

socket寻址的C接口是难以使用和易错的。很容易忽略sockaddr_in的清零或把端口号转换到网络字节序。为使应用与这些低级细节相屏蔽，IPC SAP定义了Addr类层次（如图3-6所示）。该层次通过类型安全的C++接口支持若干不同的网络寻址格式。Addr层次消除了与直接使用基于C的struct sockaddr数据结构族相关联的常见编程错误。例如，ACE_INET_Addr的构造器自动将sockaddr寻址结构清零，并将端口号转换为网络字节序。如下所示：

```
class ACE_INET_Addr : public ACE_Addr

{

public:

ACE_INET_Addr::ACE_INET_Addr (u_short port, long ip_addr = 0)

{

       memset (&this->inet_addr_, 0, sizeof this->inet_addr_);

       this->inet_addr_.sin_family = AF_INET;

       this->inet_addr_.sin_port = htons (port);

       memcpy (&this->inet_addr_.sin_addr, &ip_addr, sizeof ip_addr);

}


private:

sockaddr_in inet_addr_;

};
```

3.9结束语

IPC SAP提供一个OO C++包装族，封装在当代的操作系统上可用的标准本地和远地IPC机制。通过使编写正确、紧凑、可移植和高效的代码变得更为容易，这些封装的接口简化了通信软件的开发。此外，包装方法还便利了向C++的有组织的迁移，通过（1）逐步对开发者进行OO设计原理教学，以及（2）有效利用现有的非C++语言的代码库。本论文通过描述使用C++实现IPC SAP的若干优点和缺点、以及概述将来进一步对IPC SAP的使用进行探究的论文来作为结束。

使用C++的优点和缺点：用C++开发包装的主要优点包括：

- *封装变种*：类隐藏寻址格式中的差异，比如Internet vs. UNIX域寻址。此外，它们还在不同的类中封装不同的接口行为。例如，ACE_SOCK_Acceptor对象的接口为服务器操作而特别作了剪裁。

- *增强功能子集划分*：继承使定义功能子集变得更为容易。例如，ACE_LSOCK类可在不支持文件句柄传递的操作系统上被忽略。

- *更高的可移植性*：模板使得不同的IPC机制可被参数化进应用，从而改善了跨平台可移植性。

C++的一个缺点是它缺少可移植的异常处理。在适当使用时，C++异常处理有助于简化错误恢复，并改善类型安全性。例如，如果ACE_INET_Addr构造器因为远地地址没有对应到有效的主机而失败的话，就可以扔出一个异常。但是，如果没有C++异常处理，就有可能在没有对IPC SAP对象进行正常初始化的情况下开始使用它。该问题将在ANSI/ISO C++异常处理机制可用于大多数OS平台时得以解决。

当前状况和未来主题：IPC SAP可在ACE构架[2]中找到。ACE支持的OS平台包括Win32（使用MSVC++和Borland C++的Win2000、WinNT 3.5.x、4.x、Win95和WinCE）、大多数版本的UNIX（SunOS 4.x和5.x；SGI IRIX 5.x和6.x；HP-UX 9.x、10.x和11.x；DEC UNIX 3.x和4.x、AIX 3.x和4.x、DG/UX、Linux、SCO、UnixWare、NetBSD和FreeBSD）、实时操作系统（VxWorks、Chorus、LynxOS和pSoS），以及MVS OpenEdition。

ACE已被用于许多大学和公司的研究和开发项目。例如，ACE已在波音被用于构建实时航空控制系统[23]；在Bellcore[22]、爱立信[24]、摩托罗拉[25]和朗讯被用于电信系统；在西门子[26]和柯达[27]被用于医学成像系统；以及在SAIC/DARPA的分布式模拟系统。它还被广泛地用于研究项目和课堂教学。

本论文中描述的所有源代码都可以在http://www.cs.wustl.edu/~schmidt/ACE.html找到。http://www.cs.wustl.edu/~schmidt/ACE-users.html描述使用ACE的许多项目。此外，comp.soft-sys.ace是专用于ACE相关主题的USENET新闻组。

参考文献

[1] 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.

[2] 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.

[3] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley, 1996.

[4] Sun Microsystems, *Network Interfaces Programmer's Guide*, Chapter 6 (TLI Interface) ed., 1992.

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

[6] W.R.Stevens, *UNIX Network Programming, Second Edition*. Englewood Cliffs, NJ: Prentice Hall, 1997.

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

[8] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.2 ed., Feb. 1998.

[9] D. Box, *Essential COM*. Addison-Wesley, Reading, MA, 1997.

[10] S. Mungee, N. Surendran, and D. C. Schmidt, "The Design and Performance of a CORBA Audio/Video Streaming Service," in *submitted to the Hawaiian International Conference on System Sciences*, Jan. 1999.

[11] A. Gokhale and D. C. Schmidt, "Measuring the Performance of Communication Middleware on High-Speed Networks," in *Proceedings of SIGCOMM '96*, (Stanford, CA), pp. 306-317, ACM, August 1996.

[12] OSI Special Interest Group, *Transport Provider Interface Specification*, December 1992.

[13] OSI Special Interest Group, *Data Link Provider Interface Specification*, December 1992.

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

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

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

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

[18] A. Stepanov and M. Lee, "The Standard Template Library," Tech. Rep. HPL-94-34, Hewlett-Packard Laboratories, April 1994.

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

[20] D. C. Schmidt, "Acceptor and Connector: Design Patterns for Actively and Passively Initializing Network Services," in *Workshop on Pattern Languages of Object-Oriented Programs at ECOOP '95*, (Aarhus, Denmark), August 1995.

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

[22] 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.), pp. 529－545, Reading, MA: Addison-Wesley, 1995.

[23] T. H. Harrison, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-time CORBA Event Service," in *Proceedings of OOPSLA '97*, (Atlanta, GA), ACM, October 1997.

[24] 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.

[25] 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.

[26] P. Jain and D. C. Schmidt, "Service Configurator: A Pattern for Dynamic Configuration of Services," in *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, USENIX, June 1997.

[27] 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.